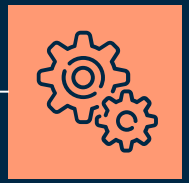
Free RTOS

TABLE OF CONTENTS

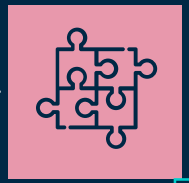
01

CARATTERISTICHE PRINCIPALI



02

CONFRONTO CON OS161



03

IMPLEMENTAZIONE DELLE CONDITION VARIABLES

01

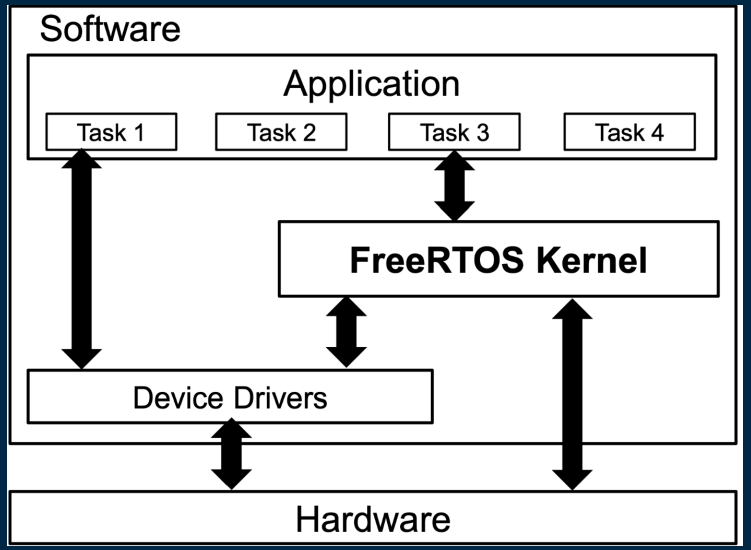
CARATTERISTICHE PRINCIPALI

CARATTERISTICHE GENERALI

- Sistema operativo in tempo reale (RTOS)
- Ottimizzato per sistemi con risorse limitate come microcontrollori e microprocessori
- Open source

ARCHITETTURA

- Basata su kernel e thread preemptive (noti come task)
- Small and Simple, dotato di flessibilità e affidabilità;
- Il Kernel consiste in tre file C:
- Task.c
- List.c
- Queue.c
- Altri 3 opzionali: timers.c; coroutine.c e event_groups.c



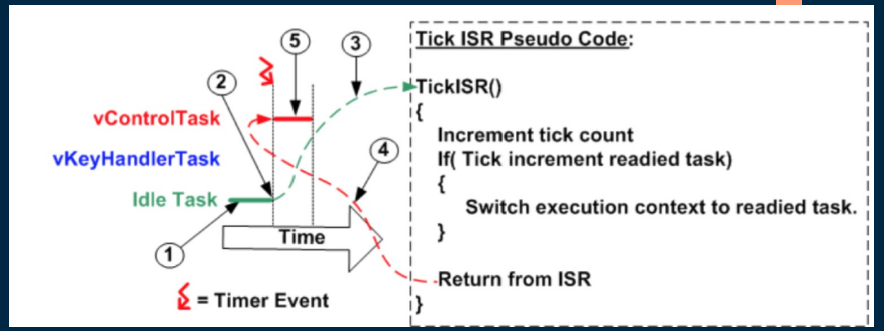
TASK

- Unità fondamentale di esecuzione (equivalente al thread in OS161)
- Può essere programmato per eseguire un'attività specifica
- Dotato di proprio stack
- Dotato di priorità:
  - Da 0 a MAX_PRIORITIES-1
  - Task in running ha priorità più alta
  - Task con stessa priorità condividono l'utilizzo della CPU usando un time sliced round robin scheduling scheme

STATI DEL TASK

TICK

- Variabile di conteggio usata dal kernel per misurare il tempo
- Quando scatta il tick interrupt, la variabile tick viene incrementata dalla TickISR, ad una risoluzione che dipende dalla frequenza del tick interrupt (tick_rate).
- Se ci sono task da eseguire con priorità maggiore del task in esecuzione, la TickISR avvia il cambio di contesto.



CONFRONTO CON OS161

02

OBIETTIVI

OS161

Sistema puramente didattico progettato per apprendere i concetti fondamentali dei sistemi operativi

FREERTOS

Sistema operativo in tempo reale utilizzato sia per scopi didattici che in ambienti lavorativi per sistemi con risorse limitate

ARCHITETTURA

OS161

- Kernel monolitico
- Architettura semplice e modulare, offrendo funzionalità di base come la gestione dei processi, thread, file e della memoria virtuale

FREERTOS

- Si basa su un kernel di scheduling prioritario supportando la programmazione multitasking offrendo funzionalità per la gestione delle code, semafori e mutex

ARCHITETTURA

OS161

- Kernel monolitico
- Architettura semplice e modulare, offrendo funzionalità di base come la gestione dei processi, thread, file e della memoria virtuale

FREERTOS

- Si basa su un kernel di scheduling prioritario supportando la programmazione multitasking offrendo funzionalità per la gestione delle code, semafori e mutex

PORTABILITÀ

OS161

- Altamente portabile

FREERTOS

- Principalmente eseguito su macchine virtuali simulate

MEMORY MANAGEMENT

OS161

- Gestione della memoria virtuale e fornisce un'implementazione semplificata di un MMU

FREERTOS

- Senza supporto per la memoria virtuale o la gestione dell'MMU -> Allocazione Contigua

Alla creazione di un task, semaforo o coda il kernel deve allocare memoria
Le standard malloc e free possono essere utilizzate ma:

- Non sono sempre disponibili nei piccoli sistemi embedded
- Non sono deterministiche
- Possono soffrire di frammentazione della memoria
- La loro implementazione può essere ampia

MEMORY MANAGEMENT

- FreeRTOS tratta l'allocazione della memoria come parte del suo portable layer
- Allocazione dinamica/statica dell'heap
- 5 differenti implementazioni disponibili per l'Heap Management:
- Possibilità di definire la propria implementazione

CONFIGURAZIONI DELL'HEAP

- heap_1.c
  - Permette la sola allocazione della memoria
  - Usata quando le applicazioni non rimuovono mai risorse (task, semafori, code)
- heap_2.c
  - Permette la deallocazione della memoria, ma non gestisce la frammentazione
  - Usata quando le applicazioni rimuovono risorse continuamente
  - Da evitare quando la quantità di stack allocata ai task non è sempre la stessa
- heap_3.c
  - Prevede dei wrapper della malloc() e della free() del C per renderle thread-safe
- heap_4.c
  - Come heap_2.c, ma prevede la gestione della frammentazione
- heap_5.c
  - Come heap_4.c, ma permette di allocare l'heap in regioni non contigue di memoria

SCHEDULING

- OS161 utilizza un algoritmo di scheduling a priorità fissa con una politica di esecuzione round-robin.
  Il sistema operativo fornisce anche la possibilità di modificare la priorità dei processi durante l'esecuzione.

- FreeRTOS utilizza un kernel di scheduling prioritario predefinito:
  I task vengono schedulati in base alla loro priorità, con la possibilità di gestire anche la preemption tra task con priorità diversa.

SCHEDULING

Scheduling preventivo a priorità fissa, con round-robin time-slicing di attività a parità di priorità:

- Priorità fissa : lo scheduler non cambierà la priorità di un'attività, anche se potrebbe aumentarla temporaneamente a causa del Priority Inheritance.
- Preventivo: lo scheduler esegue sempre l'attività RTOS con la massima priorità che è in grado di eseguire, indipendentemente da quando quest'attività è in grado di essere eseguita.
- Round-robin: le attività che condividono una priorità entrano a turno nello stato di esecuzione.
- Time sliced: lo scheduler passerà tra attività di uguale priorità su ogni interruzione del tick - il tempo tra gli interrupt del tick è una fetta di tempo. (Il tick interrupt è l'interrupt periodico utilizzato dall'RTOS per misurare il tempo.)

SCHEDULING MULTI-CORE

La politica di scheduling di freeRTOS può essere simmetrica o **asimmetrica**:

- Multiprocessing asimmetrico (AMP)
  - Luogo in cui ogni core di un dispositivo multicore esegue la propria istanza indipendente di FreeRTOS
  - Devono condividere un po' di memoria se le istanze FreeRTOS devono comunicare tra loro
  - L'algoritmo di pianificazione su un dato core è esattamente a quello adattato per un sistema single-core
  - È possibile utilizzare un flusso o un buffer di messaggi come primitiva di comunicazione inter-core in modo che le attività su un core possano entrare nello stato Bloccato per attendere che i dati o gli eventi vengano inviati da un core diverso.

SCHEDULING MULTI-CORE

La politica di scheduling di freeRTOS può essere **simmetrica** o asimmetrica:

- Multiprocessing simmetrico (SMP)
  - Un'istanza di FreeRTOS pianifica le attività RTOS su più core del processore
  - Poiché c'è solo un'istanza in esecuzione, è possibile utilizzare solo una porta di FreeRTOS alla volta
  - Il criterio di scheduling SMP utilizza lo stesso algoritmo del criterio di scheduling single-core, ma comporta che più di un'attività sia nello stato di esecuzione in un dato momento (c'è un'attività di stato di esecuzione per core).
  - L'ipotesi che un'attività a priorità inferiore verrà eseguita solo quando non ci sono attività a priorità superiore in grado di eseguire non vale più

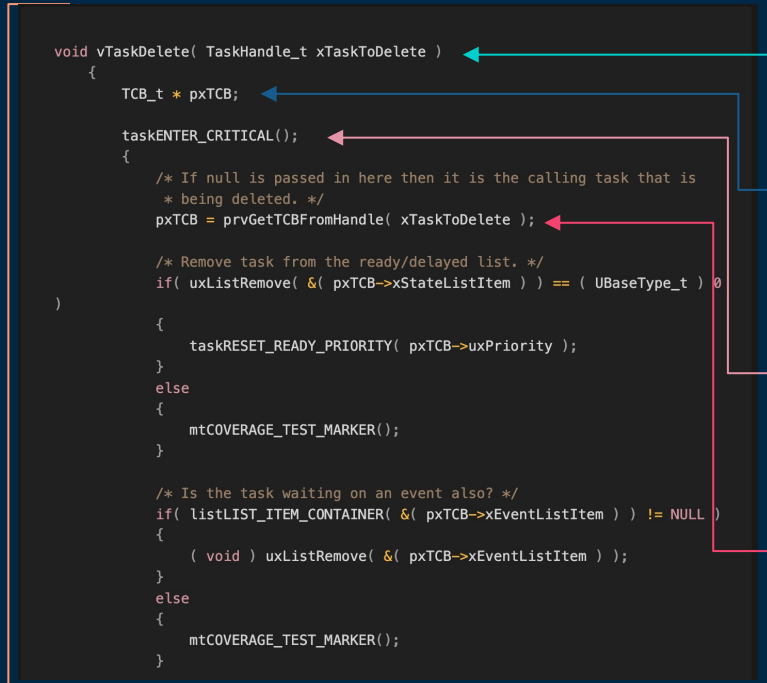
SYSTEM CALLS

- FreeRTOS fornisce le proprie API al posto delle syscall per la gestione, sincronizzazione e comunicazione tra task e altre funzionalità specifiche per i sistemi in tempo reale

- SYS_read e SYS_write possono essere implementate utilizzando API fornite da Queue.h, ovvero facendo uso di una coda, che può essere utilizzata per trasmettere o ricevere dati tra task o per comunicare con un dispositivo di I/O.

- Per la SYS_exit al suo posto viene utilizzata la funzione **vTaskDelete()** all'interno del task stesso per terminare un task in FreeRTOS, permettendo al task di terminare la sua esecuzione e liberare le risorse associate ad esso.

SYSTEM CALLS: SYS_EXIT



```
void vTaskDelete( TaskHandle_t xTaskToDelete )
    {
        TCB_t * pxTCB;

        taskENTER_CRITICAL();
        {
            /* If null is passed in here then it is the calling task that is
             * being deleted. */
            pxTCB = prvGetTCBFromHandle( xTaskToDelete );

            /* Remove task from the ready/delayed list. */
            if( uxListRemove( &( pxTCB->xStateListItem ) ) == ( UBaseType_t ) 0
)
            {
                taskRESET_READY_PRIORITY( pxTCB->uxPriority );
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }

            /* Is the task waiting on an event also? */
            if( listLIST_ITEM_CONTAINER( &( pxTCB->xEventListItem ) ) != NULL )
            {
                ( void ) uxListRemove( &( pxTCB->xEventListItem ) );
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
```

Viene preso come argomento xTaskToDelete, handler del task (NULL se si riferisce al task corrente)

Viene dichiarato un puntatore a una struttura TCB_t, che rappresenta il control block dell'attività da eliminare

Viene richiamata la macro taskENTER_CRITICAL() per entrare in una sezione critica. Questo impedisce l'interferenza di altre attività o operazioni di scheduling mentre l'attività viene eliminata.

Ottiene un puntatore al control block dell'attività da eliminare utilizzando la funzione prvGetTCBFromHandle(). Questa funzione recupera il control block associato a una TaskHandle_t (un identificatore di attività) specifica.

SYSTEM CALLS: SYS_EXIT

```
void vTaskDelete( TaskHandle_t xTaskToDelete )
    {
        TCB_t * pxTCB;

        taskENTER_CRITICAL();
        {
            /* If null is passed in here then it is the calling task that is
             * being deleted. */
            pxTCB = prvGetTCBFromHandle( xTaskToDelete );

            /* Remove task from the ready/delayed list. */
            if( uxListRemove( &( pxTCB->xStateListItem ) ) == ( UBaseType_t ) 0
)
            {
                taskRESET_READY_PRIORITY( pxTCB->uxPriority );
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }

            /* Is the task waiting on an event also? */
            if( listLIST_ITEM_CONTAINER( &( pxTCB->xEventListItem ) ) != NULL )
            {
                ( void ) uxListRemove( &( pxTCB->xEventListItem ) );
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
```

Questa istruzione rimuove l'attività dalla lista delle attività pronte o in attesa (xStateListItem) usando uxListRemove(). Se il risultato della rimozione è 0, significa che l'attività era nella lista delle attività pronte, quindi viene richiamato taskRESET_READY_PRIORITY() per reimpostare la priorità dell'attività.

verifica se l'attività è in attesa di un evento (utilizzando la lista xEventListItem) e, se lo è, la rimuove dalla lista.

SYSTEM CALLS: SYS_EXIT

```
    uxTaskNumber++;

    if( pxTCB == pxCurrentTCB )
    {
        vListInsertEnd( &xTasksWaitingTermination, &( pxTCB->xStateListItem ) );

        /* Increment the ucTasksDeleted variable so the idle task knows
         * there is a task that has been deleted and that it should therefore
         * check the xTasksWaitingTermination list. */
        ++uxDeletedTasksWaitingCleanUp;

        /* Call the delete hook before portPRE_TASK_DELETE_HOOK() as
         * portPRE_TASK_DELETE_HOOK() does not return in the Win32 port. */
        traceTASK_DELETE( pxTCB );

        /* The pre-delete hook is primarily for the Windows simulator,
         * in which Windows specific clean up operations are performed,
         * after which it is not possible to yield away from this task -
         * hence xYieldPending is used to latch that a context switch is
         * required. */
        portPRE_TASK_DELETE_HOOK( pxTCB, &xYieldPending );
    }
    else
    {
        --uxCurrentNumberOfTasks;
        traceTASK_DELETE( pxTCB );

        /* Reset the next expected unblock time in case it referred to
         * the task that has just been deleted. */
        prvResetNextTaskUnblockTime();
    }
}
```

Incrementa uxTaskNumber, che è utilizzato da strumenti di debug per rilevare quando le liste delle attività devono essere rigenerate.

Verifica se l'attività sta eliminando se stessa. Se lo è, viene inserita nella lista delle attività in attesa di terminazione (xTasksWaitingTermination) e viene incrementata uxDeletedTasksWaitingCleanUp, che è utilizzata dall'Idle task per rilevare attività da eliminare, poi richiama una hook specifica della porta, che può essere utilizzata per effettuare operazioni di pulizia o gestione specifiche della piattaforma prima dell'eliminazione dell'attività.

Altrimenti decrementa il numero totale di attività attualmente in esecuzione e reimposta il tempo di sblocco dell'attività successiva nel caso in cui si riferisse all'attività appena eliminata con **prvResetNextTaskUnblockTime().**

SYSTEM CALLS: SYS_EXIT

```
    taskEXIT_CRITICAL();

    /* If the task is not deleting itself, call prvDeleteTCB from outside of
     * critical section. If a task deletes itself, prvDeleteTCB is called
     * from prvCheckTasksWaitingTermination which is called from Idle task. */
    if( pxTCB != pxCurrentTCB )
    {
        prvDeleteTCB( pxTCB );
    }

    /* Force a reschedule if it is the currently running task that has just
     * been deleted. */
    if( xSchedulerRunning != pdFALSE )
    {
        if( pxTCB == pxCurrentTCB )
        {
            configASSERT( uxSchedulerSuspended == ( UBaseType_t ) 0U );
            portYIELD_WITHIN_API();
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
}
```

Fuori dalla sezione critica (**taskEXIT_CRITICAL()**), viene chiamata **prvDeleteTCB(pxTCB)** per effettivamente eliminare il control block dell'attività e gestire la memoria associata a essa

Se il pianificatore è in esecuzione (xSchedulerRunning != pdFALSE), viene verificato se l'attività eliminata è quella in esecuzione (pxTCB == pxCurrentTCB). In caso affermativo, viene forzata una reschedule chiamando portYIELD_WITHIN_API() per cedere il controllo a un'altra attività.

MECCANISMI DI SINCRONIZZAZIONE

OS161

- Spinlock e Semafori già implementati.
- Manca il supporto per lock e cpndition variables

FREERTOS

- Semafori binarti
- Semafori a conteggio
- Mutex
  - Ricorsivi
  - Priority Inheritance

SEMAFORI BINARI

La funzione xSemaphoreCreateBinary crea un semaforo binario, ritornando un handle attraverso il quale si può far riferimento al semaforo

```
//semphr.h
#define xSemaphoreCreateBinary() xQueueGenericCreate( ( UBaseType_t ) 1, semSEMA
PHORE_QUEUE_ITEM_LENGTH, queueQUEUE_TYPE_BINARY_SEMAPHORE )

//interfaccia effettiva
SemaphoreHandle_t xSemaphoreCreateBinary(void);
```

Si nota come, per implementare i semafori, freeRTOS adatta la struttura già esistente di queue , usata per la comunicazione tra processi. La lunghezza della coda è 1 perché si tratta di un semaforo binario

L'interfaccia effettiva dichiara una funzione xSemaphoreCreateBinary() che restituisce un **SemaphoreHandle_t**.

SEMAFORI A CONTEGGIO

```
// semphr.h
#define xSemaphoreCreateCounting( uxMaxCount, uxInitialCount )
        xQueueCreateCountingSemaphore( ( uxMaxCount ), ( uxInitialCount ) )
//interfaccia effettiva
SemaphoreHandle_t xSemaphoreCreateCounting(
UBaseType_t uxMaxCount, UBaseType_t uxInitCount)
```

Crea un semaforo di conteggio e restituisce un handle con il quale è possibile fare riferimento al semaforo appena creato.

- uxMaxCount: Massimo valore di conteggio
- uxInitCount: Valore di conteggio assegnato alla creazione

Usati per:

- Conteggio degli eventi
  - un gestore di eventi "darà" un semaforo ogni volta che si verifica un evento e un'attività del gestore "prenderà" un semaforo ogni volta che elabora un evento.

  Il valore di conteggio è quindi la differenza tra il numero di eventi che si sono verificati e il numero che sono stati elaborati.
- Gestione delle risorse

ELIMINARE UN SEMAFORO

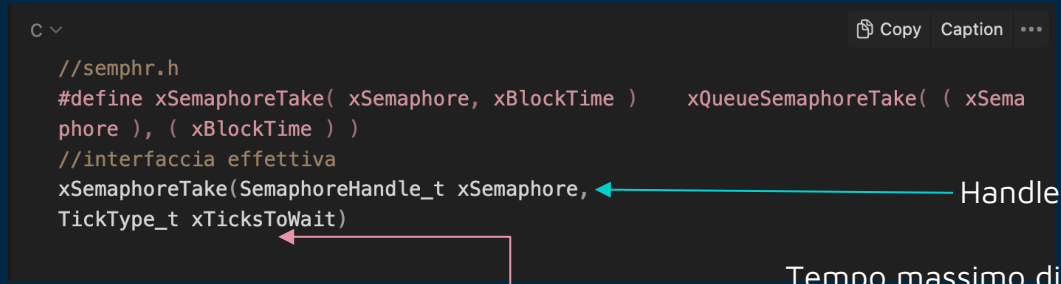
- Si usa la funzione vSemaphoreDelete

```
//semphr.h
#define vSemaphoreDelete( xSemaphore )    vQueueDelete( ( QueueHandle_t ) ( xSem
aphore))
//interfaccia effettiva
void vSemaphoreDelete(SemaphoreHandle_t xSemaphore)
```

- Non bisogna però elimInare un semaforo che ha attività bloccate su di esso
- Per eliminare il semaforo si fa uso anche in questo caso della struttura già esistente di queue, in particolare della funzione vQueueDelete()

OTTENERE UN SEMAFORO

- Si utilizza la macro xSemaphoreTake. Il semaforo deve essere stato creato in precedenza con una chiamata a xSemaphoreCreateBinary(), xSemaphoreCreateMutex() o xSemaphoreCreateCounting().



RILASCIARE UN SEMAFORO

- Si utilizza la macro xSemaphoreGive().
  Anche in questo caso il semaforo deve essere stato creato in precedenza.

```
//semphr.h
#define xSemaphoreGive( xSemaphore )    xQueueGenericSend( ( QueueHandle_t ) ( x
Semaphore ), NULL, semGIVE_BLOCK_TIME, queueSEND_TO_BACK )

//interfaccia effettiva
xSemaphoreGive(SemaphoreHandle_t xSemaphore)
```

- La macro xSemaphoreGive() è un wrapper per xQueueGenericSend(), una funzione interna di FreeRTOS utilizzata per inviare un messaggio o acquisire un semaforo all'interno di una coda.

MUTEX

```
//semphr.h
#define xSemaphoreCreateMutex()    xQueueCreateMutex( queueQUEUE_TYPE_MUTEX )
//interfaccia effettiva
SemaphoreHandle_t xSemaphoreCreateMutex(void)
```

- Viene utilizzata la funzione xQueueCreateMutex, essa riceve un parametro, queueQUEUE_TYPE_MUTEX, che specifica il tipo di semaforo che deve essere creato, ovvero un mutex
- Crea un mutex, ritornando un handle attraverso il quale si può far riferimento al mutex
  Esso è simile ai semafori binari, ma implementano il meccanismo di Priority Inheritance.
- I mutex non possono essere utilizzati nelle routine di servizio interrupt.

MUTEX RICORSIVI

- FreeRTOS offre anche la possibilità di creare un mutex ricorsivo con la funzione **xSemaphoreCreateRecursiveMutex.**
- Un mutex usato ricorsivamente può essere "preso" ripetutamente dal proprietario. Il mutex non diventa più disponibile fino a quando il proprietario non ha chiamato xSemaphoreGiveRecursive() per ogni richiesta xSemaphoreTakeRecursive() riuscita.

CONDITION VARIABLES

- Os161 offre un’interfaccia, ma non il supporto, per le condition variable. Sta quindi all’utilizzatore implementare le funzioni che garantiscono un corretto funzionamento di queste.
- In FreeRTOS, invece, il concetto di condition variable è del tutto assente.

IMPLEMENTAZIONE DELLE CONDITION VARIABLES

03

IMPLEMENTAZIONE DELLE CV

```
typedef QueueHandle_t SemaphoreHandle_t;
```

Viene definito un alias **SemaphoreHandle_t** per **QueueHandle_t**. Questo è comune in FreeRTOS, dove i semafori e le code condividono la stessa struttura dati di base

```
struct CondVarHandle_t{
SemaphoreHandle_t mutex;
SemaphoreHandle_t coda;
}
```

Viene definita una struttura che rappresenta un semaforo condizionale personalizzato.
Contiene due campi:

- Mutex: utilizzato per proteggere l'accesso alla coda condizionale
- Coda: rappresenta la coda condizionale stessa.

CREARE UN SEMAFORO CON CV

- Viene utilizzata la funzione xCondVarCreate()

```
#if ( configSUPPORT_DYNAMIC_ALLOCATION == 1 )
#define xCondVarCreate()
    {
        struct CondVarHandle_t xCondVar;
        (xCondVar->mutex) = xSemaphoreCreateMutex();
        (xCondVar->coda) = xSemaphoreCreateCounting(1,0);

           return xCondVar
    }
#endif
```

- Crea il semaforo mutex con xSemaphoreCreateMutex()
- Crea la coda condizionale con xSemaphoreCreateCounting()

ATTENDERE UN SEMAFORO CON CV

```
#define xCondVarWaitCondition(xCondVar, xBlockTime)
{
    xSemaphoreGive(xCondVar->mutex);
    xSemaphoreTake(xCondVar->coda, xBlockTime);
    xSemaphoreTake(xCondVar->mutex, xBlockTime);
}
```

Argomenti:

- xCondVar: il semaforo condizionale
- xBlockTime: tempo massimo di attesa.

Questa funzione rilascia il mutex utilizzando la funziona già definita in semphr.h xSemaphoreGive.
Questa macro quindi assume che l'attività abbia già acquisito questo mutex in precedenza.

RILASCIARE UN SEMAFORO CON CV

```
#define xCondVarNotify(xCondVar)
{
    xSemaphoreGive(xCondVar->coda);
    xSemaphoreGive(xCondVar->mutex);
}
```

Argomenti:

- xCondVar: il semaforo condizionale

Rilascia il semaforo di conteggio chiamando xSemaphoreGive(xCondVar->coda). Il rilascio di questo semaforo indica che una condizione specifica all'interno del semaforo condizionale è stata soddisfatta, consentendo alle attività in attesa di procedere

Tramite la xSemaphoreGive(xCondVar->mutex) viene rilasciato anche il mutex

ELIMINARE UN SEMAFORO CON CV

```
#define vCondVarDelete(xCondVar)
{
    vSemaphoreDelete(xCondVar->mutex);
    vSemaphoreDelete(xCondVar->coda);
}
```

- Prende come argomento xCondVar,la struttura CondVarHandle_t che rappresenta il semaforo condizionale personalizzato da eliminare
- vSemaphoreDelete(xCondVar->mutex) elimina il mutex associato al semaforo condizionale
- vSemaphoreDelete(xCondVar->coda) elimina il semaforo di conteggio associato alla coda condizionale

FINE