

FreeRTOS

[STM32 FreeRTOS] 자료 Review#2



이지훈

2020. 3. 1. 22:53

이웃추가

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

https://github.com/eziya/STM32F4_HAL_FREERTOS_LAB

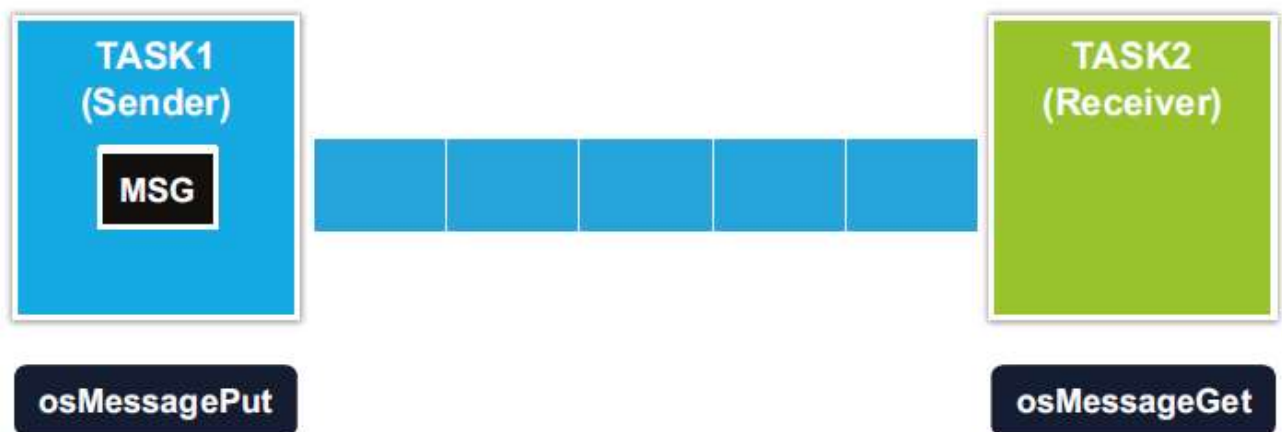
**eziya/STM32F4_HAL_FREERTOS_LAB**

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.

Queue 는 task 간 동기화와 함께 데이터를 공유하는 용도로 사용가능



osMessageCreate() 함수로 생성

osMessagePut() 함수로 큐에 데이터 전달

osMessageGet() 함수로 큐에서 데이터 수신

- Create Queue:

```
osMessageQId osMessageCreate(const osMessageQDef_t *queue_def,
osThreadId id)
```

- Put data into Queue

```
osStatus osMessagePut(osMessageQId id, uint32_t info, uint32_t
millisec)
```

Item to send

- Receive data from Queue

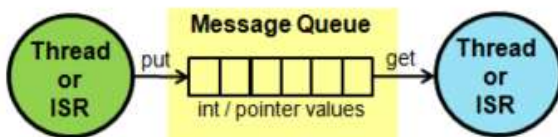
```
osEvent osMessageGet(osMessageQId id, uint32_t millisec)
```

Structure with status and with received item

osEvent 구조체는 union 을 이용하여 uint32_t 값 형태의 데이터 v를 사용하거나 pointer 변수 p를 사용할 수 있음. osMailCreate 를 사용하는 경우에는 mail_id 를 사용하고 osMessageCreate 를 사용한 경우에는 message_id 를 사용함.

Message 와 Mail 은 모두 Queue 인데 둘의 차이점은 아래 그림처럼 API 를 살펴보면 Message 는 큐의 기본 기능에 충실하게 Get / Put / Peek 등의 API 를 제공하는 반면, Mail은 메모리 블록을 alloc / free 하는 기능을 추가적으로 제공한다는 점이다.

Message passing is another basic communication model between threads. In the message passing model, one thread sends data explicitly, while another thread receives it. The operation is more like some kind of I/O rather than a direct access to information to be shared. In CMSIS-RTOS, this mechanism is called a **message queue**. The data is passed from one thread to another in a FIFO-like operation. Using message queue functions, you can control, send, receive, or wait for messages. The data to be passed can be of integer or pointer type:



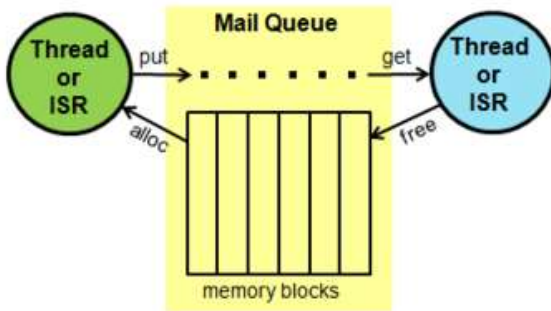
CMSIS-RTOS Message Queue

Compared to a **Memory Pool**, message queues are less efficient in general, but solve a broader range of problems. Sometimes, threads do not have a common address space or the use of shared memory raises problems, such as mutual exclusion.

아래 글에는 Mail 이 Message 보다 좋은 점이 서로 다른 task 간에 포인터만을 이동 시켜서 사용이 가능한 점을 장점이라고 기술하고 있는데 결국 Message 로도 못하는 것이 아니기 때문에 별다른 advantage 라고 생각이 들지는 않는다. (정확한 이점을 알고 계시는 분은 feedback 좀 부탁드립니다!)

Description

A **mail queue** resembles a **Message Queue**, but the data that is being transferred consists of memory blocks that need to be allocated (before putting data in) and freed (after taking data out). The mail queue uses a **Memory Pool** to create formatted memory blocks and passes pointers to these blocks in a message queue. This allows the data to stay in an allocated memory block while only a pointer is moved between the separate threads. This is an advantage over **messages** that can transfer only a 32-bit value or a pointer. Using the mail queue functions, you can control, send, receive, or wait for mail.



[LAB#3]

프로젝트 구성

- Sender, Receiver 2개의 Task 를 생성한다.
- Sender, Receiver 는 osPriorityNormal 로 우선순위를 설정한다.
- 128 size 의 Queue 를 한개 생성한다.
- Sender task는 1초다마 osMessagePut 기능을 이용해서 데이터를 전송한다.
- Receiver task 는 루프에서 지속적으로 osMessageGet 기능을 이용해서 데이터를 수신한다.

동작

- Receiver task 가 osMessageGet 을 호출할 때 큐에 데이터가 없는 경우에는 blocked 상태로 전환되고 context switching 이 발생한다.
- Sender 와 Receiver 가 모두 blocked 상태일 때는 Idle task 가 수행된다.
- osMessageGet 함수가 지정된 timeout 값을 초과하면 ready 상태가 된다.
- Receiver task 가 osMessageGet 을 호출할 때 큐에 데이터가 있는 경우에는 그대로 task 가 수행된다.
- 메시지 수신 시 osEvent.status 값은 osEventMessage 가 된다.
- 포인터 데이터는 osEvent.value.p 로 접근할 수 있다.

```
/* USER CODE END Header_SenderTask */
void SenderTask(void const * argument)
{
    /* USER CODE BEGIN SenderTask */
    _Message msg;
    msg.buf[0] = 0xAA;
    msg.buf[1] = 0xBB;
    msg.idx = 0;

    /* Infinite loop */
    for (;;)
    {
```

```

    {
        printf("Sender enqueues\n");
        osMessagePut(Queue1Handle, &msg, 100); //enqueue
        msg.idx++;
        printf("Sender delays for a sec.\n");
        osDelay(1000);
    }
/* USER CODE END SenderTask */
}

/* USER CODE BEGIN Header_ReceiverTask */
/**
 * @brief Function implementing the Receiver thread.
 * @param argument: Not used
 * @retval None
 */
/* USER CODE END Header_ReceiverTask */
void ReceiverTask(void const * argument)
{
    /* USER CODE BEGIN ReceiverTask */
    _Message *pMsg;
    osEvent retVal;

    /* Infinite loop */
    for (;;)
    {
        printf("Receiver is trying to dequeue\n");
        retVal = osMessageGet(Queue1Handle, 500); //dequeue
        if(retVal.status == osEventMessage)
        {
            pMsg = retVal.value.p;
            printf("Receiver received: msg.buf[0]=0x%X, msg.idx=%u\n", p
        }
    }
/* USER CODE END ReceiverTask */
}

```

```

Console Problems Executables Debugger Console Memory SWV ITM Data Console
포트 0
Sender delays for a sec.
Receiver received: msg.buf[0]=0xAA, msg.idx=9
Receiver is trying to dequeue
Receiver is trying to dequeue
Sender enqueues
Sender delays for a sec.
Receiver received: msg.buf[0]=0xAA, msg.idx=10
Receiver is trying to dequeue
Receiver is trying to dequeue
Sender enqueues
Sender delays for a sec.
Receiver received: msg.buf[0]=0xAA, msg.idx=11
Receiver is trying to dequeue
Receiver is trying to dequeue
Sender enqueues
Sender delays for a sec.
Receiver received: msg.buf[0]=0xAA, msg.idx=12
Receiver is trying to dequeue
Receiver is trying to dequeue

```

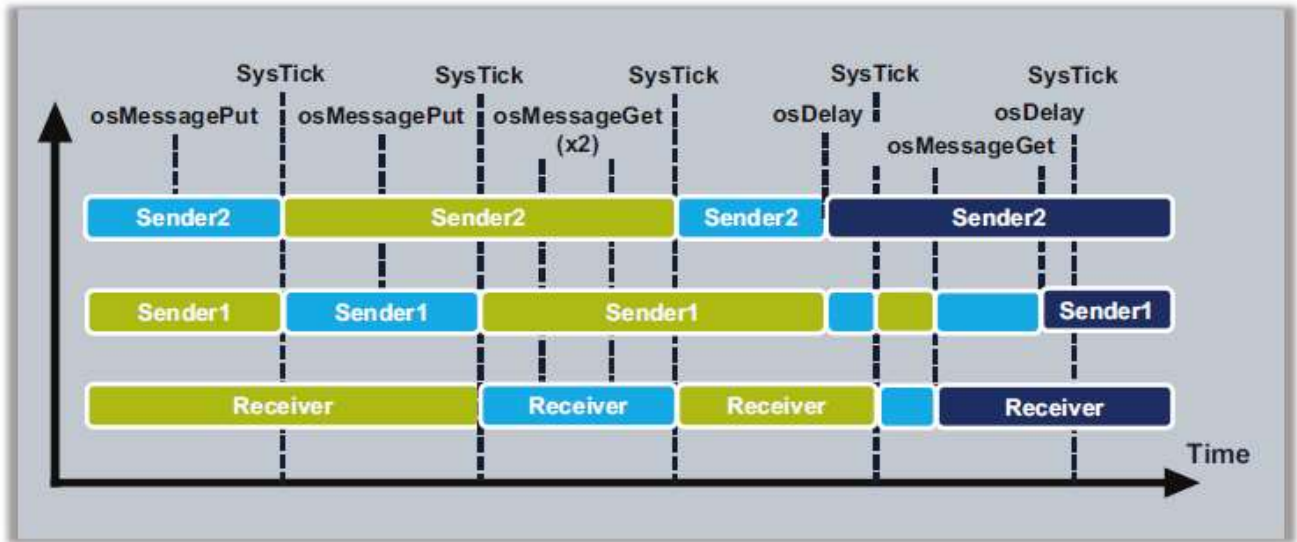
[LAB#4]

프로젝트 구성

- 2개의 Sender Task 와 1개의 Receiver Task 를 생성한다.
- Sender1, Sender2, Receiver 는 osPriorityNormal 로 우선순위를 설정한다.
- 128 size 의 Queue 를 한개 생성한다.
- Sender task 들은 2초다마 osMessagePut 기능을 이용해서 데이터를 전송한다.
- Receiver task 는 루프에서 지속적으로 osMessageGet 기능을 이용해서 데이터를 수신한다.

동작

- Sender1, Sender2 가 큐에 메시지를 put 한 이후에 Receiver 가 동작한다.
- Receiver 의 우선 순위가 높지 않기 때문에 즉시 큐에서 메시지를 Get 하지 못한다.
- 만일 Sender 가 많다면 Receiver 가 큐에서 메시지를 빨리 Get 하지 못하기 때문에 Full 이 발생할 소지가 있다.



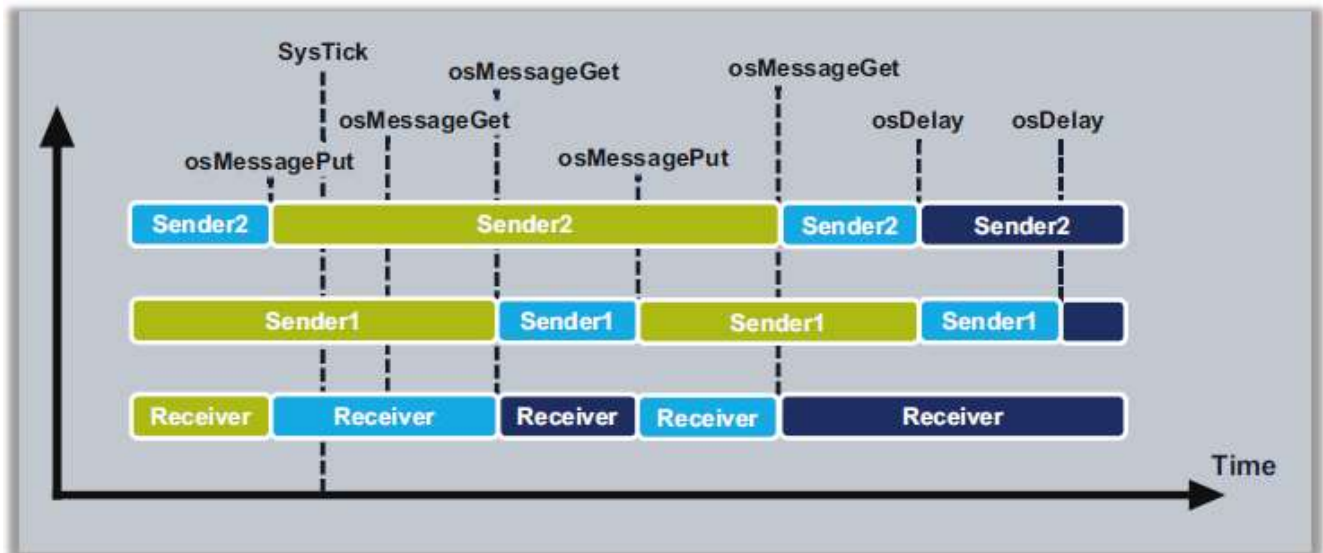
실제 실행을 해 보면 위 그림처럼 Sender1, Sender 2가 put 을 수행하고 난 뒤에 Receiver 가 연속으로 2개의 메시지를 get 하는 것을 확인할 수 있다. 그리고 더 이상 메시지가 없기 때문에 Receiver timeout 이 발생한다.

```

Console Problems Executables Debugger Console Memory SWV ITM Data Console
포트 0
Don't remove this printf for debugging.
Sender1
Sender1 delay
Sender2
Sender2 delay
Receiver 1
Receiver 2
Receiver timeout
Sender1
Sender1 delay
Sender2
Sender2 delay
Receiver 1
Receiver 2
Receiver timeout
Sender1
Sender1 delay
Sender2
Sender2 delay

```

- 만일 Receiver 의 우선순위가 Sender 들 보다 높다면 동작 Sender 들이 메시지를 Put 하고 osDelay 를 수행할 때마다 Receiver 가 우선순위가 높기 때문에 즉시 메시지를 Get 할 수 있다.



```

/* USER CODE BEGIN Header_SenderTask1 */
/**
 * @brief Function implementing the Sender1 thread.
 * @param argument: Not used
 * @retval None
 */
/* USER CODE END Header_SenderTask1 */
void SenderTask1(void const * argument)
{
    /* USER CODE BEGIN SenderTask1 */
    /* Infinite loop */
    for (;;)
    {
        printf("Sender1\n");
        osMessagePut(myQueue01Handle, 0x01, 100);
        printf("Sender1 delay\n");
        osDelay(2000);
    }
    /* USER CODE END SenderTask1 */
}

/* USER CODE BEGIN Header_SenderTask2 */
/**
 * @brief Function implementing the Sender2 thread.
 * @param argument: Not used
 * @retval None
 */
/* USER CODE END Header_SenderTask2 */
void SenderTask2(void const * argument)
{
    /* USER CODE BEGIN SenderTask2 */

```

```

/* USER CODE BEGIN SenderTask2 */
/* Infinite loop */
for (;;)
{
    printf("Sender2\n");
    osMessagePut(myQueue01Handle, 0x02, 100);
    printf("Sender2 delay\n");
    osDelay(2000);
}
/* USER CODE END SenderTask2 */
}

/* USER CODE BEGIN Header_ReceiverTask1 */
/**
 * @brief Function implementing the Receiver1 thread.
 * @param argument: Not used
 * @retval None
 */
/* USER CODE END Header_ReceiverTask1 */
void ReceiverTask1(void const * argument)
{
    /* USER CODE BEGIN ReceiverTask1 */
    osEvent retVal;
    /* Infinite loop */
    for (;;)
    {
        retVal = osMessageGet(myQueue01Handle, 1000);
        if(retVal.status == osEventMessage)
        {
            printf("Receiver %lu\n", retVal.value.v);
        }
        else if(retVal.status == osEventTimeout)
        {
            printf("Receiver timeout\n");
        }
    }
    /* USER CODE END ReceiverTask1 */
}

```

Receiver 의 priority 가 osPriorityAboveNormal 인 경우를 살펴보면 Sender1 이 put 한 이후 바로 Receiver 가 수행되면서 메시지를 get 하고 다시 Sender2 가 put 한 이후 바로 Receiver 가 get 하면서 메시지를 즉시 get 하는 것을 확인할 수 있다.


```

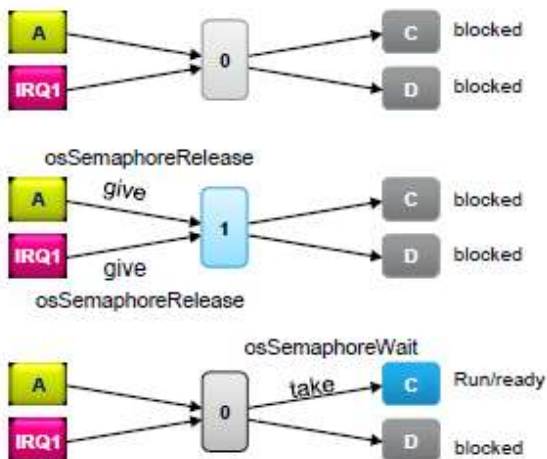
Console Problems Executables Debugger Console Memory SWV ITM Data Console
포트 0
Don't remove this printf for debugging.
Sender1
Receiver 1
Sender2
Receiver 2
Sender1 delay
Sender2 delay
Receiver timeout
Receiver timeout
Sender1
Receiver 1
Sender2
Receiver 2
Sender1 delay
Sender2 delay
Receiver timeout
Receiver timeout
Sender1
Receiver 1

```

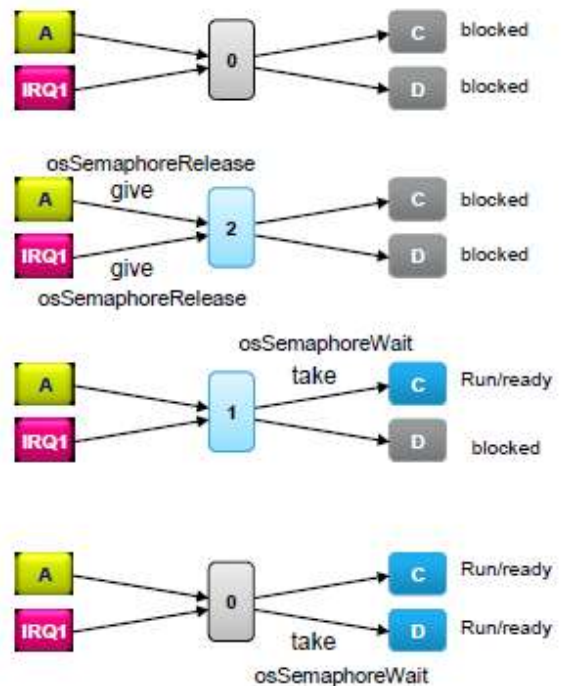
세마포어는 Task 사이 또는 Task 와 Interrupt 사이에 동기화에 사용된다.

Counting 세마포어의 경우 카운터 값인 Token 의 수만큼 Get 할 수 있고 모든 Token 을 Client 들이 Get 한 경우에는 대기해야 한다. (쉽게 말하면 식당에 좌석의 수만큼 들어가서 밥을 먹을 수 있고, 다 먹은 사람이 나가야 다음 손님이 들어갈 수 있다)

Binary 세마포어는 카운터값이 1인 Counting 세마포어이다. 즉 좌석이 하나인 식당이라 한 시점에 한명의 손님만 밥을 먹을 수 있다.



Binary



Counting

[LAB#5]

프로젝트 구성

- 2개의 Task 를 생성한다.
- Task1, Task2 는 osPriorityNormal 로 우선순위를 설정한다.
- Binary 세마포어를 한개 생성한다.
- Task1 은 2초 간격으로 세마포어를 release 한다.
- Task2 는 지속적으로 세마포어를 wait 하고, 세마포어를 획득하면 printf 로 메시지를 출력한다.

동작

- Task1 이 세마포어를 release 하면 Task2 는 blocked 상태에서 ready 상태를 거쳐 running 상태가 된다.
- Task2 가 세마포어를 1초 timeout 으로 wait 하고 Task1 이 2초마다 release 하기 때문에 Task2 는 주기적으로 timeout 이 발생한다.

```

/* USER CODE END Header_StartTask1 */
void StartTask1(void const *argument)
{
    /* USER CODE BEGIN StartTask1 */
    /* Infinite loop */
    for (;;)
    {
        osDelay(2000);
        printf("Task1 release semaphore.\n");
        osSemaphoreRelease(myBinarySem01Handle);
    }
    /* USER CODE END StartTask1 */
}

/* USER CODE BEGIN Header_StartTask2 */
/**
 * @brief Function implementing the Task2 thread.
 * @param argument: Not used
 * @retval None
 */
/* USER CODE END Header_StartTask2 */
void StartTask2(void const *argument)
{
    /* USER CODE BEGIN StartTask2 */
    int32_t retVal;
    /* Infinite loop */
    for (;;)
    {
        retVal = osSemaphoreWait(myBinarySem01Handle, 1000);
        if(retVal == 0)
        {
            printf("Task2 synchronized...\n");
        }
        else
        {
            printf("Task2 timeout or error.\n");
        }
    }
    /* USER CODE END StartTask2 */
}

```

Task2 난 처음에는 binary 세마포어의 token 이 1로 존재하기 때문에 동기화 된다.

다음에는 아직 Task1 이 세마포어를 release 하지 않았기 때문에 timeout 이 발생한다.
Task1 이 세마포어를 release 하면 Task2 는 다시 동기화 된다.

```

Console Problems Executables Debugger Console Memory SWV ITM Data Console
포트 0
Don't remove this printf for debugging.
Task2 synchronized...
Task2 timeout or error.
Task1 release semaphore.
Task2 synchronized...
Task2 timeout or error.
Task1 release semaphore.
Task2 synchronized...
Task2 timeout or error.
Task1 release semaphore.
Task2 synchronized...
Task2 timeout or error.
Task1 release semaphore.
Task2 synchronized...
Task2 timeout or error.
Task1 release semaphore.
Task2 synchronized...
Task2 timeout or error.

```

[LAB#6]

프로젝트 구성

- 3개의 Task 를 생성한다.
- Task1, Task2, Task3 는 osPriorityNormal 로 우선순위를 설정한다.
- USE_COUNTING_SEMAPHORES 를 Enable 한다.
- Counting 세마포어를 한개 생성하고 Count 값은 2로 한다. (2개의 Token)
- Task1 과 Task2 는 2초 간격으로 세마포어를 release 한다.
- Task3 은 Task1 과 Task2 에서 release 한 세마포어 Token 2개를 wait 하고, 세마포어를 획득 하면 printf 로 메시지를 출력한다.

동작

- Task3 는 세마포어를 wait 하면서 blocked 상태로 전환된다.
- Task1 이 세마포어를 release 하면 Task3 는 ready 상태를 거쳐 running 상태가 되고 다시 wait 하면서 blocked 상태가 된다.
- Task2 이 세마포어를 release 하면 Task3 는 ready 상태를 거쳐 running 상태가 되고 printf 를 통해서 동기화 완료 메시지를 출력한다.

```

/* USER CODE BEGIN Header_StartTask1 */
/**
 * @brief Function implementing the Task1 thread.
 * @param argument: Not used
 * @retval None
 */

```

```
/* USER CODE END Header_StartTask1 */
void StartTask1(void const *argument)
{
    /* USER CODE BEGIN StartTask1 */
    /* Infinite loop */
    for (;;)
    {
        osDelay(2000);
        printf("Task1 release counting semaphore.\n");
        osSemaphoreRelease(myCountingSem01Handle);
    }
    /* USER CODE END StartTask1 */
}

/* USER CODE BEGIN Header_StartTask2 */
/**
 * @brief Function implementing the Task2 thread.
 * @param argument: Not used
 * @retval None
 */
/* USER CODE END Header_StartTask2 */
void StartTask2(void const *argument)
{
    /* USER CODE BEGIN StartTask2 */
    /* Infinite loop */
    for (;;)
    {
        osDelay(2000);
        printf("Task2 release counting semaphore.\n");
        osSemaphoreRelease(myCountingSem01Handle);
    }
    /* USER CODE END StartTask2 */
}

/* USER CODE BEGIN Header_StartTask3 */
/**
 * @brief Function implementing the Task3 thread.
 * @param argument: Not used
 * @retval None
 */
/* USER CODE END Header_StartTask3 */
void StartTask3(void const *argument)
{
    /* USER CODE BEGIN StartTask3 */
```

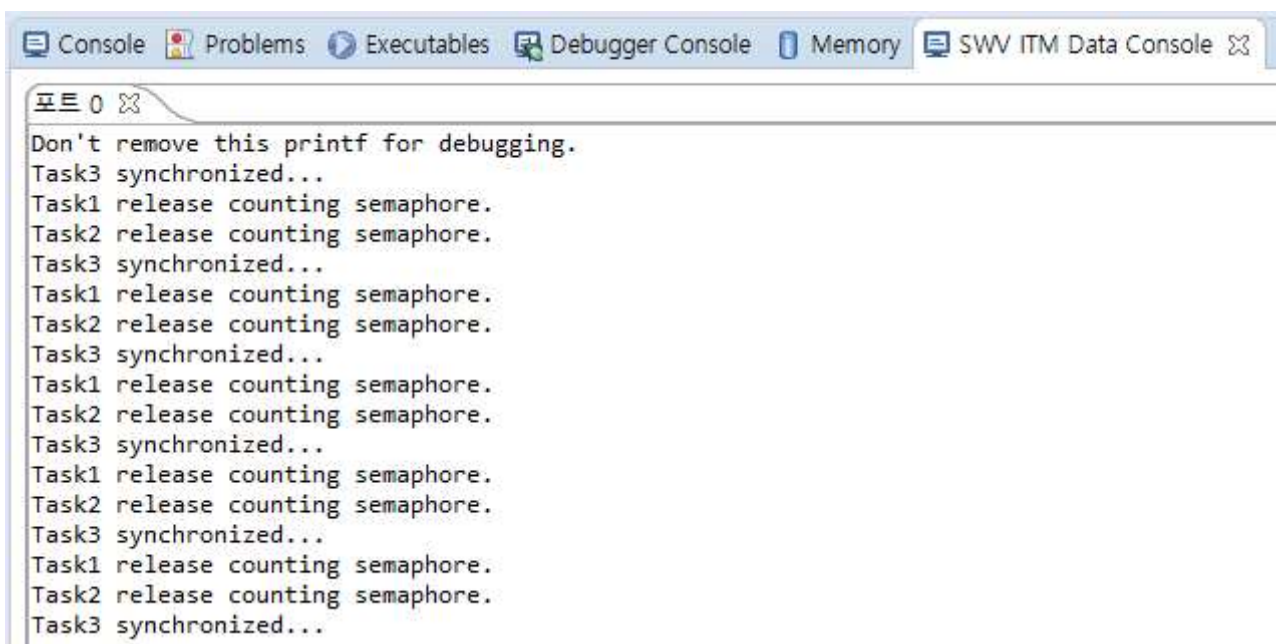
```

/* USER CODE BEGIN StartTask3 */
int32_t retVal;
/* Infinite loop */
for (;;)
{
    osSemaphoreWait(myCountingSem01Handle, 4000);
    retVal = osSemaphoreWait(myCountingSem01Handle, 4000);

    if(retVal == 0)
    {
        printf("Task3 synchronized...\n");
    }
    else
    {
        printf("Task3 timeout or error.\n");
    }
}
/* USER CODE END StartTask3 */
}

```

- 최초 구동 시 세마포어 Token 2개를 모두 get 하고 Task3 가 수행된다.
- Task1 과 Task2 가 세마포어를 release 하면 Task3 는 wait 상태에서 벗어나서 동기화를 수행한다.
- 복수개의 task 수행을 완료 후 동작해야 하는 케이스를 모사한 예제인데 개인적으로는 이런 케이스에는 카운팅 세마포어보다는 가볍고 task 별로 이벤트를 명확하게 구분할 수 있는 Signal 을 쓰는게 더 좋지 않을까 싶다. 카운팅 세마포어는 복수개의 task 가 리소스에 접근할 수 있는 경우 사용하는 것이 더 직관적일 것 같다.



```

Console Problems Executables Debugger Console Memory SWV ITM Data Console
포트 0
Don't remove this printf for debugging.
Task3 synchronized...
Task1 release counting semaphore.
Task2 release counting semaphore.
Task3 synchronized...
Task1 release counting semaphore.
Task2 release counting semaphore.
Task3 synchronized...
Task1 release counting semaphore.
Task2 release counting semaphore.
Task3 synchronized...
Task1 release counting semaphore.
Task2 release counting semaphore.
Task3 synchronized...
Task1 release counting semaphore.
Task2 release counting semaphore.
Task3 synchronized...

```


#stm32 #freertos

5

1



이지훈

달릴 준비만 하는거 아냐...달려야 하는데...^^; <https://github.com/eziya>