Programmazione di sistemi multicore FreeRTOS #2

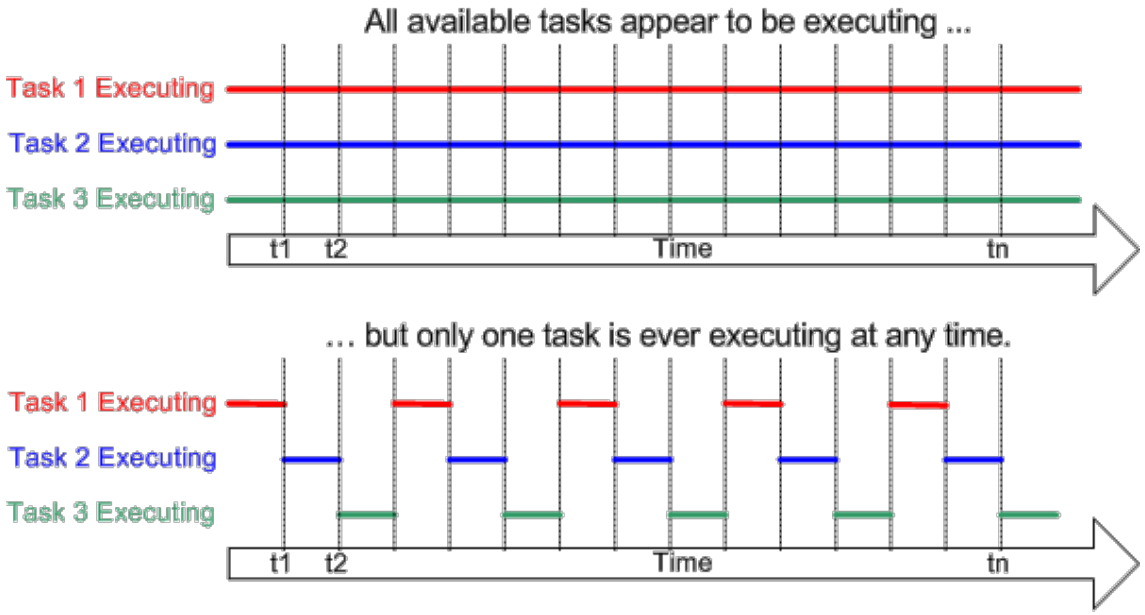
Fabrizio Gattuso
fabrizio.gattuso@wsense.it

SAPIENZA
UNIVERSITÀ DI ROMA

W • SENSE
INTEGRATED CABLELESS SOLUTIONS

RTOS

Un Real Time Operating System (RTOS) è un sistema operativo progettato per fornire un modello di esecuzione predicibile **(deterministico).**



Gestione dei task

I task sono funzioni C, che ritornano void e prendere come parametro un puntatore a void:

*void ATaskFunction(void *pvParameters);*

Normalmente sono pensati come task infiniti e sono implementati grazie a *while(1)* o *for(;;)*

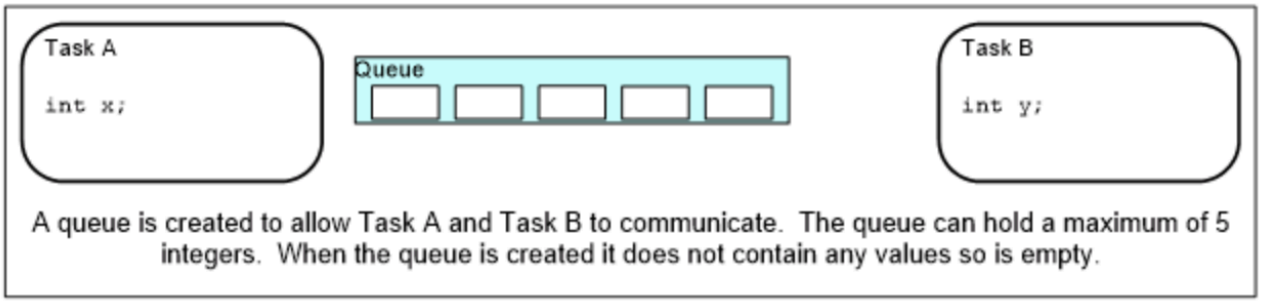
Queue

Le code sono utilizzate come mezzo di comunicazione:

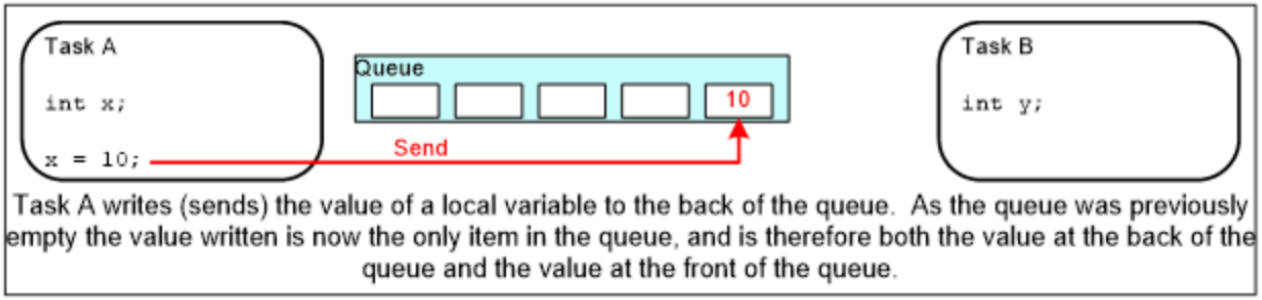
1. Tra task
2. Da task a interrupt (e viceversa)

Una coda può contenere un **numero finito e di dimensione fissa di elementi**. Il numero massimo di elementi è chiamato **length**. La lunghezza della coda e la dimensione del singolo elemento sono fissati in fase di creazione.

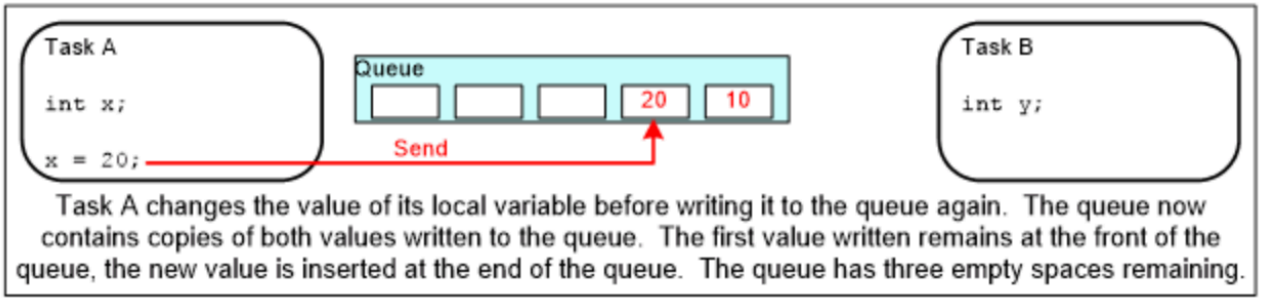
Queue (2)



A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.

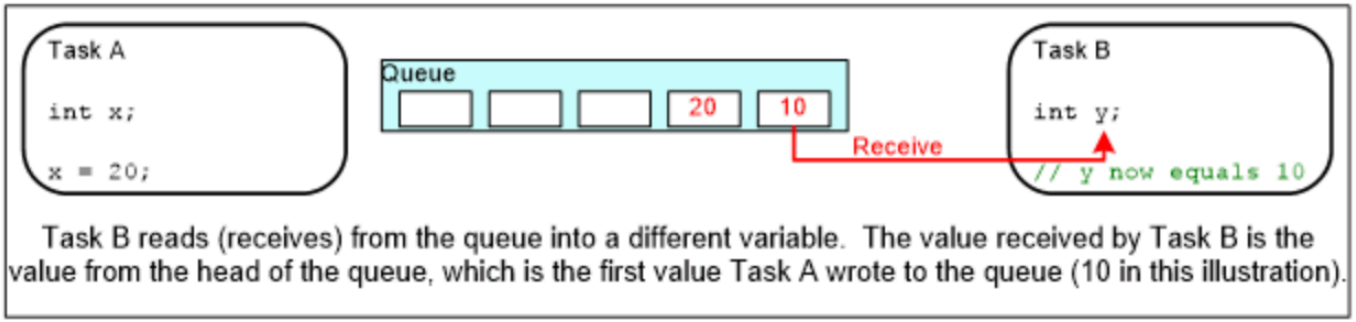


Task A writes (sends) the value of a local variable to the back of the queue. As the queue was previously empty the value written is now the only item in the queue, and is therefore both the value at the back of the queue and the value at the front of the queue.

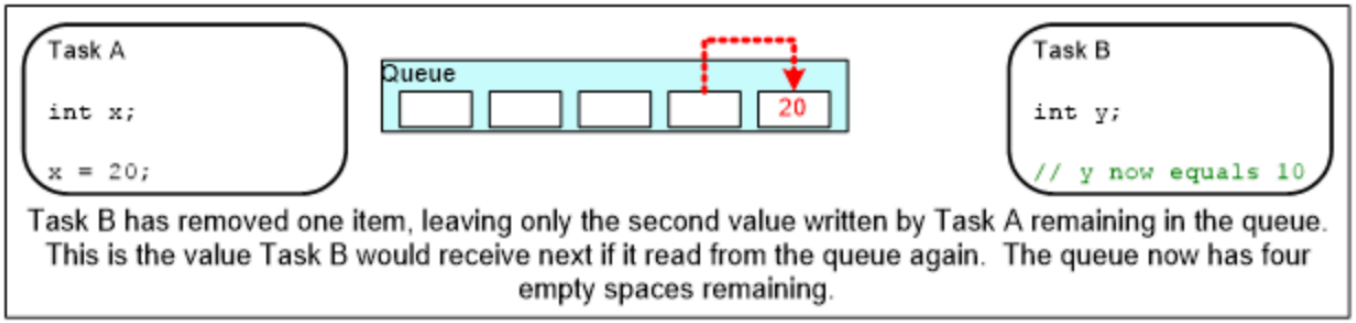


Task A changes the value of its local variable before writing it to the queue again. The queue now contains copies of both values written to the queue. The first value written remains at the front of the queue, the new value is inserted at the end of the queue. The queue has three empty spaces remaining.

Queue (3)



Task B reads (receives) from the queue into a different variable. The value received by Task B is the value from the head of the queue, which is the first value Task A wrote to the queue (10 in this illustration).



Task B has removed one item, leaving only the second value written by Task A remaining in the queue. This is the value Task B would receive next if it read from the queue again. The queue now has four empty spaces remaining.

Le code funzionano copiando un elemento nella propria memoria (no referenze)

Queue (4)

QueueHandle_t **xQueueCreate**(UBaseType_t **uxQueueLength**,
UBaseType_t **uxItemSize**);

BaseType_t **xQueueSendToFront**(QueueHandle_t **xQueue**,
const void * **pvItemToQueue**, TickType_t xTicksToWait);

BaseType_t **xQueueSendToBack**(QueueHandle_t **xQueue**,
const void * **pvItemToQueue**, TickType_t xTicksToWait);

BaseType_t **xQueueReceive**(QueueHandle_t **xQueue**, void * const **pvBuffer**,
TickType_t xTicksToWait);

Queue con puntatori

Quando la dimensione dei dati da salvare è notevole è possibile passare un puntatore alla coda. Questo introduce delle difficoltà:

- La memoria non va mai modifica simultaneamente rendendo la stessa **inconsistente.**
- Se la memoria è allocata dinamicamente allora un task deve essere responsabile della **deallocazione** di essa.
- Non bisogna mai passare puntatori a della memoria allocata all'interno del singolo task. La memoria verrà liberata se il task verrà eliminato.

Software timer

I software timer sono utilizzati per schedulare delle operazioni nel futuro.

Non bisogna confonderli con i timer hardware ma non utilizzano nessuno ciclo di clock per gestire l'attesa.

È necessario includere il file **<timer.c>** nel proprio progetto e configurare la variabile configUSE_TIMERS = 1 nel file FreeRTOSConfig.h

Mai utilizzare codice bloccante all'interno di un Timer!

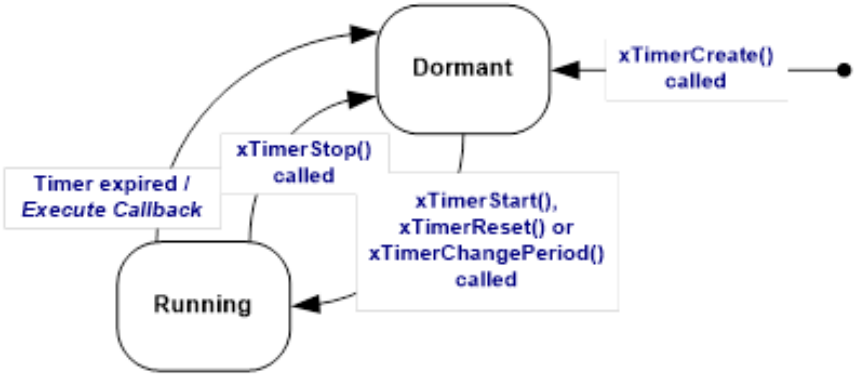
Software timer (2)

- **One shot**

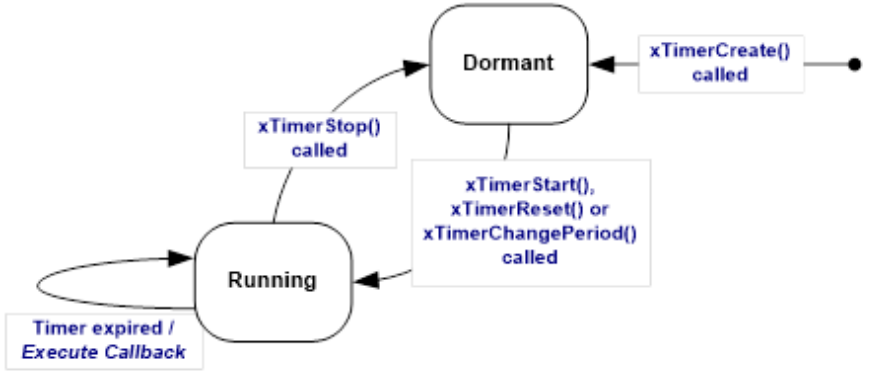
  Questo tipo di timer vengono eseguiti **una sola volta**. Un one-shot timer può essere riavviato manualmente.

- **Auto-reloaded**

  Un timer di questo genere viene rieseguito a cadenza prefissata chiamando la sua callback function.



One Shot



Auto-reloaded

Software timers (3)

TimerHandle_t **xTimerCreate**(const char * const **pcTimerName**,
TickleType_t **xTimerPeriodInTicks**,
UBaseType_t **uxAutoreload**, void * **pvTimerID**,
TimerCallBackFunction_t **pxCallbackFunction**);

TimerHandle_t **xTimerStart**(TimerHandle_t **xTimer**, TickType_t **xTicksToWait**);

TimerHandle_t **xTimerReset**(TimerHandle_t **xTimer**, TickType_t **xTicksToWait**);

TimerHandle_t **xTimerStop**(TimerHandle_t **xTimer**, TickType_t **xTicksToWait**);

TimerHandle_t **xTimerChangePeriod**(TimerHandle_t **xTimer**,
TickType_t **xNewTimerPeriodInTicks**, TickType_t **xTicksToWait**);

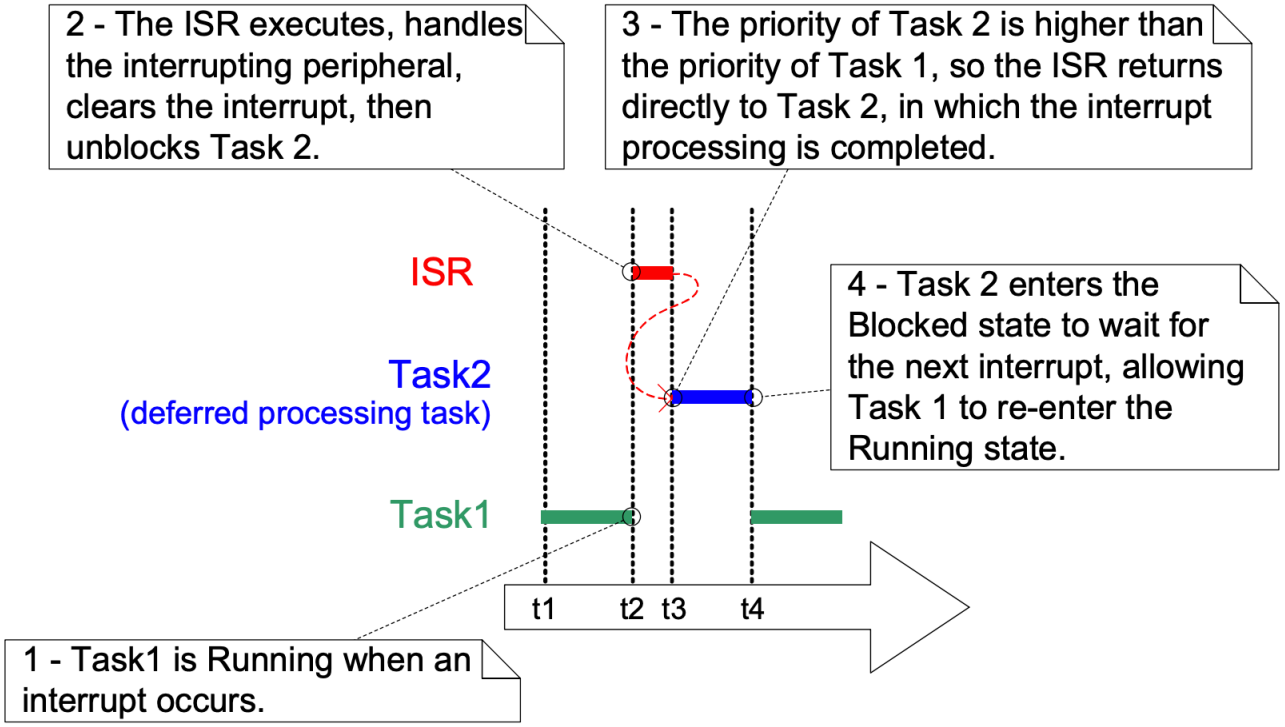
Interrupt

Nel mondo embedded gli interreput sono detti interrupt service routine (ISR). Sono funzionalità fornite dall'**hardware** perché è l'hw stesso a definire quale ISR eseguire e quando. FreeRTOS fornisce librerie specifiche da utilizzare all'interno degli interrupt **(FROM_ISR).**

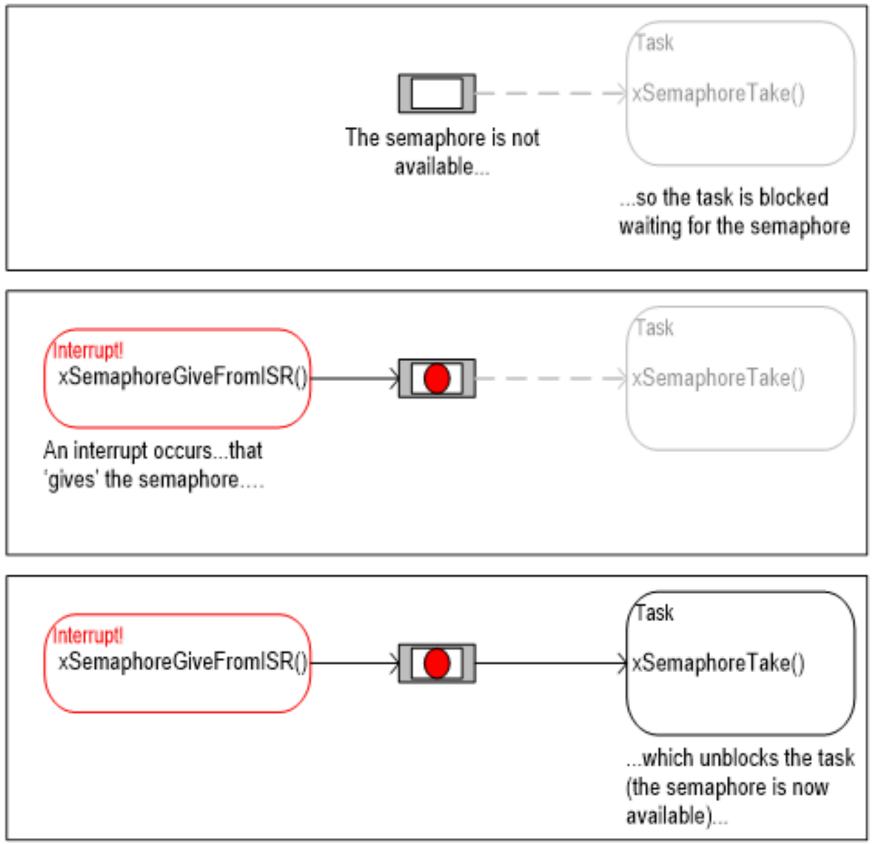
- Non è consigliato effettuare operazioni complicate
- Gli interrupt non sono deterministici
- Il lavoro principale va delegato a un task

I semafori e i mutex sono usati come strumenti di sincronizzazione

Interrupt (2)


2 - The ISR executes, handles the interrupting peripheral, clears the interrupt, then unblocks Task 2.
3 - The priority of Task 2 is higher than the priority of Task 1, so the ISR returns directly to Task 2, in which the interrupt processing is completed.
ISR
Task2
(deferred processing task)
4 - Task 2 enters the Blocked state to wait for the next interrupt, allowing Task 1 to re-enter the Running state.
Task1
t1
t2
t3
t4
1 - Task1 is Running when an interrupt occurs.

Interrupt (3)

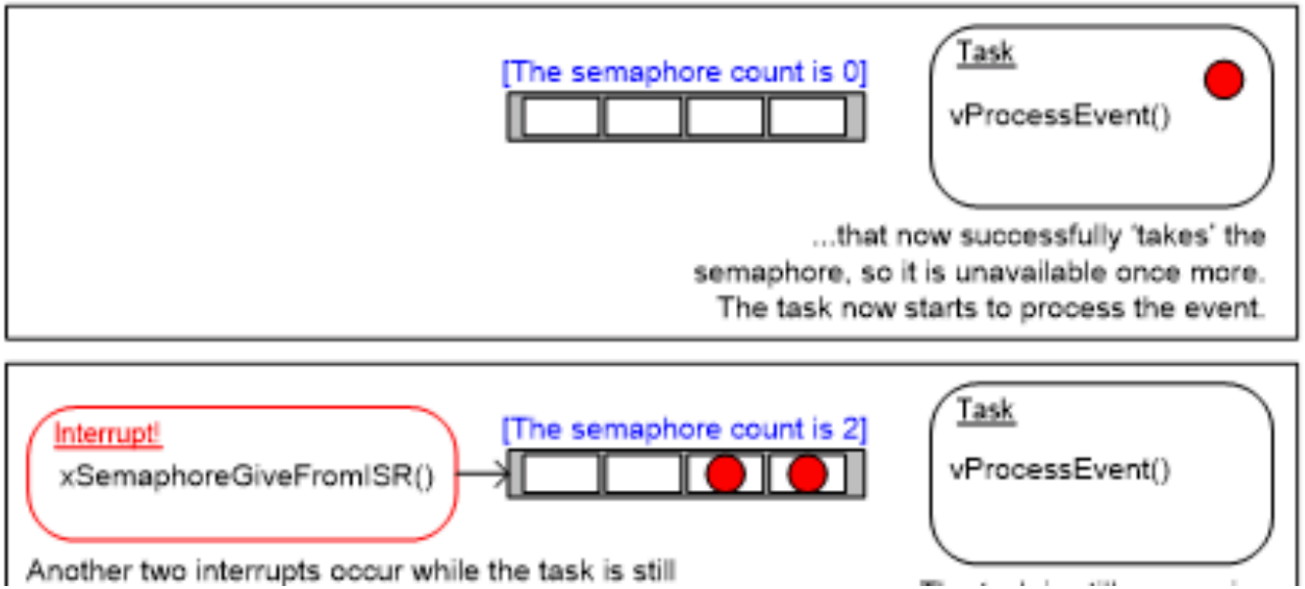


Abbiamo un problema.
Sapete quale?

Interrupt (4)


[The semaphore count is 0]
Task
xSemaphoreTake()
The task is blocked waiting for a semaphore
Interrupt!
xSemaphoreGiveFromISR()
[The semaphore count is 1]
Task
xSemaphoreTake()
An interrupt occurs...that 'gives' the semaphore....
Interrupt!
xSemaphoreGiveFromISR()
[The semaphore count is 1]
Task
xSemaphoreTake()
...which unblocks the task (the semaphore is now available)...

Interrupt (5)



Possiamo utilizzare un altro strumento di sincronizzazione che abbiamo già visto?

Interrupt (6)

È possibile sincronizzare una ISR con diversi metodi:

- Binary Semaphore
- Counting Semaphore
- Mutex *(è meglio utilizzare i semafori binari)*
- Queue

Semafori binari e con contatore

SemaphoreHandle_t **xSemaphoreCreateBinary**(void);

SemaphoreHandle_t **xSemaphoreCreateCounting**(UBaseType_t uxMaxCount, UBaseType_t uxInitialCount);

SemaphoreHandle_t **xSemaphoreCreateMutex**(void);

*I **mutex** e **semafori binari** sono molto simili ma di solito si usano i semafori binari come **strumento di sincronizzazione** e i mutex come **strumento di mutua esclusione***

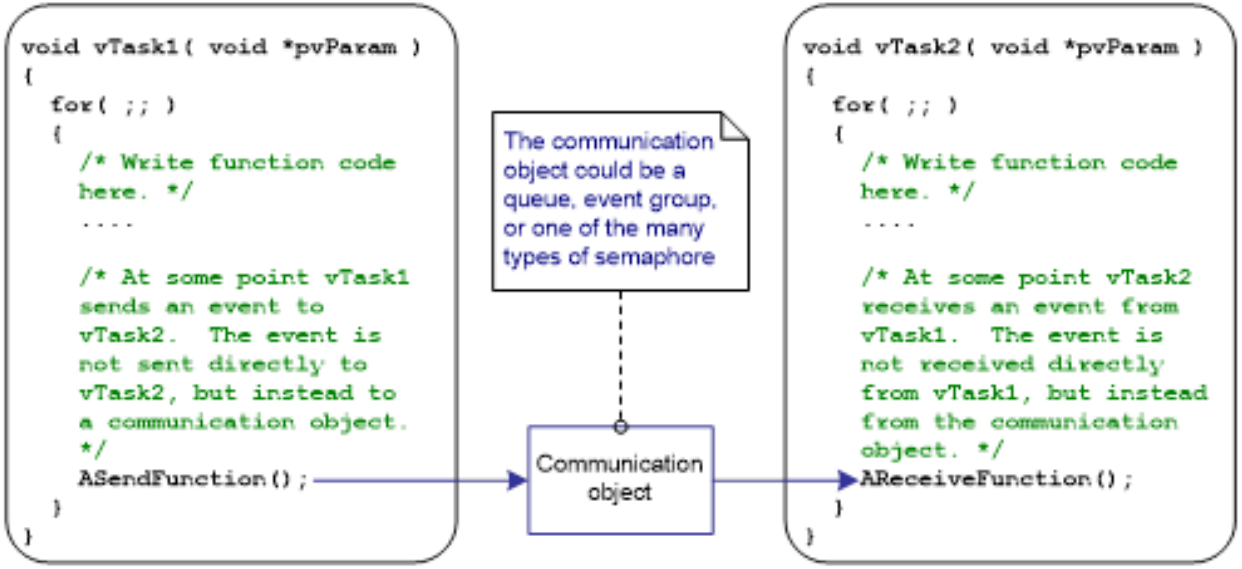
BaseType_t **xSemaphoreTake**(SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait);

BaseType_t **xSemaphoreGive**(SemaphoreHandle_t xSemaphore);

BaseType_t **xSemaphoreGiveFromISR**(SemaphoreHandle_t xSemaphore, BaseType_t *pxHigherPriorityTaskWoken);

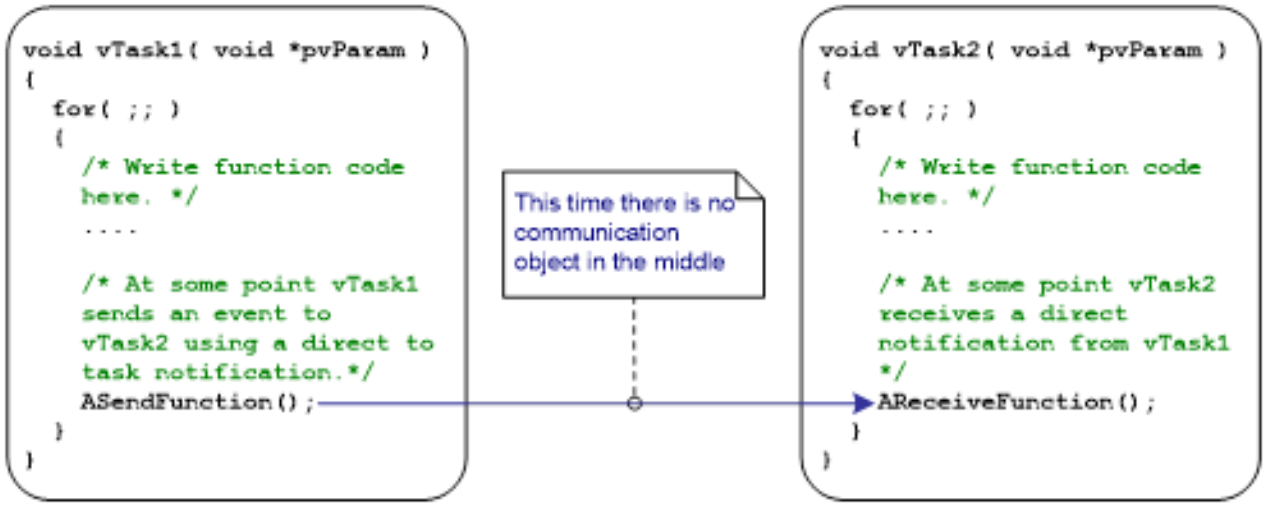
Task notifications

Due task possono parlare solamente tramite degli oggetti intermediari



Task notifications (2)

I task notifications permettono di comunicare tra task senza oggetti intermediari e sincronizzarsi direttamente con gli interrupt



Task notifications (3)

Vengono abilitati tramite il **configUSE_TASK_NOTIFICATIONS = 1** in **FreeRTOSConfig.h**

Quando vengono attivati ogni task ha uno stato: **pending** o **not pending**. Quando una notifica è ricevuta lo stato del task passa a pending.

È un metodo più veloce di inviare dati o eventi ad un altro task rispetto a code e semafori.

- Comunicazione tra **ISR a task** o **task to task.**
- La notifica è inviata direttamente al task ricevente, quindi **può essere processata solo dal task che ha ricevuto la notifica.**
- Una notifica può contenere un elemento alla volta.

Task notifications (4)

BaseType_t **xTaskNotifyGive**(TaskHandle_t **xTaskToNotify**);

BaseType_t **xTaskNotifyGiveFromISR**(TaskHandle_t **xTaskToNotify,**
BaseType_t *pxHigherPriorityTaskWoken);

uint32_t **ulTaskNotifyTake**(BaseType_t **xClearCountOnExit**,
TickType_t xTicksToWait);

Altre funzioni disponibili **xTaskNotify**, **xTaskNotifyFromISR**,
xTaskNotifyWait.

MPI

Per la prossima lezione avrete bisogno di **OPENMPI v3**.

I sw che svilupperemo sono compatibili anche con INTELMPI e MPICH ma la libreria di riferimento sarà OPENMPI.

Per scaricare il sw e installarlo fare riferimento a:

https://github.com/open-mpi/ompi

Non ci sarà bisogno di hw esterno. Ognuno di voi potrà programmare direttamente dal proprio laptop.