FreeRTOS

[STM32 FreeRTOS] 자료 Review#1


이지훈
2020. 2. 26. 23:31

※ 내용에 오류가 있을 수 있습니다. 오류에 대해서는 Feedback 부탁 드리겠습니다.

https://github.com/eziya/STM32F4_HAL_FREERTOS_LAB

eziya/STM32F4_HAL_FREERTOS_LAB
FreeRTOS Testing for STM32F4. Contribute to eziya/STM...
github.com

ST 에서 제공하는 자료들을 기반으로 FreeRTOS 의 기본 특성에 대해서 정리하여 보고 해당 자료의 Lab 코드들을 테스트해 보고자 합니다.

API 를 살펴보면 Task 관리, 스케쥴링, 동기화, 타이머 기능등을 제공하는 것을 확인할 수 있다.

API category	FreeRTOS API	Description
Task creation	xTaskCreate	Creates a new task
	vTaskDelete	Deletes a task
Task control	vTaskDelay	Task delay
	vTaskPrioritySet	Sets task priority
	uxTaskPriorityGet	Get task priority
	vTaskSuspend	Suspends a task
	vTaskResume	Resumes a task
Kernel control	vTaskStartScheduler	Starts kernel scheduler
	vTaskSuspendAll	Suspends all tasks
	xTaskResumeAll	Resumes all tasks
	taskYIELD	Forces a context switch
	taskENTER_CRITICAL taskEXIT_CRITICAL	Enter(Exit from) a critical section (When entering, it stops context switching)

API category	FreeRTOS API	Description
Message Queue	xQueueCreate	Creates a queue
	xQueueSend	Sends data to queue
	xQueueReceive	Receive data from the queue
Semaphore Mutex	xSemaphoreCreateBinary	Creates a binary semaphore
	xSemaphoreCreateCounting	Creates a counting semaphore
	xSemaphoreCreateMutex	Creates a mutex semaphore
	xSemaphoreTake	Semaphore take
	xSemaphoreGive	Semaphore give
Timers	xTimerCreate	Creates a timer
	xTimerStart	Starts a timer
	xTimerStop	Stops a timer

FreeRTOS API 와 CMSIS(Cortex Microcontroller Software Standard)-RTOS API 를 모두 사용할 수 있으며 CMSIS-RTOS API 를 사용하여 개발하는 경우, 하위 RTOS 가 FreeRTOS 가 아닌 다른 CMSIS-RTOS API 를 지원하는 타 RTOS 이식하는 경우 이식성이 좋아질 수 있다.

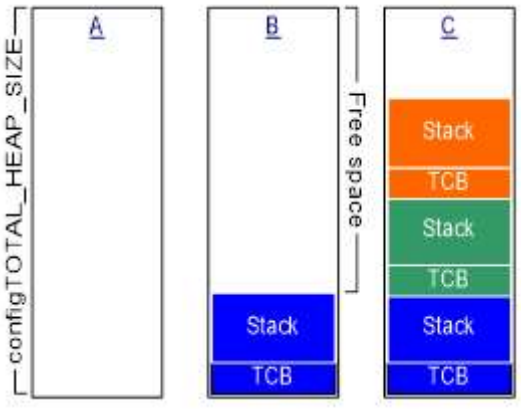
FreeRTOS 설정관련 옵션들은 FreeRTOSConfig.h 파일에 선언되어 있으며 주요 항목은 아래와 같다.

Config option	Description
configUSE_PREEMPTION	Enables Preemption
configCPU_CLOCK_HZ	CPU clock frequency in hertz
configTICK_RATE_HZ	Tick rate in hertz
configMAX_PRIORITIES	Maximum task priority
configTOTAL_HEAP_SIZE	Total heap size for dynamic allocation
configLIBRARY_LOWEST_INTERRUPT_PRIORITY	Lowest interrupt priority (0xF when using 4 cortex preemption bits)
configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY	Highest thread safe interrupt priority (higher priorities are lower numeric value)

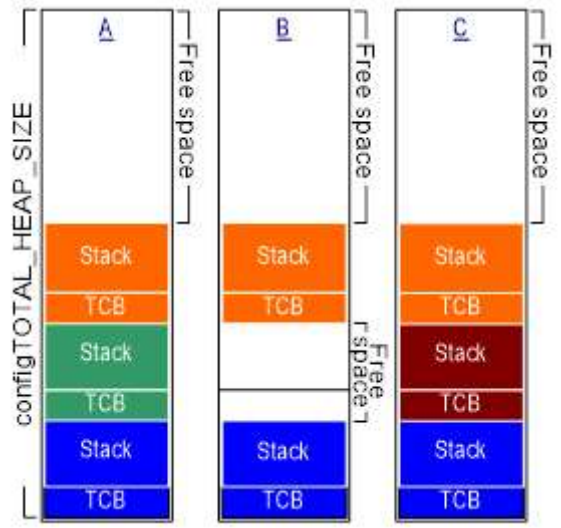
Tickless 모드는 System Tick 인터럽트를 중지시키고 MCU 를 low power 모드로 진입시켜 저전력 동작을 수행할 수 있도록 하는 모드이다. (configUSE_TICKLESS_IDLE 설정)
Sleep 모드에 진입한 경우에는 System 인터럽트나 이벤트를 이용하여 Wakeup 할 수 있다.

FreeRTOS 는 별도의 Heap 영역을 사용하며 해당 Heap 영역내에 Task, 세마포어, 큐 등을 할당한다.
따라서, Total Heap 의 크기는 사용하는 Task 나 동기화 컴포넌트의 개수와 크기에 따라 변경되어야 한다.

Heap 영역을 관리하는 방식은 현재 5가지 정도의 방식을 지원한다.
Heap1 방식은 할당한 메모리를 Free 하지 않는 가장 단순한 방식으로 Task 가 생성되거나 종료되지 않는 경우에 적합할 수 있다. 따라서 pvPortMalloc() 만을 사용하고 pvPortFree() 는 사용하지 않는다.
아래 그림에서 생성된 Task 들은 Free 할 수 없다.



Heap2 방식은 할당한 메모리를 Free 할 수 있지만 인접 Free 블록들을 묶어주지 않기 때문에 단편화가 발생할 수 있다.



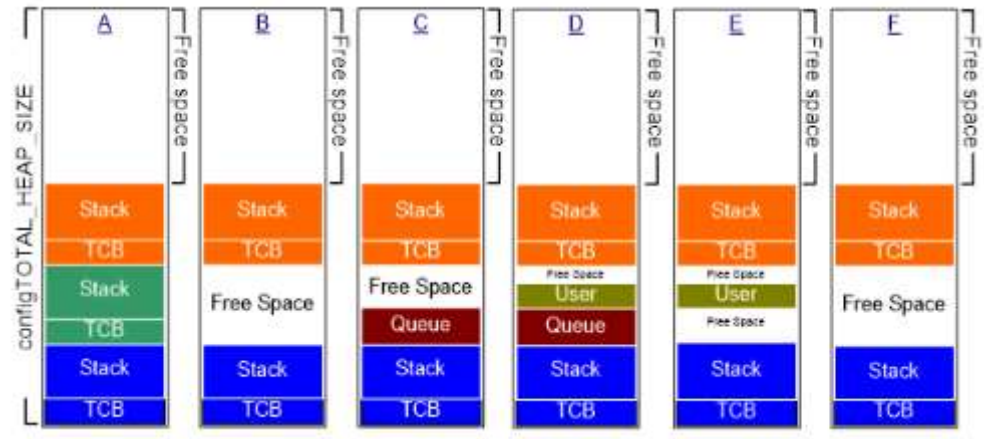
Heap3 방식은 아래 FreeRTOS 문서를 살펴보면 pvPortMalloc() 과 pvPortFree() 를 사용하지 않고 표준 함수인 malloc() 과 free() 를 사용한다고 기술되어 있다. 따라서 FreeRTOS 전용 힙 영역을 구성하지 않기 때문에 configTOTAL_HEAP_SIZE 설정은 효과가 없으며 FreeRTOS 가 사용하는 전체 Heap 크기를 알 수 없다.

heap_3

heap_3.c는 표준 라이브러리 함수인 malloc()과 free()를 사용하기 때문에 링커 구성에서 힙의 크기를 정의합니다. 따라서 configTOTAL_HEAP_SIZE 설정은 아무런 효과가 없습니다.

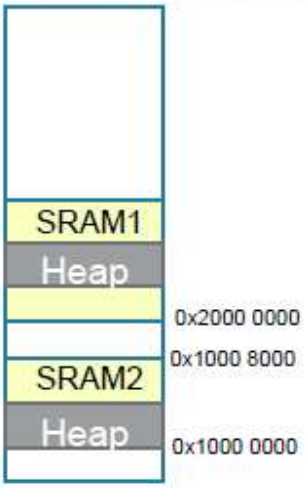
heap_3은 FreeRTOS 스케줄러를 일시적으로 중지하여 malloc()과 free()를 스레드 세이프로 만듭니다. 스레드 세이프와 스케줄러 일시 중지에 대한 자세한 내용은 리소스 관리 단원을 참조하십시오.

Heap4 방식은 Heap2 방식과 유사하며 다만 근처 Free 블록을 묶어주기 때문에 단편화 문제를 해결할 수 있다. 아래 그림을 보면 Queue와 User 리소스 해제후에 인접 Free 영역이 합쳐지면서 단편화 문제가 발생하지 않는 것을 확인할 수 있다. Heap2 방식이었다면 해당 영역은 3개의 Free 블록으로 쪼개져 있어서 각각의 영역보다 큰 메모리 할당이 불가하다.



Heap5 방식은 Heap4 방식과 동일하다 다만 Heap5 는 서로 분리된 메모리 영역에서 메모리를 할당할 수 있는데 예를 들어서 STM32L476 과 같이 내부 SRAM1 과 SRAM2 로 분리된 경우에 사용될 수 있다.

- **Heap_5.c (2/2)**
  - An example for STM32L476 device with SRAM1 and SRAM2 areas.:



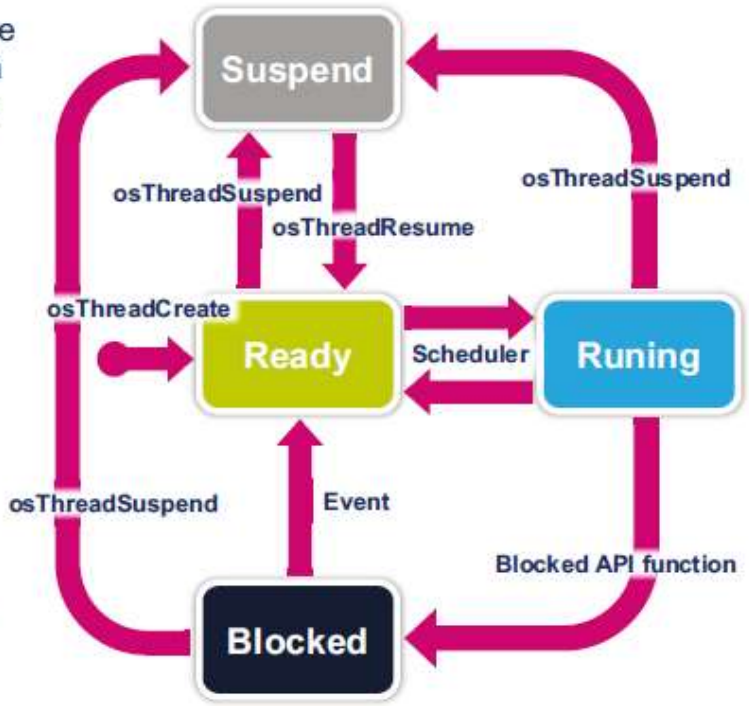

```
#define SRAM1_OS_START (uint8_t *)0x2000 1000
#define SRAM1_OS_SIZE  0x0800 //2kB
#define SRAM2_OS_START (uint8_t *)0x1000 0000
#define SRAM2_OS_SIZE  0x1000 //4kB

Const HeapRegion_t xHeapRegions[] =
{
 {SRAM2_OS_START, SRAM2_OS_SIZE},
 {SRAM1_OS_START, SRAM1_OS_SIZE},
 {NULL,0} /*terminates the array*/
}

/*before call of any OS create function*/
vPortDefineHeapRegions(HeapRegions);
```

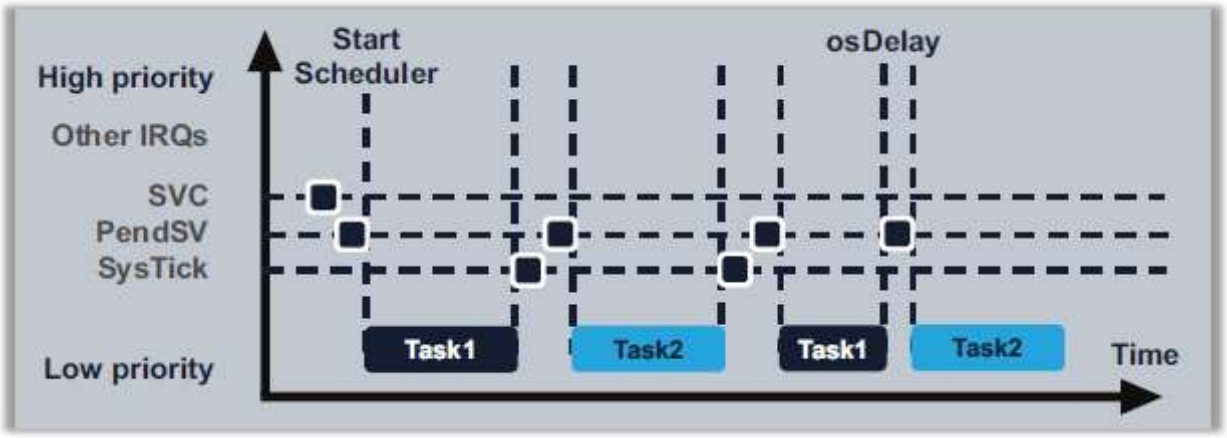
FreeRTOS 에서 Task 의 상태는 4가지로 구분할 수 있다.
Running 상태의 Task 는 한 시점에서 오로지 한개의 Task 만 가능하다.
Running 상태로 전환할 준비가 된 Task 들은 Ready 상태이다.
Delay 나 세마포어, 뮤텍스 등의 동기화 기능을 사용 중 대기가 필요한 경우 Block 상태로 전환되며 이 때에는 Context Switching 이 발생한다.
Suspended 상태는 스케쥴링 대상에서 제외된 Task 들의 상태이다.

- **Ready**
  - Tasks are ready to execute but are not currently executing because a different task with equal or higher priority is running
- **Running**
  - When task is actually running
- **Blocked**
  - Task is waiting for a either a temporal or external event
- **Suspended**
  - Task not available for scheduling



Running 상태에 있는 Task 는 오로지 한개이기 때문에 다수의 Task 가 수행하기 위해서는 Context Switching 이 필요하다. Context Switching 이 발생하면 현재 Running 상태의 Task 는 현재 상태를 Stack 에 저장하고 Ready 상태에 있던 가장 우선순위가 높은 Task 가 Running 상태로 전환된다.

Context Switching 관련되어 있는 인터럽트는 SVC, PendSV, SysTick 인터럽트이다.
아래 그림에서 보듯이 SVC, PendSV, SysTick 인터럽트의 Priority 가 낮게 되어 있다. 따라서 다른 IRQ 의 서비스 루틴이 수행 중에는 Context Switching 이 발생하지 않는다.
SysTick 인터럽트도 코드를 살펴보면 아래와 같이 PendSV 인터럽트를 발생하도록 하여 Context Switching 이 수행되도록 하고 있다.




```
void xPortSysTickHandler( void )
{
    /* The SysTick runs at the lowest interrupt priority, so when this i
    executes all interrupts must be unmasked.  There is therefore no nee
    save and then restore the interrupt mask value as its value is alrea
    known. */
    portDISABLE_INTERRUPTS();
    {
        /* Increment the RTOS tick. */
        if( xTaskIncrementTick() != pdFALSE )
        {
            /* A context switch is required.  Context switching is perfc
            the PendSV interrupt.  Pend the PendSV interrupt. */
            portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT;
        }
    }
    portENABLE_INTERRUPTS();
}
```

FreeRTOS 에서 사용되는 Stack Pointer 는 MSP(Main Stack Pointer) 와 PSP(Process Stack Pointer) 가 있다. MSP 는 인터럽트용으로 사용되는 Main Stack 의 포인터이며, PSP 는 FreeRTOS Heap 영역에 할당된 Task 별 Stack 포인터이다.

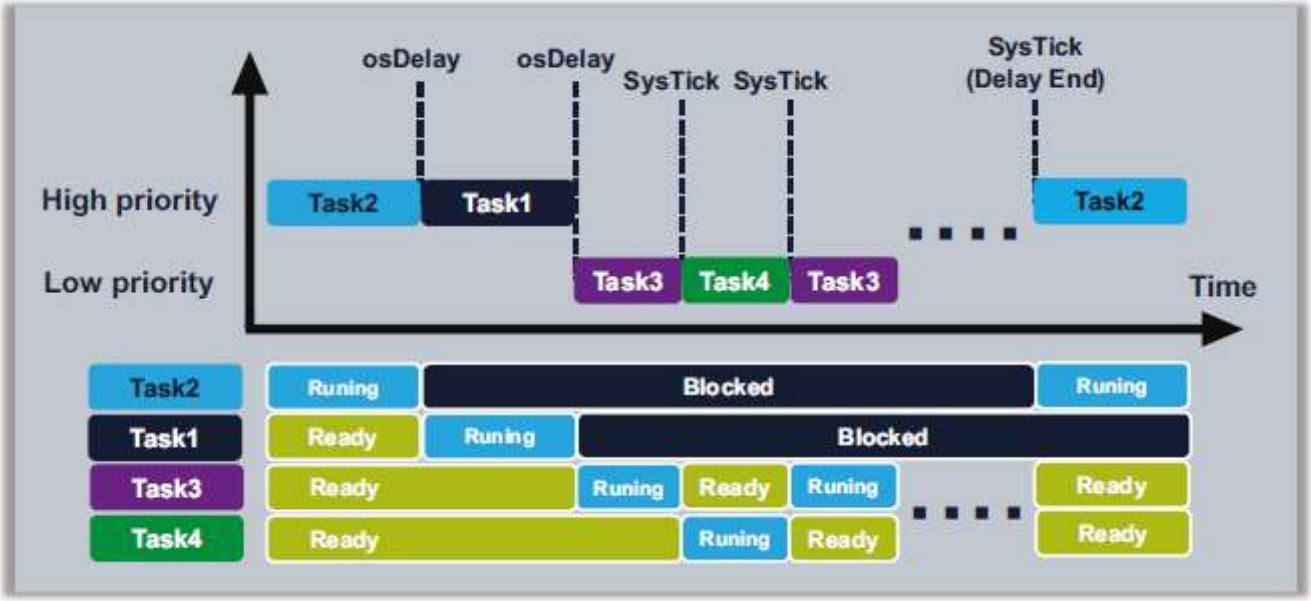
[Lab#1]

프로젝트 구성

- Task1, Task2, Task3, Task4 개의 Task 를 생성한다.
- Task1, Task2 는 osPriorityAboveNormal 로 우선순위를 설정한다.
- Task3, Task4 는 osPriorityNormal 로 우선순위를 설정한다.
- Task1, Task2 는 osDelay API 를 사용해서 각 1초 Delay 를 부여한다.
- Task3, Task4 는 HAL_Delay API 를 사용하여 1초 Delay 를 부여한다.

동작

- osDelay 는 Context Switching 을 발생시키지만, HAL_Delay 는 Context Switching 을 발생시키지 않는다. 따라서 Task3과 Task4 는 SysTick 에 의해서 Context Switching 을 수행하며, Blocked 상태로 전환되지 않는다.




```
/* USER CODE BEGIN Header_StartTask01 */
/**
 * @brief  Function implementing the task1 thread.
 * @param  argument: Not used
 * @retval None
 */
/* USER CODE END Header_StartTask01 */
void StartTask01(void const * argument)
{
  /* USER CODE BEGIN StartTask01 */
```

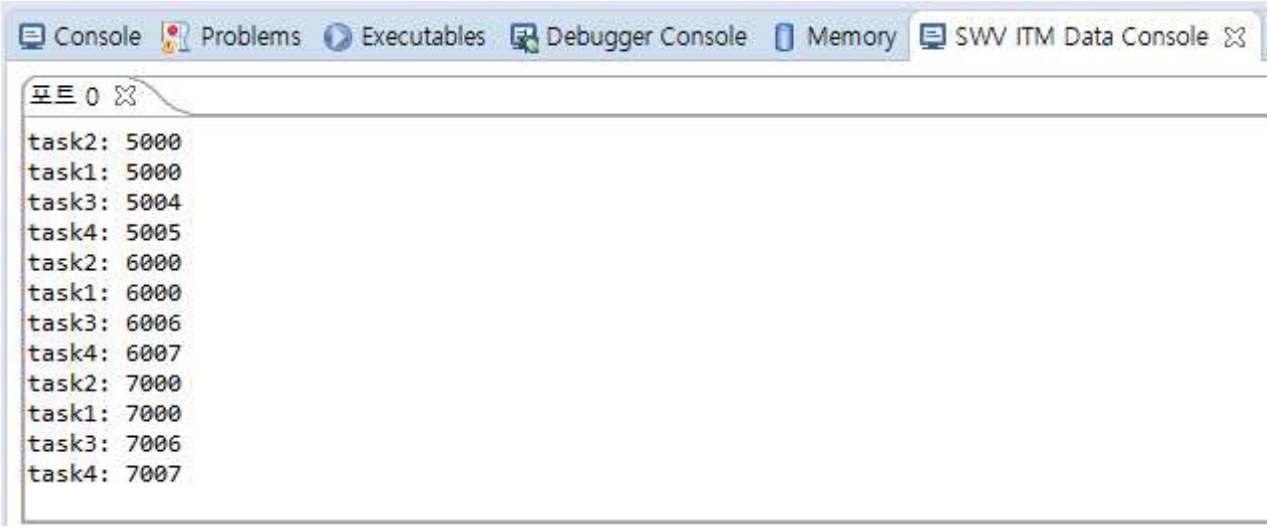
```
    /* Infinite loop */
    for (;;) {
        printf("task1: %lu\r\n", osKernelSysTick());
        osDelay(1000);
    }
  /* USER CODE END StartTask01 */
}

/* USER CODE BEGIN Header_StartTask02 */
/**
 * @brief Function implementing the task2 thread.
 * @param argument: Not used
 * @retval None
 */
/* USER CODE END Header_StartTask02 */
void StartTask02(void const * argument)
{
  /* USER CODE BEGIN StartTask02 */
    /* Infinite loop */
    for (;;) {
        printf("task2: %lu\r\n", osKernelSysTick());
        osDelay(1000);
    }
  /* USER CODE END StartTask02 */
}

/* USER CODE BEGIN Header_StartTask03 */
/**
 * @brief Function implementing the task3 thread.
 * @param argument: Not used
 * @retval None
 */
/* USER CODE END Header_StartTask03 */
void StartTask03(void const * argument)
{
  /* USER CODE BEGIN StartTask03 */
    /* Infinite loop */
    for (;;) {
        printf("task3: %lu\r\n", osKernelSysTick());
        HAL_Delay(1000);
    }
  /* USER CODE END StartTask03 */
}
```

```
/* USER CODE BEGIN Header_StartTask04 */
/**
 * @brief Function implementing the task4 thread.
 * @param argument: Not used
 * @retval None
 */
/* USER CODE END Header_StartTask04 */
void StartTask04(void const * argument)
{
  /* USER CODE BEGIN StartTask04 */
    /* Infinite loop */
    for (;;) {
        printf("task4: %lu\r\n", osKernelSysTick());
        HAL_Delay(1000);
    }
  /* USER CODE END StartTask04 */
}
```

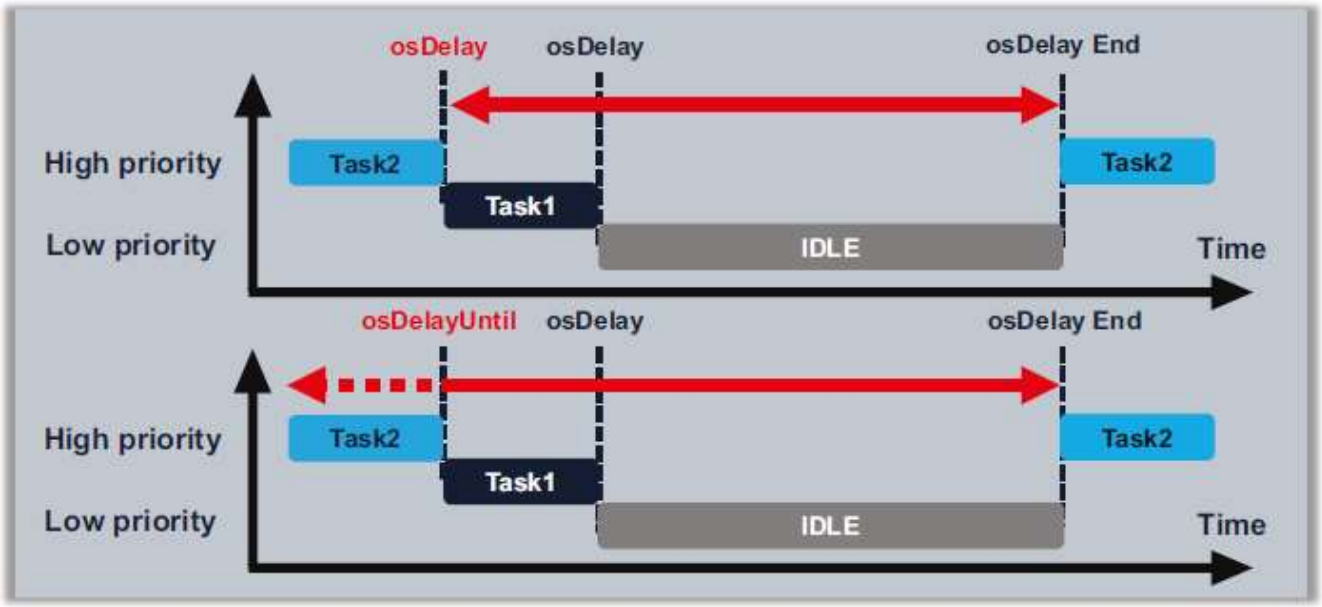
SWV 데이터를 확인하면 osDelay 를 사용하던 HAL_Delay 를 사용하던 1초 주기는 유지하는 것으로 보인다. 다만 앞서 언급한 것과 같이 HAL_Delay 를 사용하면 task 가 blocked 상태로 전화되지 않고 지속적으로 running, ready 상태를 반복한다.
task1, task2 의 우선순위를 높게 하였기 때문에 task1, task2 의 주기가 정확한 반면 task3, task4 의 주기는 약간씩 밀리는 모습이 관찰된다.



osDelay 는 task 를 blocked 상태로 전환시키고 context switching 을 발생시킨다. 지정된 delay 시간 후 ready 상태에서 스케쥴러에 의해서 다시 running 상태가 될 수 있다.

osDelayUntil 은 지정된 시점으로부터 일정 기간 task 를 delay 시킨다.

아래 그림을 보면 osDelay 의 경우, task2 가 osDelay 를 호출한 시점부터 지정된 period 만틈 delay 수행 후에 task2 는 다시 running 상태로 전환되며 osDelayUntil 의 경우에는 osDelayUntil 함수 호출에 앞서 지정된 시점부터 지정된 period 만큼 delay 를 수행후에 다시 task2 가 running 상태로 전환된다.



osDelay 와 osDelayUntil 을 사용하기 위해서는 vTaskDelay 와 vTaskDelayUntil 항목이 Enable 되어 있어야 한다. CubeMx 에서 두항목은 default 로 enable 상태이다.

[Lab#2]

프로젝트 구성

- Task1, Task2 2개의 Task 를 생성한다.
- Task1, Task2 는 osPriorityNormal 로 우선순위를 설정한다.
- Task1 은 루프 진입전 시점에서 osDelay를 사용해서 2초 Delay 를 부여한다.
- Task2 는 루프 내에서 osDelayUntil 을 사용해서 2초 Delay 를 부여한다.

동작

- Task1 은 osDelay 를 사용하기 때문에 중간에 어떤 작업의 시간이 가변이라면 Task 의 주기성을 보장하기 힘들다. 예를 들어서 아래 코드처럼 중간에 HAL_Delay 에 의한 1초 딜레이가 들어간다면 실제 동작 주기는 3초가 될 것이다. 그런데 HAL_Delay 대신 어떤 작업이 수행시간이 가변이라면 동작 주기는 가변이 된다.

- 반면, Task2 는 루프 내에서 osDelayUntil 을 사용하기 때문에 중간에 HAL_Delay 값이 2초 이내에서 가변한다면 동작의 주기성을 보장할 수 있다.

```
/* USER CODE END Header_StartTask01 */
void StartTask01(void const * argument)
{
  /* USER CODE BEGIN StartTask01 */

    /* Infinite loop */
    for (;;) {
        printf("task1 %lu\n", osKernelSysTick());
        HAL_Delay(1000); //중간의 딜레이가 가변이라면 task1 의 동작 주기는 일
        osDelay(2000);
    }
  /* USER CODE END StartTask01 */
}

/* USER CODE BEGIN Header_StartTask02 */
/**
 * @brief Function implementing the myTask02 thread.
 * @param argument: Not used
 * @retval None
 */
/* USER CODE END Header_StartTask02 */
void StartTask02(void const * argument)
{
  /* USER CODE BEGIN StartTask02 */
    uint32_t lastTime;

    /* Infinite loop */
    for (;;) {
        lastTime = osKernelSysTick();
        printf("task2 %lu\n", lastTime);
        HAL_Delay(1000); //중간의 딜레이가 가변이라도 task2 의 동작 주기는 일
        osDelayUntil(&lastTime, 2000);
    }
  /* USER CODE END StartTask02 */
}
```

디버깅을 하면 앞서 언급한 것과 같이 Task1은 3초 주기로 동작하고, Task2는 2초 주기로 동작한다.

#freertos #stm32

3 4