

# Progetto di un'applicazione basata su kernel FreeRTOS per il controllo automatico di un umidificatore e confronto con versione Bare Metal

Il presente documento si propone di illustrare il progetto di un'applicazione per la scheda **STM32F303VC**, finalizzato alla realizzazione di un **umidificatore automatico**. L'obiettivo principale di questo progetto è quello di sviluppare un'applicazione basata sul kernel **FreeRTOS** per la gestione efficace dei task associati a sensori di temperatura e umidità, rilevamento del livello dell'acqua e controllo di un nebulizzatore per l'umidificazione dell'ambiente circostante. Sarà pertanto possibile gestire in modo efficiente i diversi task, garantendo un controllo del nebulizzatore alle variazioni dei livelli di umidità. Sarà inoltre sviluppata **un'analisi qualitativa** e comparativa tra la versione costruita con *FreeRTOS* e un'altra in *Bare Metal*.



Evaluation Board utilizzata: STM32F303VC

## Contents

### Panoramica del Sistema

### Architettura del Sistema

- DHT11 - Sensore di temperatura e umidità
  - Interconnessione pin DHT11 - STM32

- Water level - Sensore del livello dell'acqua
  - Interconnessione pin Water level - Sensore del livello dell'acqua - STM32

- Nebulizzatore
  - Interconnessione pin Nebulizer - STM32

- Architettura complessiva

### FreeRTOS

#### Tasks

- Ciclo di vita di un task
  - Gestione delle priorità
- Scheduling sulla base delle priorità
- System Tick
  - Gestione della temporizzazione e requisiti real-time
  - Configurazione dei timer
- Intercomunicazione tra i task
  - Gestione delle code
  - Configurazione delle code
- Sincronizzazione tra i task: semafori binari e mutex
  - Gestione del semaforo binario
  - Configurazione del semaforo binario
- Gestione della memoria
  - Allocazione dinamica dello stack: heap\_4.c

- Applicazione e debug
  - Utilizzo delle API Hardware Abstraction Layer

- Implementazione: Air Humidifier

- Codice
  - Main
  - DHT11 Task
  - Water Level Task
  - Handler Nebulizer Task
  - Nebulizer Task
- Makefile per la compilazione completa

- Tool per il debug
  - Toolchain ARM: GDB
  - OpenOCD
- Verifica della gestione delle priorità e sincronizzazione
- Verifica dello scambio di messaggi tramite coda

- Confronto dell'applicazione nella versione con FreeRTOS e Bare Metal

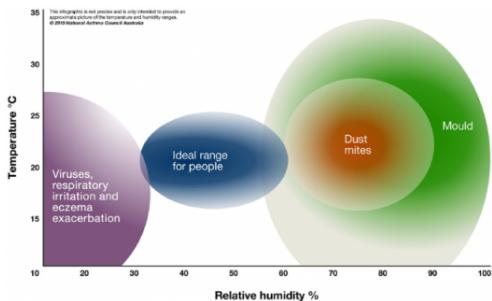
- Tempi di esecuzione
- Utilizzo della memoria
- Tempi di compilazione
- Analisi qualitativa

- Conclusioni

### Bibliografia

## Panoramica del Sistema

Il sistema è composto da un sensore di temperatura e umidità, un sensore di livello dell'acqua e un nebulizzatore. L'obiettivo del sistema è mantenere livelli di umidità ottimali nell'ambiente circostante attivando il nebulizzatore in maniera automatica e portando dopo un certo tempo di transizione l'umidità nell'ambiente intorno alle **percentuali** indicate nel seguente *infographic* della National Asthma Council Australia.



Infographic della National Asthma Council Australia utilizzato per la scelta dei range di abilitazione e disabilitazione del nebulizzatore

Il sensore di temperatura e umidità, pertanto, **monitorano costantemente i parametri ambientali**, consentendo al sistema di adattare l'operazione del nebulizzatore di conseguenza. Il sensore di livello dell'acqua evita il funzionamento del nebulizzatore quando il livello dell'acqua è insufficiente. Il nebulizzatore, controllato, vaporizza l'acqua per aumentare l'umidità nell'ambiente. Questo sistema automatizzato offre un ambiente più sano e confortevole per gli utenti.

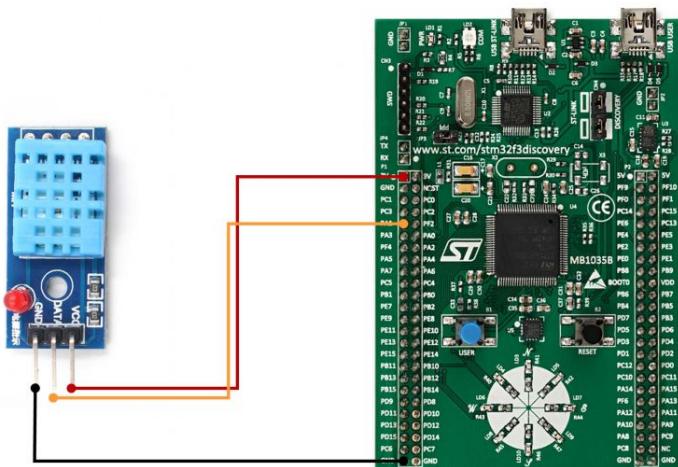
## Architettura del Sistema

Si descrivono di seguito i componenti utilizzati per il progetto in esame:

### DHT11 - Sensore di temperatura e umidità

Utilizza un sensore di umidità capacitivo e un termistore per misurare l'aria circostante ed emette un segnale digitale sul pin dati (non sono necessari pin di ingresso analogico). È abbastanza semplice da usare, ma richiede un **protocollo** (<https://www.mouser.com/datasheet/2/758/DHT11-Technical-Data-Sheet-Translate-d-Version-1143054.pdf>) e una tempificazione accurata tale per acquisire i dati. È dunque doveroso utilizzare un timer per fornire una base dei tempi. Si è scelto a tale scopo il **TIM7** (Basic Timers).

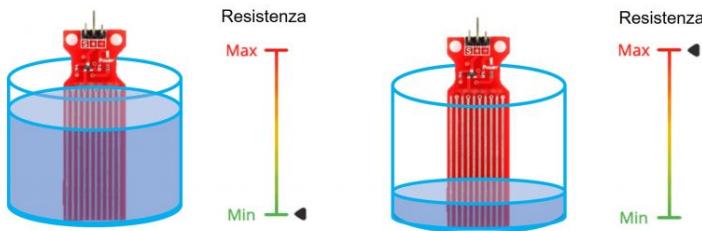
### Interconnessione pin DHT11 - STM32



Interconnessione pin DHT11

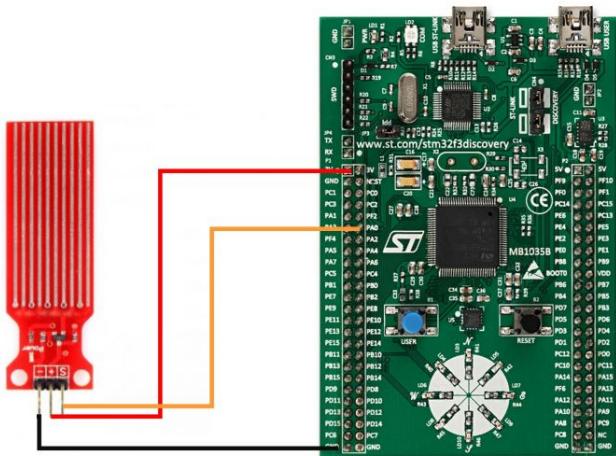
### Water level - Sensore del livello dell'acqua

Si tratta di un sensore di livello dell'acqua di tipo conduttivo, in cui la variazione della resistenza di fili paralleli su diverse profondità dell'acqua viene convertita in una tensione. Il Sensore ha una serie di dieci «Traces» parallele di rame, cinque delle quali sono traces di alimentazioni e altre sono traces sensoriali. Queste traces non sono collegate inizialmente, ma appena sono sommersi in acqua si crea un collegamento tra di esse che genera una variazione di resistenza. Come dimostrato nella seguente Figura:



Funzionamento concettuale del Water Level

#### Interconnessione pin Water level - Sensore del livello dell'acqua - STM32

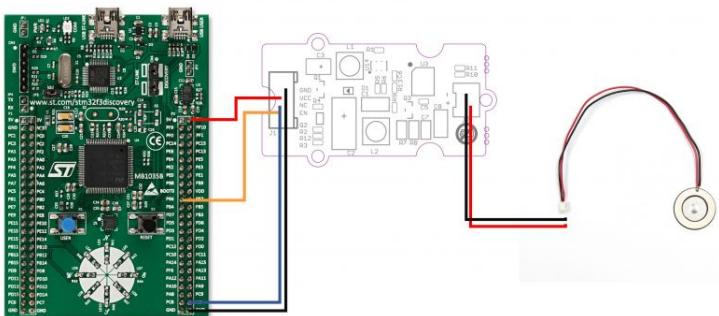


Interconnessione Water level

#### Nebulizzatore

Il Nebulizzatore è un atomizzatore ad ultrasuoni, non fa altro che ridurre un liquido in minutissime goccioline. Per usarlo, si fa uso dell'interfaccia **GROVE**.

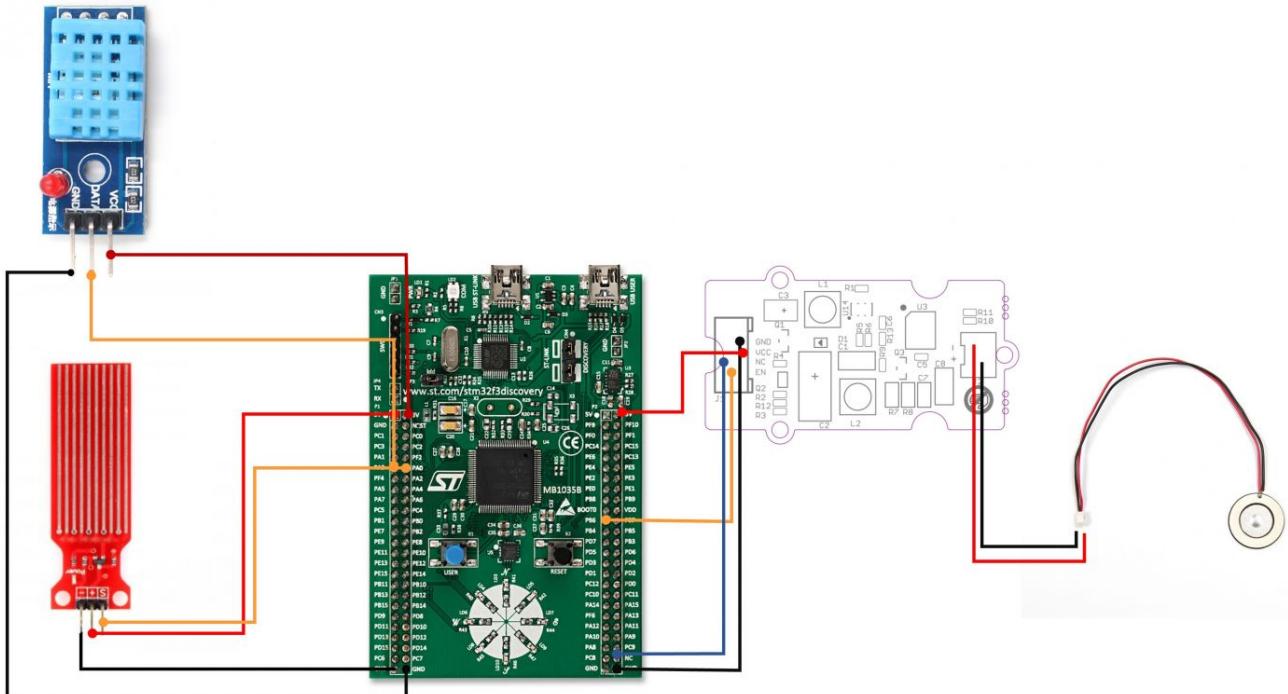
#### Interconnessione pin Nebulizer - STM32



Interconnessione Nebulizer sulla STM32F3Discovery Board

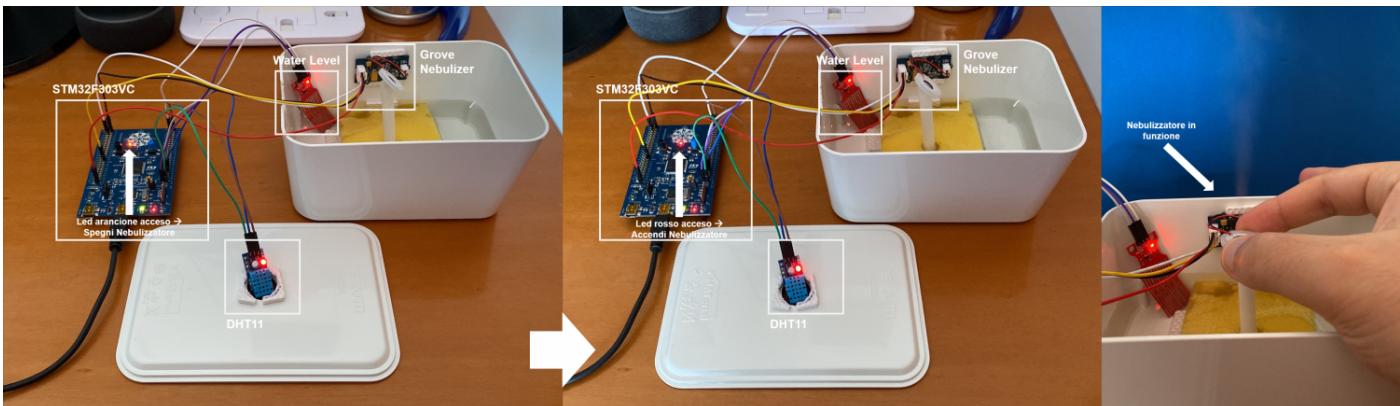
#### Architettura complessiva

Collegando opportunamente tutte le componenti si mostra l'architettura complessiva del sistema:



## Architettura complessiva del sistema

Si mostra inoltre una **panoramica di funzionamento** generale del sistema:



### Il sistema in funzionamento

# FreeRTOS

FreeRTOS è un sistema operativo in tempo reale (RTOS). Distribuito gratuitamente con licenza open source MIT, FreeRTOS include un kernel e un set crescente di librerie adatte all'uso in tutti i settori industriali, grazie alla particolare attenzione all'affidabilità, all'accessibilità e alla facilità d'uso.

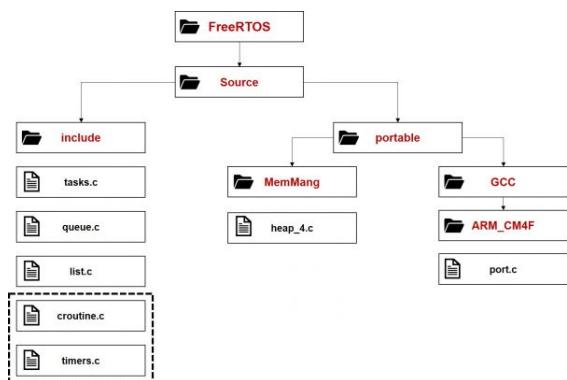
Il codice **FreeRTOS** di base è contenuto in tre file, chiamati

- *tasks.c*,
  - *queue.c*,
  - *list.c*.

Questi tre file si trovano nella directory *FreeRTOS/Source*. La stessa directory contiene due file opzionali chiamati *timers.c* e *croutine.c* che implementano rispettivamente la funzionalità software timer e co-routine. Il Kernel FreeRTOS è diviso in due parti: la prima parte del kernel di FreeRTOS è detta **hardware-independent** in quanto non cambia rispetto all'architettura target su cui esegue. Ogni processore supportato richiede un ulteriore file contenente il codice per l'interfacciamento di FreeRTOS con la specifica architettura del processore stesso. Questa sezione del kernel di FreeRTOS è detta **hardware-dependent** e si trova nelle sottodirectory

*FreeRTOS/Source/Portable/[compiler]/[architecture]*, dove [compiler] e [architecture] sono il compilatore utilizzato per creare il port e l'architettura sulla quale si esegue il port, nel caso in esame rispettivamente **GCC** e **ARM\_CM4F** in quanto la evaluation board in uso prevede il processore ARM Cortex M4.

Anche gli schemi di **allocazione dell'heap** di esempio si trovano nel livello portatile di FreeRTOS. I vari file *heap\_x.c* per la gestione della memoria si trovano nella directory *FreeRTOS/Source/portable/MemMang*. La struttura del kernel di FreeRTOS adattata al caso in esame si mostra nella Figura a destra.



La struttura del Kernel di FreeRTOS adattata al caso in esame

Inoltre, FreeRTOS è totalmente **personalizzabile** utilizzando un file di configurazione chiamato **FreeRTOSConfig.h**. Ogni applicazione FreeRTOS deve avere un file di intestazione **FreeRTOSConfig.h** nel relativo percorso di inclusione del pre-processore, specificato tramite la clausola `#define`. **FreeRTOSConfig.h adatta il kernel RTOS all'applicazione in fase di compilazione.**

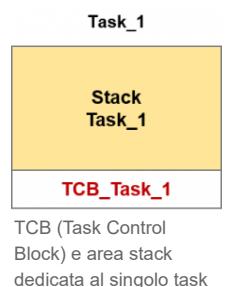
**Si noti:** essendo quindi *specifico dell'applicazione* e non dell'RTOS, la configurazione dovrebbe trovarsi in una directory dell'applicazione, non in una delle directory del codice sorgente del kernel RTOS, motivo per il quale non è stato riportato nella Figura precedente.

## Tasks

FreeRTOS è organizzato come un insieme di **task indipendenti**, ognuno dei quali **opera nel proprio contesto** senza presentare dipendenze involontarie da altri task o dallo scheduler RTOS stesso. Durante l'esecuzione dell'applicazione, *soltanto un task può essere in esecuzione* in un determinato istante, e spetta allo *scheduler* di FreeRTOS decidere quale task debba essere eseguito, tenendo conto di fattori quali **la priorità dei task** e altri parametri rilevanti. Per meglio comprendere il funzionamento di un task FreeRTOS, prendiamo in considerazione il progetto in questione. Vengono creati **quattro task indipendenti**, corrispondenti ai sensori e agli attuatori (nebulizer), oltre a un handler per il controllo del nebulizzatore. Quando un task viene interrotto per consentire l'esecuzione di un altro task, il contesto di esecuzione viene conservato nello stack del task corrente (come nella Figura accanto), garantendo un ripristino accurato del contesto quando il task viene successivamente eseguito.

Si mostra la **primitiva di creazione** di un task:

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
                        const char * const pcName,
                        configSTACK_DEPTH_TYPE usStackDepth,
                        void *pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t *pxCreatedTask
);
```



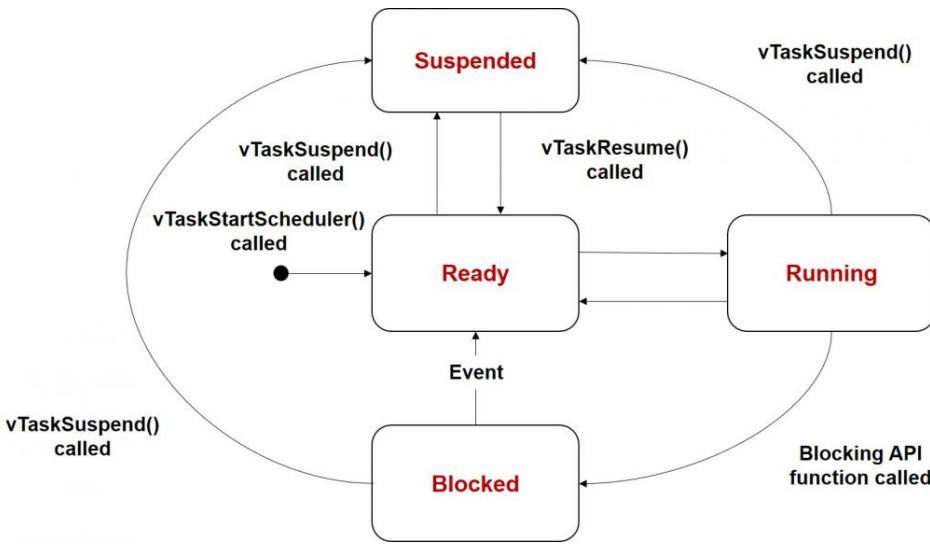
È necessario che la configurazione **configSUPPORT\_DYNAMIC\_ALLOCATION** sia impostata su 1 in *FreeRTOSConfig.h* per utilizzare questa funzione. I parametri includono il puntatore alla funzione del task (pvTaskCode), un nome descrittivo per il task, la dimensione dello stack, eventuali parametri per il task, la priorità e un puntatore per ottenere il task handle.

Ogni task, infatti, ha una **priorità assegnata** dall'utente compresa tra 0 (la priorità più bassa) e il valore **configMAX\_PRIORITIES-1** settato in fase di compilazione (la priorità più alta). Poiché un task non è a conoscenza dello Scheduler, è responsabilità di quest'ultimo quella di garantire che il contesto del processore (ovvero i valori di registro, il contenuto dello stack, e così via) venga scambiato. Questa funzione alloca automaticamente la memoria necessaria per lo stato (TCB) e lo stack del task.

### Ciclo di vita di un task

Un task può sussistere in uno dei seguenti stati:

- **Running:** rappresenta lo stato in cui il task è effettivamente in esecuzione. Ovvero, il task in questo stato sta utilizzando il processore. Se il processore su cui è in esecuzione l'RTOS ha un solo core, può esserci solo un task nello stato Running in un dato momento.
- **Ready:** i task pronti sono quelli che possono essere eseguiti (ovvero non sono nello stato Blocked o Suspended) ma che non sono attualmente in esecuzione perché un altro task con priorità uguale o superiore è già nello stato Running.
- **Blocked:** un task nello stato Blocked è attualmente in attesa di un evento temporale o esterno. Ad esempio, se un task chiama **vTaskDelay()**, verrà bloccato (passa nello stato Blocked) fino alla scadenza del periodo di delay. I task possono anche essere bloccati in attesa di un messaggio su una coda, su un semaforo, etc. I task nello stato Blocked normalmente hanno un periodo di "timeout", dopo il quale il task scadrà (passa nello stato Suspended) o verrà sbloccato (passa nello stato Ready), anche se l'evento che il task stava aspettando non si è verificato.
- **Suspended:** analogamente ai task che si trovano nello stato Blocked, i task nello stato Suspended non possono essere selezionati per passare allo stato Running, ma a differenza dello stato Blocked i task nello stato Suspended non hanno un timeout. Infatti, i task entrano o escono dallo stato Suspended solo quando richiesto esplicitamente tramite le chiamate API **vTaskSuspend()** e **xTaskResume()** rispettivamente.



Ciclo di vita dei Task in FreeRTOS

### Gestione delle priorità

La gestione delle priorità dei task viene effettuata utilizzando il concetto di "**task priority**". Ogni task viene assegnato a una specifica priorità che determina l'ordine in cui i task vengono eseguiti dallo scheduler. Le priorità dei task, come detto in precedenza, vanno da 0 a **configMAX\_PRIORITIES-1**, dove **configMAX\_PRIORITIES** è un valore definito nel file **FreeRTOSConfig.h** e rappresenta il numero massimo di priorità supportate dal sistema. Durante l'esecuzione, lo *scheduler* di FreeRTOS

assegna il tempo di CPU ai task in base alla loro priorità. Un task con una priorità più alta viene eseguito prima di un task con una priorità più bassa, consentendo di gestire l'ordine di esecuzione delle attività in modo deterministico. Se due o più task **hanno la stessa priorità**, FreeRTOS utilizza un **algoritmo di round-robin** per assegnare il tempo di CPU tra i task con la stessa priorità in modo equo. Inoltre, FreeRTOS offre la possibilità di utilizzare "task notifications" per implementare una comunicazione e un coordinamento tra task di diverse priorità. Le notifiche consentono a un task di sospendersi fino a quando non riceve una notifica da un altro task, consentendo un'interazione sincrona tra i task con priorità diverse.

Nel caso in esame, si è **scelta** questa politica di "task priority" per i seguenti task:

- **DHT11 (Sensore della temperatura) - Priorità 3:** A questo task è stato assegnato la massima priorità in quanto svolge un ruolo centrale nella misurazione della temperatura e nel confronto con i valori di riferimento per l'attivazione del nebulizzatore. La sua priorità elevata garantisce che la misurazione della temperatura venga effettuata tempestivamente e con precisione, fornendo i dati necessari per il corretto funzionamento dell'umidificatore.
- **Water Level - Priorità 3:** Questo task ha la stessa priorità rispetto al sensore di temperatura in quanto il livello dell'acqua nell'umidificatore è un aspetto altrettanto critico rispetto alla misurazione della temperatura, poiché in assenza di acqua il nebulizzatore non funzionerà.
- **Handler Nebulizer - Priorità 2:** Questo task svolge il ruolo di gestire l'abilitazione e il controllo del nebulizzatore. La sua priorità è stata scelta in modo da essere inferiore a quella dei sensori, consentendo a questi ultimi di completare le rispettive misurazioni prima che il nebulizzatore venga attivato. In questo modo, si assicura un corretto coordinamento tra i task e una gestione adeguata delle dipendenze.
- **Nebulizer - Priorità 1:** Questo task ha la priorità più bassa tra tutti i task del sistema. La sua esecuzione dipende direttamente dall'Handler Nebulizer ed esso viene attivato solo quando le misurazioni dei sensori sono state completate e il nebulizzatore verrà abilitato/disabilitato in funzione dei valori misurati da quest'ultimi. La scelta di una priorità bassa per il task Nebulizer garantisce che venga eseguito solo quando tutte le condizioni necessarie sono soddisfatte.

## Scheduling sulla base delle priorità

Lo scheduler FreeRTOS, come già detto, si occupa di gestire l'esecuzione dei task sulla CPU in base alla loro priorità. Esso ha unico scheduler che garantisce che i task nello stato di Ready o Running abbiano accesso alla CPU. Un task, ad un certo istante temporale, con una **priorità più alta prelazionerà sempre un task con una priorità più bassa**, indipendentemente dalle configurazioni. Pertanto, il task in esecuzione sarà sempre quello con la priorità più alta e pronto ad eseguire.

## System Tick

Il System Tick in FreeRTOS è un meccanismo fondamentale utilizzato per il monitoraggio del tempo e la **gestione degli eventi temporizzati nel sistema operativo in tempo reale**. Il System Tick si basa su un timer hardware (nel caso in esame TIM6 della board) o su un interrupt periodico generato da un timer hardware per generare interruzioni a intervalli regolari. Quando avviene un'interruzione del System Tick, lo scheduler dell'RTOS viene invocato per eseguire un'operazione di scheduling. Durante questa operazione, il task in esecuzione viene sospeso e viene selezionato un nuovo task da eseguire sulla base delle politiche di scheduling configurate, come la priorità dei task. Il System Tick può essere **configurato** per generare interruzioni a intervalli di tempo definiti, comunemente chiamati "**tick rate**". Questo intervallo di tempo è impostato nel file di configurazione di FreeRTOS (FreeRTOSConfig.h) tramite il parametro **configTICK\_RATE\_HZ**. Le **interruzioni del System Tick** sono essenziali per consentire a FreeRTOS di gestire le operazioni temporizzate, come il passaggio dei task da uno stato all'altro, l'esecuzione di task periodici o l'attesa di eventi che devono verificarsi entro un determinato tempo. Il System Tick di FreeRTOS è cruciale per mantenere l'accuratezza e la precisione delle temporizzazioni nel sistema, consentendo un'efficace gestione delle priorità dei task e la sincronizzazione degli eventi temporizzati.

## Gestione della tempificazione e requisiti real-time

Nel caso in esame vengono **usati 3 timer** per la gestione della tempificazione, in particolare, come già anticipato si usa il **TIM6 come SysTick di FreeRTOS**. Inoltre, si usa il **TIM1 per scandire il periodo di esecuzione del task che si occupa di gestire il sensore di temperatura**, infatti, attraverso un semaforo binario si gestisce la sezione critica relativa al DHT11 facendo in modo che una volta acquisito il semaforo, esso venga rilasciato solo dopo un certo periodo di tempo all'interno della callback del TIM1 opportunamente configurato. In questo modo il task di gestione del sensore di temperatura anche se viene schedulato perché è il task a priorità maggiore non potrà eseguire perché il semaforo non verrà rilasciato fintantoché non sarà trascorso il periodo di tempo impostato nella configurazione del TIM1, come vedremo successivamente. Infine, si usa il **TIM7 per gestire il delay necessario al DHT11 durante la sua fase di startup**, in particolare il TIM7 viene configurato in maniera tale da permettere di introdurre dei ritardi nell'ordine dei microsecondi.

**Si noti:** Non essendo l'applicazione di tipo safety critical non sono stati previsti hard real-time stringenti per l'esecuzione, ma si è valutata in maniera del tutto qualitativa la responsività dell'applicazione in termini di latenza percepita dall'utente come il tempo intercorso tra un cambiamento dei valori rilevati dai sensori e l'accensione del nebulizzatore, **senza notare ritardi evidenti**.

## Configurazione dei timer

Il TIM1 viene configurato nel seguente modo:

- **Modalità:** UP Counting;
- **Prescaler:** 48000-1;
- **Counter Period:** 2000-1.

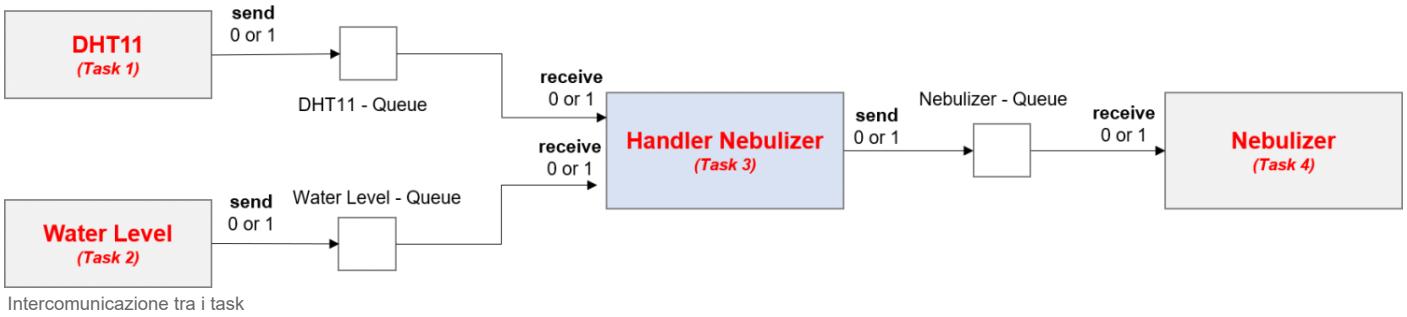
Il TIM7 viene configurato nel seguente modo:

- **Modalità:** UP Counting;
- **Prescaler:** 48-1;
- **Counter Period:** 65535-1.

Il TIM6 viene impostato come Timebase Source del SysTick.

## Intercomunicazione tra i task

Si mostra un ipotetico schema generale di intercomunicazione tra i task dell'applicazione:



#### Gestione delle code

La gestione delle code avviene attraverso l'utilizzo di oggetti di tipo "**queue**" (coda di messaggi). Essa rappresenta un'area di memoria in cui i task possono inserire dati in modo asincrono e prelevarli successivamente. Le code sono *utilizzate per la comunicazione e la sincronizzazione tra i task*.

Per **creare una coda** in FreeRTOS, viene utilizzata la funzione **xQueueCreate()** che restituisce un handle alla coda appena creata. È possibile specificare la dimensione massima della coda e la dimensione dei messaggi che può contenere.

Si mostra di seguito la primitiva xQueueCreate() utilizzata:

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

Per **inviare un messaggio nella coda**, viene utilizzata la funzione **xQueueSend()**. Questa funzione prende come parametri l'handle della coda, un puntatore al messaggio da inviare e un timeout opzionale. Il **timeout specifica per quanto tempo il task rimarrà in attesa se la coda è piena**, per un comportamento bloccante si setta tale parametro pari a portMAX\_DELAY.

Si mostra di seguito la primitiva xQueueSend() utilizzata:

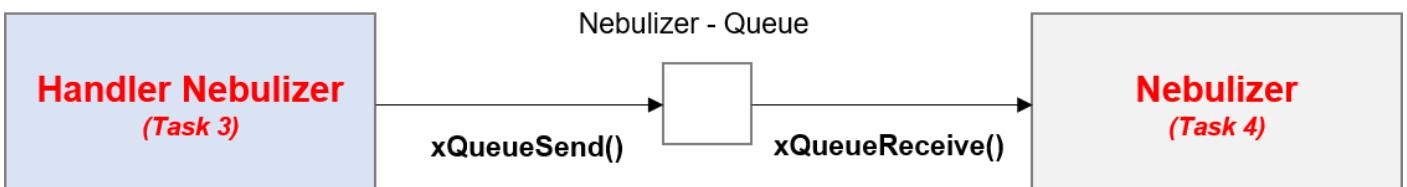
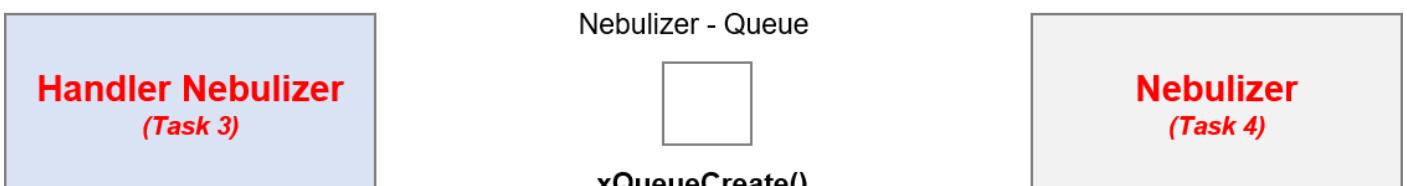
```
 BaseType_t xQueueSend(QueueHandle_t xQueue,
                        const void * pvItemToQueue,
                        TickType_t xTicksToWait
);
```

Per ricevere un messaggio dalla coda, viene utilizzata la funzione **xQueueReceive()**. Questa funzione prende come parametri l'handle della coda, un puntatore in cui memorizzare il messaggio ricevuto e un timeout opzionale. Il timeout specifica per quanto tempo il task rimarrà in attesa se la coda è vuota.

Si mostra di seguito la primitiva xQueueReceive() utilizzata

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

Si mostra un esempio schematico per la create, send e receive di un dato tra i task.



Esempio schematico per la create, send e receive di un dato tra i task

#### Configurazione delle code

Si mostra la configurazione delle code nel caso in esame. A valle della istanziazione delle tre strutture code:

```
xQueueHandle xQueue1;
xQueueHandle xQueue2;
```

```
xQueueHandle xQueue3;
```

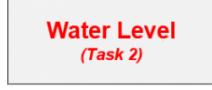
Si ricorda che l'handle consente di accedere e manipolare la coda utilizzando le funzioni API appropriate fornite da FreeRTOS elencate in precedenza.

Si creano pertanto le rispettive queue:

```
xQueue1 = xQueueCreate(mainQUEUE_LENGTH, sizeof(uint8_t));
```



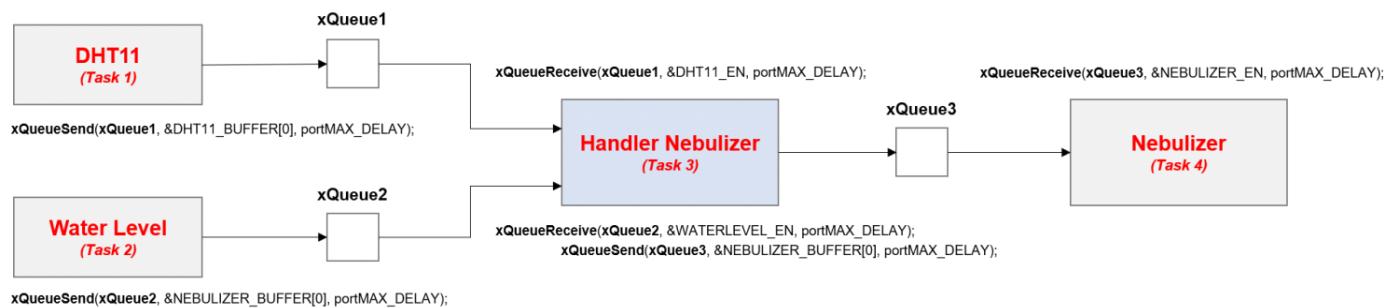
```
xQueue3 = xQueueCreate(mainQUEUE_LENGTH, sizeof(uint8_t));
```



```
xQueue2 = xQueueCreate(mainQUEUE_LENGTH, sizeof(uint8_t));
```

Creazione delle code con xQueue

#### Configurazione complessiva:



Configurazione complessiva delle code

## Sincronizzazione tra i task: semafori binari e mutex

FreeRTOS mette a disposizione i seguenti tipi di sincronizzazione: **semafori, semafori binari e mutex**.

I semafori binari vengono utilizzati sia per la mutua esclusione che per la sincronizzazione. I semafori binari e i mutex sono molto simili ma presentano alcune sottili differenze, che li portano ad essere usati in casi ortogonali tra di loro. Infatti, una prima differenza sta nel fatto che i mutex includono un meccanismo di priority inheritance (per mitigare il problema dell'inversione della priorità), mentre i semafori binari no. Inoltre, un mutex implica che lo stesso task "prenda" e "rilasci" il mutex, il che significa che il task "possiede" il mutex durante l'esecuzione della sezione critica. Un semaforo viene dato e preso da task diversi, il che significa che è possibile utilizzarlo come meccanismo di segnalazione per sincronizzare i task. Ciò rende i **semafori binari la scelta migliore per implementare la sincronizzazione** (tra task diversi o tra task e interrupt) e i mutex la scelta migliore per implementare la semplice mutua esclusione. Queste valutazioni hanno, dunque, condotto alla scelta dell'utilizzo di un semaforo binario nel caso in esame, in quanto non si desidera che un'ISR (Interrupt Service Routine) blocchi l'accesso alla CPU, perché utilizza un mutex al suo interno. Tuttavia, è possibile utilizzare un semaforo in un'ISR per segnalare ad altri task che è stato eseguito e che alcuni dati sono pronti per essere utilizzati.

#### Gestione del semaforo binario

Le funzioni API del semaforo binario consentono di specificare un tempo di blocco. Il tempo di blocco indica il numero massimo di "tick" in cui un'attività deve entrare nello stato Blocked quando tenta di "prendere" quel semaforo, se il semaforo non è immediatamente disponibile. Questo meccanismo, all'interno del seguente progetto è stato sfruttato per sincronizzare il task di gestione del sensore di temperatura con l'interrupt proveniente dal TIM1.

Infatti, per gestire il DHT11 il task dovrebbe effettuare un'attività di polling sulla periferica, andando a sprecare le risorse della CPU e impedendo l'esecuzione di altri task. È quindi preferibile che il task trascorra la maggior parte del suo tempo nello stato Blocked (**consentendo l'esecuzione di altri task**) esegua solo quando c'è effettivamente qualcosa da fare. Ciò si ottiene utilizzando un semaforo binario che blocca il task quando tenta di "prendere" il semaforo. Viene quindi scritta una **callback per l'interrupt del TIM1** che "rilascia" il semaforo solo quando è trascorso il periodo di tempo necessario al DHT11 (protocollo di inizializzazione del DHT11). In pratica, il task "acquisisce" sempre il semaforo binario, ma non lo "rilascia" mai. **La callback "rilascia" sempre il semaforo binario ma non lo prende mai.**

**Si noti:** all'interno della callback che "rilascia" il semaforo binario occorre usare la funzione xSemaphoreGiveFromISR(), e non la sua versione classica, in quanto le ISR sono eseguite a livello di interrupt, in un contesto diverso rispetto al contesto dei task del RTOS. In particolare, le funzioni che terminano con "fromISR()" sono state progettate per essere utilizzate all'interno di una ISR e presentano alcune caratteristiche specifiche:

- Accedono alle risorse in modo sicuro per evitare condizioni di race condition o accessi concorrenti non sicuri;
- Effettuano operazioni che sono permesse all'interno di una ISR, ad esempio, non è possibile bloccare o sospendere l'esecuzione dell'ISR per un tempo prolungato, poiché ciò potrebbe influire sulla gestione degli interrupt;
- Mantengono la priorità dell'ISR corrente, garantendo che il RTOS possa riprendere l'esecuzione nel modo corretto e senza interferenze dopo l'esecuzione dell'ISR.

Questo meccanismo fornito dal semaforo binario lavora correttamente, solo se la definizione delle priorità dei task è effettuata in maniera coerente con le periferiche che gestiscono, generando in modo efficace uno schema di "interruzione differita" (si noti che FreeRTOS ha anche un meccanismo di interrupt differito incorporato).

#### Configurazione del semaforo binario

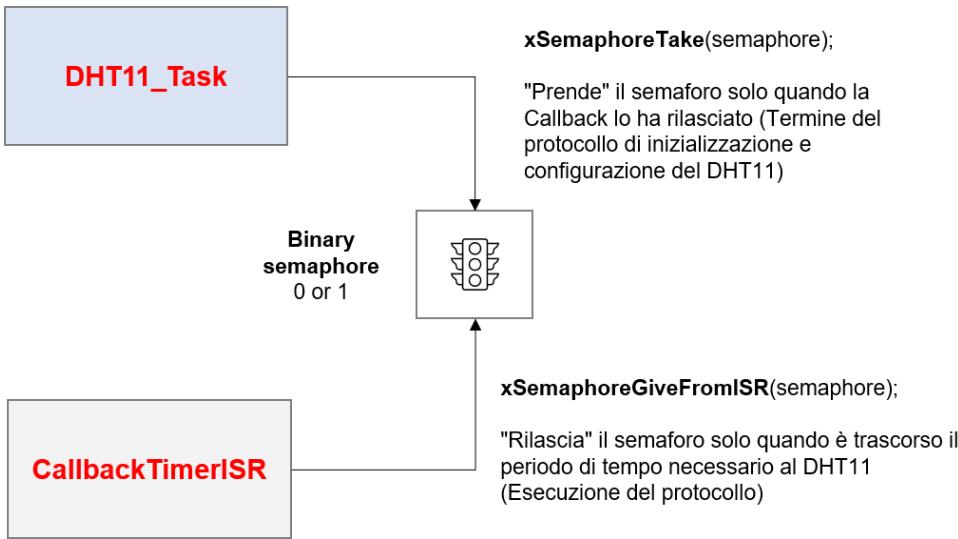
Per utilizzare le API relative ai semafori è innanzitutto necessario includere l'header file opportuno presente nella directory *FreeRTOS/Source/include/semphr.h*. La creazione del semaforo binario avviene invocando la seguente funzione che non ha nessun parametro in input:

```
SemaphoreHandle_t xSemaphoreCreateBinary( void );
```

Mentre le API utilizzate per implementare le funzioni di sincronizzazione sono:

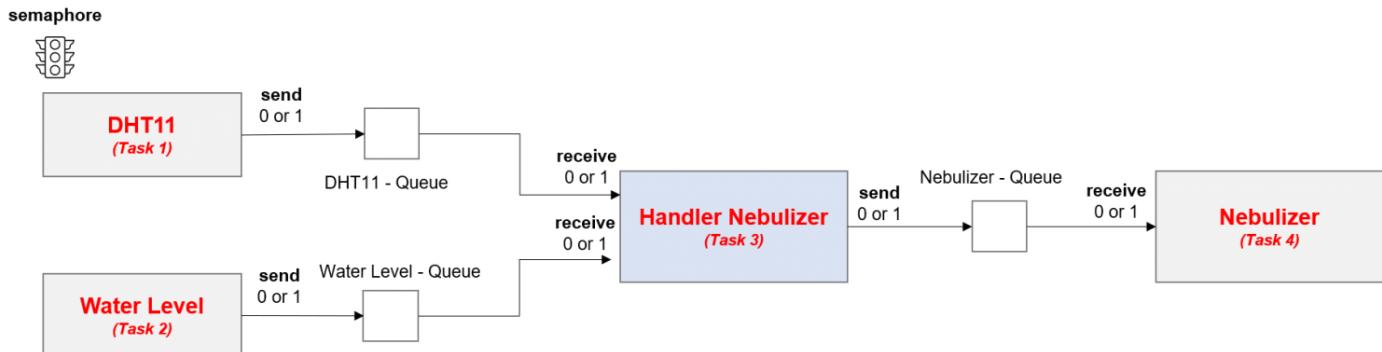
```
xSemaphoreTake(SemaphoreHandle_t xSemaphore,
    TickType_t xTicksToWait
);
xSemaphoreGiveFromISR(SemaphoreHandle_t xSemaphore,
    signed BaseType_t *pxHigherPriorityTaskWoken
);
```

Il **funzionamento** del semaforo implementato:



Il funzionamento del semaforo implementato

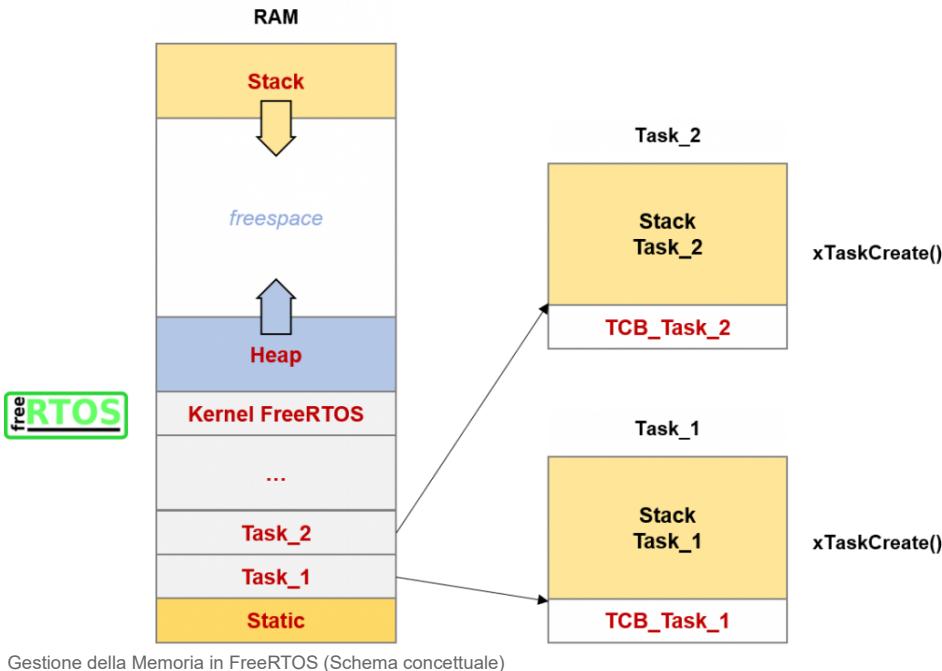
Si schematizza dunque la schedulazione completa concettuale del sistema con semafori e code:



Schedulazione completa concettuale - semafori e code

## Gestione della memoria

Il kernel di un sistema operativo in tempo reale (RTOS) richiede l'accesso alla memoria RAM ogni volta che vengono creati elementi come task, code, mutex, timer software, semafori o gruppi di eventi. La RAM può essere assegnata dinamicamente tramite l'heap del RTOS o fornita direttamente dall'applicazione stessa. Si mostra l'architettura della memoria a valle della creazione dei task:



Gestione della Memoria in FreeRTOS (Schema concettuale)

Tuttavia, l'impiego delle funzioni standard `malloc()` e `free()` della libreria C per l'allocazione dinamica della memoria può risultare impraticabile o inappropriato in ambienti embedded. Oltre a occupare spazio di codice, queste funzioni non garantiscono la sicurezza in contesti multithreading e non offrono tempi di esecuzione deterministici. Per superare queste limitazioni, in generale, FreeRTOS mette a disposizione un'API di allocazione della memoria nel suo strato portabile. Quando il kernel del RTOS necessita di RAM, invece di utilizzare `malloc()`, viene invocata la funzione `pvPortMalloc()`. Analogamente, per liberare la RAM, viene chiamata la funzione `vPortFree()` al posto di `free()`.

FreeRTOS offre **diverse implementazioni** per la gestione dell'heap, ognuna con livelli di complessità e funzionalità differenziate. In particolare, abbiamo le seguenti implementazioni:

- **heap\_1** - Consente solo l'allocazione di memoria e viene utilizzata quando le applicazioni non rilasciano mai risorse, come task, semafori o code, e quindi non necessitano di liberare la memoria.
- **heap\_2** - Permette il rilascio della memoria, ma non gestisce la frammentazione unendo blocchi di memoria liberi adiacenti. Questa implementazione viene preferita quando le risorse vengono frequentemente rimosse dall'applicazione. Tuttavia, va evitata quando la quantità di memoria stack allocata ai task varia.
- **heap\_3** - Si limita ad incapsulare le funzioni standard `malloc()` e `free()` per garantire la sicurezza in contesti multithreading.
- **heap\_4** - Simile a heap\_2, ma gestisce anche la frammentazione unendo blocchi di memoria liberi adiacenti. Inoltre, offre l'opzione di posizionare gli indirizzi di memoria in modo assoluto.
- **heap\_5** - Come heap\_4, ma consente di allocare l'heap su diverse aree di memoria non contigue.

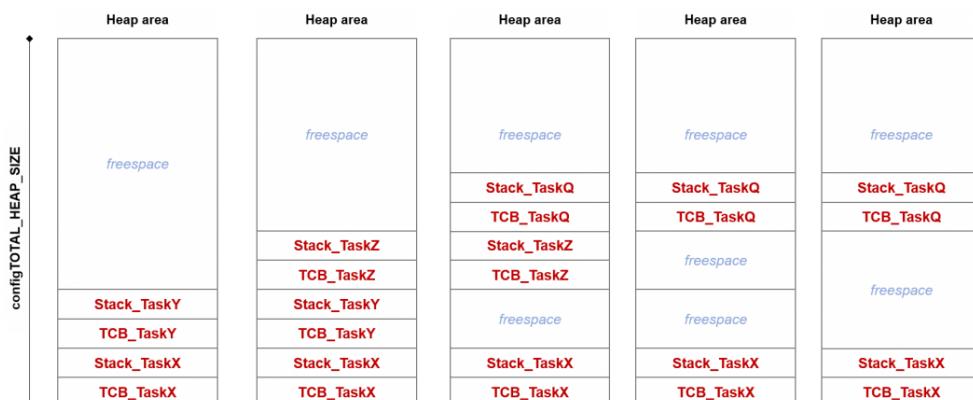
Nel contesto in questione, è stato adottato l'utilizzo del "memory management" fornito da heap\_4. Tale scelta è stata effettuata nonostante non sia strettamente richiesta dall'applicazione, dato che i task vengono allocati ma non eliminati, potendo quindi considerare l'utilizzo di heap\_1. Tuttavia, è stata presa la decisione di utilizzare heap\_4 poiché ha consentito di ridurre l'impronta (footprint) dell'applicazione rispetto ad altre implementazioni. Ciò è stato possibile grazie al suo meccanismo ottimizzato per l'allocazione della memoria dinamica per i task, le code e i semafori binari.

#### Allocazione dinamica dello stack: heap\_4.c

Questa implementazione consente di evitare la frammentazione che potrebbe verificarsi quando vengono allocate e deallocate ripetutamente porzioni di memoria.

Quando viene richiesta una nuova allocazione di memoria, heap\_4 cerca tra i blocchi di memoria liberi disponibili per trovare uno spazio sufficientemente grande da assegnare alla richiesta. Se il blocco libero individuato è più grande del necessario, viene suddiviso in due parti: una parte viene assegnata alla richiesta e la restante viene mantenuta come blocco libero.

Quando viene deallocata una porzione di memoria, heap\_4 controlla se i blocchi liberi adiacenti possono essere fusi insieme per formare un blocco più grande. In tal caso, i blocchi liberi vengono uniti per formare un unico blocco di memoria libero. Questa operazione riduce la frammentazione della memoria, consentendo di utilizzare in modo più efficiente gli spazi disponibili.



Il funzionamento dell'Heap 4

Abilitazione del memory management fornito da **heap\_4** nel makefile, utilizzato per compilare i file sorgente dell'applicazione, è fatta con il seguente codice:

```
$(FREERTOS_ROOT_DIR)/portable/MemMang/heap_4.c
```

## Applicazione e debug

A questo punto si passa alla fase di implementazione e debug del codice relativo all'applicazione di gestione del nebulizzatore smart su STM32F303VC realizzato sfruttando le funzioni e potenzialità messe a disposizione dal kernel FreeRTOS nella **versione 10.0.1**.

### Utilizzo delle API Hardware Abstraction Layer

Si utilizzano le librerie **HAL (Hardware Abstraction Layer)** fornite dai produttori della ST o in generale di microcontrollori per semplificare l'interfacciamento con l'hardware specifico e di basso livello di un determinato microcontrollore. Queste librerie astraggono i dettagli specifici dell'hardware e forniscono un'interfaccia coerente e portabile per accedere alle periferiche e alle funzionalità del microcontrollore. Forniscono pertanto un'interfaccia comoda per configurare e utilizzare le periferiche hardware del microcontrollore, come i timer, gli interrupt, le porte I/O, etc (Nel caso in esame HAL GPIO, HAL ADC, etc.). In questo progetto si è scelto di utilizzare tali librerie per rendere il codice il più possibile portabile e riusabile su altre piattaforme che supportano le medesime tecnologie.

## Implementazione: Air Humidifier

Si mostra l'applicazione in funzionamento dell'umidificatore:

### Codice

Si mostra in breve porzioni di codice dei relativi task e main dell'applicazione:

#### Main

Nel **main.c** vengono inizializzate le periferiche necessarie all'applicazione tramite le opportune funzioni fornite dal produttore del microcontrollore (ST). Vengono poi creati nell'ordine, il semaforo binario DHT\_SEM, le 3 code per lo scambio dei messaggi di abilitazione del nebulizzatore ed infine, i 4 task che implementano le funzioni di monitoraggio dei sensori e gestione del nebulizzatore stesso.

```
int main(void)
{
    /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
    HAL_Init();

    /* Configure the system clock */
    SystemClock_Config();

    /* Initialize all configured peripherals */
    MX_GPIO_Init();
    MX_ADC1_Init();
    MX_TIM1_Init();
    MX_TIM7_Init();

    DHT_SEM = xSemaphoreCreateBinary();

    xQueue1 = xQueueCreate(mainQUEUE_LENGTH, sizeof(uint8_t));
    xQueue2 = xQueueCreate(mainQUEUE_LENGTH, sizeof(uint8_t));
    xQueue3 = xQueueCreate(mainQUEUE_LENGTH, sizeof(uint8_t));

    xTaskCreate(DHT11_Task, "DHT11", 128, NULL, 4, &DHT11_Handler);
    xTaskCreate(WaterLevel_Task, "WaterLevel", 128, NULL, 3, &WaterLevel_Handler);
    xTaskCreate(HandNeb_Task, "HandNeb", 128, NULL, 2, &HandNeb_Handler);
    xTaskCreate(Nebulizer_Task, "Nebulizer", 128, NULL, 1, &Nebulizer_Handler);

    HAL_TIM_Base_Start(&htim7); // us delay timer
    HAL_TIM_Base_Start_IT(&htim1); // periodic delay timer

    vTaskStartScheduler();

    while (1)
    {
    }
}
```

#### DHT11 Task

Il task del DHT11 si occupa della **lettura dei valori di Temperatura e Umidità dal sensore**, sulla base di quest'ultimi il task invia il bit opportuno al task dell'Handler tramite la xQueue1. Questa funzione viene regolata tramite l'accesso al semaforo binario creato nel main ed in particolare, il task eseguirà solo quando il TIM1 all'interno della propria callback rilascia il semaforo.

```
void DHT11_Task (void *argument)
{
    while (1)
    {
        if (xSemaphoreTake(DHT_SEM, 2500) == pdTRUE)
        {
            DHT11_Get_Data(&Temperature, &Humidity);

            if (Temperature > 23 && (Humidity < 60))
            {
                /* ACCENDI UMIIDIFICATORE
                 * Temperature > 23° --> Humidity 30%-60%
                 */
                HAL_GPIO_WritePin(GPIOE, GPIO_PIN_8, GPIO_PIN_RESET); //BLU
                HAL_GPIO_WritePin(GPIOE, GPIO_PIN_9, GPIO_PIN_SET); //ROSSO
                HAL_GPIO_WritePin(GPIOE, GPIO_PIN_10, GPIO_PIN_RESET); //ARANCIONE
            }
        }
    }
}
```

```

        DHT11_BUFFER[0] = 1UL;
    }
    else if ((Temperature <= 23 && Temperature >= 18) && (Humidity < 55))
    {
        /* ACCENDI UMIDIFICATORE
         * Temperature 18°-23° --> Humidity 30%-55%
         */
        HAL_GPIO_WritePin(GPIOE, GPIO_PIN_8, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOE, GPIO_PIN_9, GPIO_PIN_RESET);
        HAL_GPIO_WritePin(GPIOE, GPIO_PIN_10, GPIO_PIN_RESET);
        DHT11_BUFFER[0] = 1UL;
    }
    else if (Temperature < 18 && (Humidity < 50))
    {
        /* ACCENDI UMIDIFICATORE
         * Temperature < 18° --> Humidity 30%-50%
         */
        HAL_GPIO_WritePin(GPIOE, GPIO_PIN_8, GPIO_PIN_RESET);
        HAL_GPIO_WritePin(GPIOE, GPIO_PIN_9, GPIO_PIN_RESET);
        HAL_GPIO_WritePin(GPIOE, GPIO_PIN_10, GPIO_PIN_SET);
        DHT11_BUFFER[0] = 1UL;
    }
    else
    {
        HAL_GPIO_WritePin(GPIOE, GPIO_PIN_8, GPIO_PIN_RESET);
        HAL_GPIO_WritePin(GPIOE, GPIO_PIN_9, GPIO_PIN_RESET);
        HAL_GPIO_WritePin(GPIOE, GPIO_PIN_10, GPIO_PIN_SET);
        DHT11_BUFFER[0] = 0UL;
    }
}
xQueueSend(xQueue1,&DHT11_BUFFER[0],portMAX_DELAY);
}
}
}

```

### Water Level Task

Il task si occupa di **leggere il valore del livello d'acqua all'interno della vasca di pescaggio del nebulizzatore**, in particolare, il sensore necessita di una conversione Analogico-Digitale per leggere tale livello. In base al fatto, che l'acqua sia presente o meno si invia l'opportuno bit al task dell'Handler tramite la xQueue2.

```

void WaterLevel_Task (void *argument)
{
    while (1)
    {
        HAL_ADC_Start(&hadc1);
        HAL_ADC_PollForConversion(&hadc1, 10);
        ADC_VAL = HAL_ADC_GetValue(&hadc1);
        HAL_ADC_Stop(&hadc1);

        if (ADC_VAL >= 200UL)
        {
            WATERLEVEL_BUFFER[0] = 1UL;
        } else {
            WATERLEVEL_BUFFER[0] = 0UL;
        }

        xQueueSend(xQueue2,&WATERLEVEL_BUFFER[0],portMAX_DELAY);

        vTaskDelay(500);
    }
}

```

### Handler Nebulizer Task

L'Handler riceve i due bit dai task rispettivamente del DHT11 e del Livello d'Acqua, **se entrambi i bit sono alti allora bisogna accendere il nebulizzatore** inviando sulla xQueue3 il bit di NEBULIZER\_EN pari a 1, altrimenti il nebulizzatore viene spento inviando il bit pari a 0 (l'Handler si comporta come una AND logica).

```

void HandNeb_Task (void *argument)
{
    while(1)
    {

        xQueueReceive(xQueue1,&DHT11_EN,portMAX_DELAY);
        xQueueReceive(xQueue2,&WATERLEVEL_EN,portMAX_DELAY);

        if (DHT11_EN == 1UL && WATERLEVEL_EN == 1UL)
        {
            NEBULIZER_BUFFER[0] = 1UL;
        } else {
            NEBULIZER_BUFFER[0] = 0UL;
        }

        xQueueSend(xQueue3,&NEBULIZER_BUFFER[0],portMAX_DELAY);
        vTaskDelay(250);
    }
}

```

### Nebulizer Task

Questo task è il responsabile dell'accensione vera e propria del nebulizzatore. In particolare, sulla base del valore ricevuto sulla xQueue3 dall'Handler va ad **abilitare/disabilitare il nebulizzatore** ponendo il pin PB6 (impostato come Output pin) alto/basso.

```

void Nebulizer_Task (void *argument)
{
    while(1)
    {

        xQueueReceive(xQueue3,&NEBULIZER_EN,portMAX_DELAY);
        if (NEBULIZER_EN == 1UL)
        {
            Nebulizer_Actived = 1UL;
            HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_SET);
        }
    }
}

```

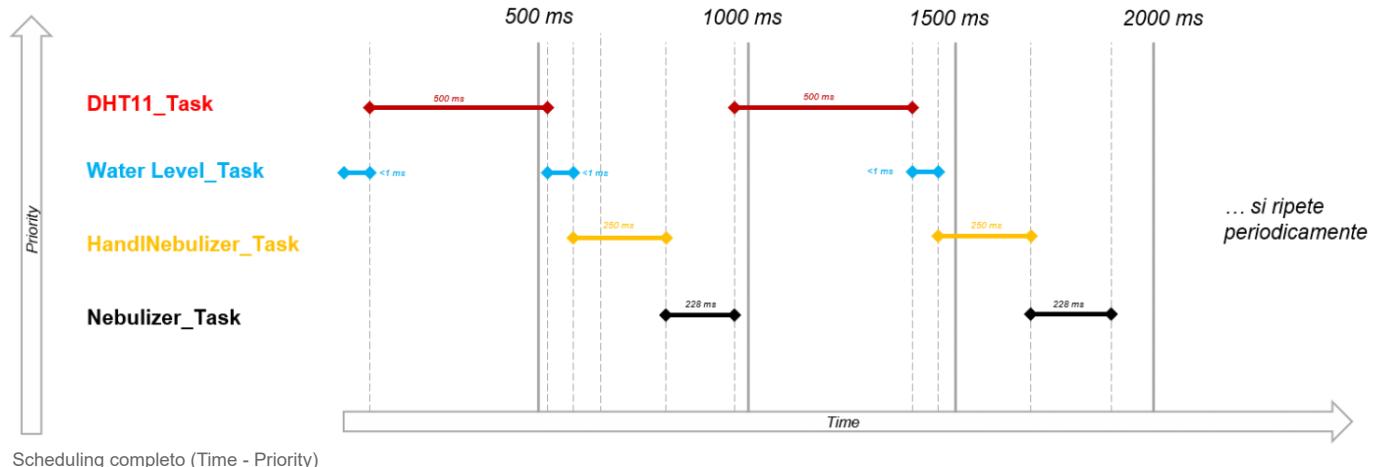
```

        } else {
            Nebulizer_Actived = 0UL;
            HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_RESET);
        }

        vTaskDelay(250);
    }
}

```

Si mostra lo scheduling ipotetico dei task sulla base del codice completo presente al seguente link: [Codice Completo \(https://github.com/giuseppericcio/ES\\_AirHumidifier\\_FreeRTOS\)](https://github.com/giuseppericcio/ES_AirHumidifier_FreeRTOS)



## Makefile per la compilazione completa

Per **compilare i file sorgente viene utilizzato un apposito MAKEFILE**, che tramite il suo linguaggio di scripting permette di automatizzare l'intera fase di compilazione e linking. In particolare, il Makefile scritto esegue queste operazioni in 2 step, il primo invocato tramite il target “*prepare*” si occupa del retrieve di tutte le dipendenze necessarie per la creazione degli eseguibili tra cui le librerie dell’HAL ed i file del kernel FreeRTOS. Il secondo step riguarda la fase di *compilazione e linking* vera e propria, effettuata tramite il compilatore GCC fornito dalla toolchain ARM, la quale permette di creare l'eseguibile dell'applicazione in formato .elf.

```

# How to use this makefile:
#
# > make prepare # copy all source files to a working directory (src)
# > make          # build the default target (freertosproject.bin)
# > make debug   # run the GDB debugger
# > make clean    # remove all temporary files in the working directories (src, obj, dep)

# +++ CHECK ALL THE SETTINGS BELOW AND ADAPT THEM IF NEEDED +++

# default target and name of the image and executable files to generate
TARGET      = freertosproject

# path to the root folder of the STM32Cube platform
STM_DIR     = /c/Users/<USERNAME>/Documents/EmbeddedSystems/STM32CubeF3
# Board and MCU names as used in the linker script path and file name, e.g. "$(STM_DIR)/Demonstrations/SW4STM32/STM32F3-DISCO/STM32F303VGTX_FLASH.ld"
BOARD_UC    = STM32F3-DISCO
MCU_UC      = STM32F303VC

# board name as used in the STM32cube Drivers/BSP folder, e.g. "$(STM_DIR)/Drivers/BSP/STM32F3-Discovery"
#BSP_BOARD   = STM32F3-Discovery

# MCU name as used in the .s source file name, e.g. "startup_stm32f303xx.s"
MCU_LC      = stm32f303vctx

# pre-processor symbol to be defined for the compilation (will be used in a -Dxxx flag in gcc)
MCU_MC      = STM32F303xC

FREERTOS_PORT ?= ARM_CM4F

#####
# Directories

HAL_DIR     = $(STM_DIR)/Drivers/STM32F3xx_HAL_Driver
CMSIS_DIR   = $(STM_DIR)/Drivers/CMSIS
DEV_DIR     = $(CMSIS_DIR)/Device/ST/STM32F3xx
FREERTOS_ROOT_DIR := ./FreeRTOS/Source
FREERTOS_PORT_ROOT += \
$(FREERTOS_ROOT_DIR)/portable/GCC/$(FREERTOS_PORT)

#####
# Source files

FREERTOS_KERNEL_SOURCES += \
$(FREERTOS_ROOT_DIR)/tasks.c \
$(FREERTOS_ROOT_DIR)/queue.c \
$(FREERTOS_ROOT_DIR)/list.c \
$(FREERTOS_ROOT_DIR)/timers.c \
$(FREERTOS_ROOT_DIR)/event_groups.c \
$(FREERTOS_ROOT_DIR)/stream_buffer.c \
$(FREERTOS_ROOT_DIR)/croutine.c \
$(FREERTOS_PORT_ROOT)/port.c \
$(FREERTOS_ROOT_DIR)/portable/MemMang/heap_4.c

SRCS = \
$(HAL_DIR)/Src/stm32f3xx_hal.c \

```

```

$(HAL_DIR)/Src/stm32f3xx_hal_cortex.c \
$(HAL_DIR)/Src/stm32f3xx_hal_gpio.c \
$(HAL_DIR)/Src/stm32f3xx_hal_dma.c \
$(HAL_DIR)/Src/stm32f3xx_hal_rcc.c \
$(HAL_DIR)/Src/stm32f3xx_hal_rcc_ex.c \
$(HAL_DIR)/Src/stm32f3xx_hal_flash.c \
$(HAL_DIR)/Src/stm32f3xx_hal_flash_ex.c \
$(HAL_DIR)/Src/stm32f3xx_hal_pcd.c \
$(HAL_DIR)/Src/stm32f3xx_hal_adc.c \
$(HAL_DIR)/Src/stm32f3xx_hal_adc_ex.c \
$(HAL_DIR)/Src/stm32f3xx_hal_exti.c \
$(HAL_DIR)/Src/stm32f3xx_hal_tim.c \
$(HAL_DIR)/Src/stm32f3xx_hal_tim_ex.c \
$(HAL_DIR)/Src/stm32f3xx_hal_pwr.c \
$(HAL_DIR)/Src/stm32f3xx_hal_pwr_ex.c \
Sensors/main.c \
Sensors/DHT11.c \
Sensors/freertos.c \
Sensors/stm32f3xx_hal_msp.c \
Sensors/stm32f3xx_hal_timebase_tim.c \
Sensors/system_stm32f3xx.c \
Sensors/stm32f3xx_it.c \
Sensors/sysmem.c \
$(FREERTOS_KERNEL_SOURCES)

# remove paths from the file names
SRCS_FN = $(notdir $SRCS)

LDFILE = Sensors/$(MCU_UC)Tx_FLASH.1d

#####
# Tools

PREFIX      = arm-none-eabi
CC          = $(PREFIX)-gcc
AR          = $(PREFIX)-ar
OBJCOPY     = $(PREFIX)-objcopy
OBJDUMP    = $(PREFIX)-objdump
SIZE        = $(PREFIX)-size
GDB         = $(PREFIX)-gdb

#####
# Options

# Defines (-D flags)
DEFS      = -D$(MCU_MC) -DUSE_HAL_DRIVER
DEFS      += -DUSE_DBPRINTF

# Include search paths (-I flags)
INCS      = -Isrc
INCS      += -I$(STM_DIR)/Drivers/CMSIS/Include
INCS      += -I$(STM_DIR)/Drivers/CMSIS/Device/ST/STM32F3xx/Include
INCS      += -I$(STM_DIR)/Drivers/STM32F3xx_HAL_Driver/Inc
INCS      += -I$(FREERTOS_ROOT_DIR)/include
INCS      += -I$(FREERTOS_PORT_ROOT)
#INCS      += -I$(STM_DIR)/Drivers/BSP/STM32F3-Discovery
#INCS      += -I$(STM_DIR)/Drivers/BSP/Components/lis302d1
#INCS      += -I$(STM_DIR)/Drivers/BSP/Components/lis3dsh

# Library search paths (-L flags)
LIBS      = -L$(CMSIS_DIR)/Lib

# Compiler flags
CFLAGS     = -Wall -g -std=c99 -Os
CFLAGS     += -mlittle-endian -mcpu=cortex-m4 -march=armv7e-m -mthumb
CFLAGS     += -mfpu=fpv4-sp-d16 -mfloat-abl-hard
CFLAGS     += -ffunction-sections -fdata-sections -ftime-report
CFLAGS     += $(INCS) $(DEFS)

# Linker flags
LDFLAGS    = -Wl,--print-memory-usage -Wl,--gc-sections
-LWl,-Map=$(TARGET).map $(LIBS) -Tsrc/linkerScript.ld

# Enable Semihosting
LDFLAGS    += --specs=rdimon.specs -lc -lrdimon
#LDFLAGS    += --specs=nosys.specs --specs=nano.specs --specs=rdimon.specs -lc -lrdimon

# Source search paths
VPATH      = ./src
#VPATH      += $(BSP_DIR)
VPATH      += $(HAL_DIR)/Src
VPATH      += $(DEV_DIR)/Source

# Debugger flags
GDBFLAGS   =

# generate OBJS and DEPS target lists by prepending obj/ and dep prefixes
OBJS      = $(addprefix obj/, $(SRCS_FN:.c=.o))
DEPS      = $(addprefix dep/, $(SRCS_FN:.c=.d))

#####
.PHONY: all dirs debug prepare clean

all: $(TARGET).bin

-include $(DEPS)

dirs: dep obj

dep obj src:
    @echo "[MKDIR]  $@"
    mkdir -p $@

obj/%.o : %.c | dirs
    @echo "generating \"$@\" from '$<''"
    $(CC) $(CFLAGS) -c -o $@ $< -MMD -MP -MF dep/$(*F).d

$(FREERTOS_PORT_ROOT):
    git clone git@github.com:FreeRTOS/FreeRTOS-Kernel.git
    $(FREERTOS_ROOT_DIR)

$(TARGET).elf: $(OBJS)
```

```

@echo "[LD]      $(TARGET).elf"
$(CC) $(CFLAGS) $(LDFLAGS) src/startup_$(MCU_lc).s $^ -o $@
@echo "[OBJDUMP] $(TARGET).lst"
$(OBJDUMP) -St $(TARGET).elf >$(TARGET).lst
@echo "[SIZE]   $(TARGET).elf"
$(SIZE) $(TARGET).elf

$(TARGET).bin: $(TARGET).elf
    @echo "[OBJCOPY] $(TARGET).bin"
    $(OBJCOPY) -O binary $< $@

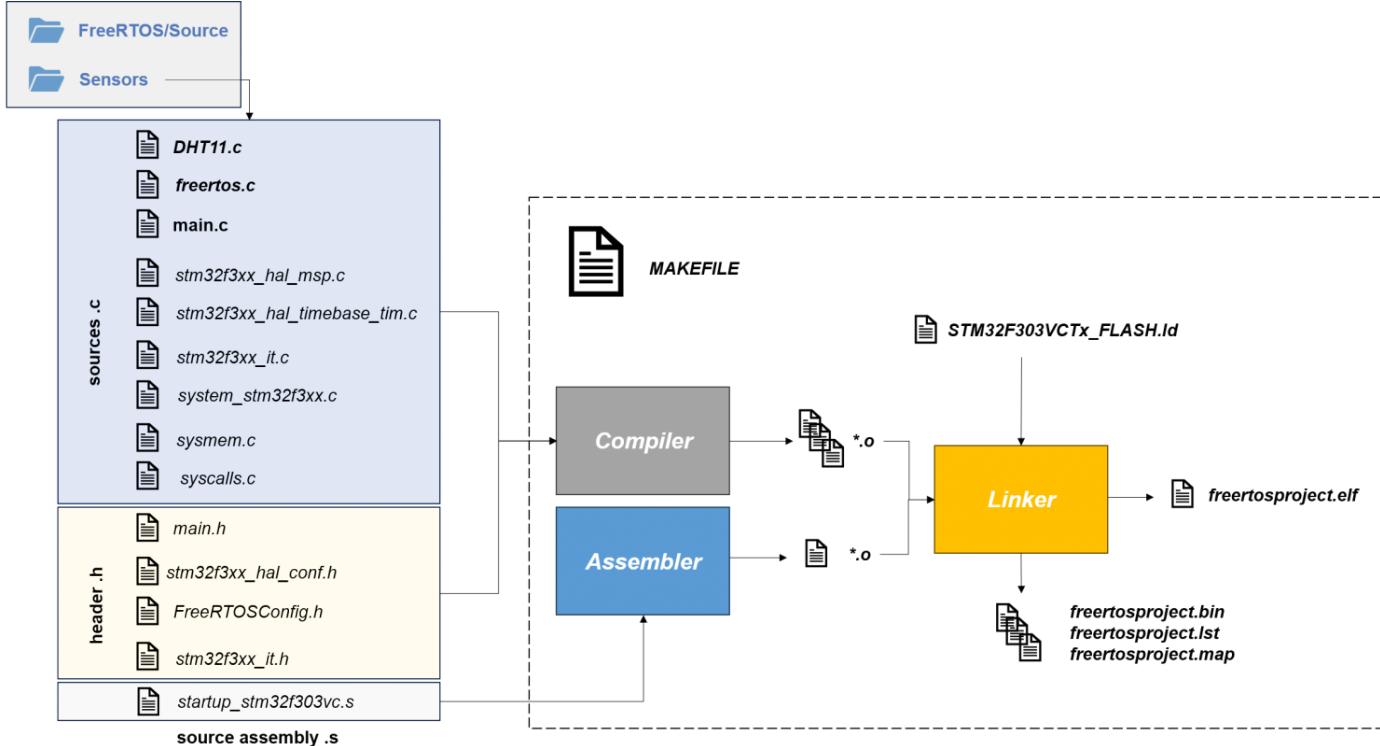
debug:
    @if ! nc -z localhost 3333; then \
        @echo "\n\t[Error] OpenOCD is not running!\n"; exit 1; \
    else \
        $(GDB) -ex "target extended localhost:3333" \
            -ex "monitor arm semihosting enable" \
            -ex "monitor reset halt" \
            -ex "load" \
            -ex "monitor reset init" \
            $(GDBFLAGS) $(TARGET).elf; \
    fi

prepare: src
    cp $(SRCS) src/
    cp $(LDFILE) src/linkerScript.ld
    cp Sensors/* src/

clean:
    @echo "[RM]      $(TARGET).bin"; rm -f $(TARGET).bin
    @echo "[RM]      $(TARGET).elf"; rm -f $(TARGET).elf
    @echo "[RM]      $(TARGET).map"; rm -f $(TARGET).map
    @echo "[RM]      $(TARGET).lst"; rm -f $(TARGET).lst
    @echo "[RM]      src files"; rm -f src/*
    @echo "[RM]      ld script"; rm -f src/linkerScript.ld
    @echo "[RMDIR]  dep" ; rm -fr dep
    @echo "[RMDIR]  obj" ; rm -fr obj
    @echo "[RMDIR]  obj" ; rm -fr src

```

Schema della gerarchia dei file che compongono il progetto. Dai file.c, .h, .o, passando per il linkerscript, fino al .elf



Schema della gerarchia dei file che compongono il progetto. Dai file.c, .h, .o, passando per il linkerscript, fino al .elf

In particolare:

- **stm32f3xx\_hal\_msp.c**: Fa parte della libreria HAL (Hardware Abstraction Layer) per i microcontrollori STM32F3xxx. Contiene le funzioni per l'inizializzazione e la configurazione dei periferiche di memoria nel microcontrollore.
- **stm32f3xx\_it.c**: Fa parte della libreria HAL per i microcontrollori STM32F3xxx. Contiene le funzioni degli interrupt per il microcontrollore, che vengono chiamate quando si verificano specifici interrupt.
- **stm32f3xx\_timebase\_tim.c**: Fa parte della libreria HAL per i microcontrollori STM32F3xxx. Contiene le funzioni per la configurazione e l'utilizzo dei timer nel microcontrollore, che possono essere utilizzati per generare interruzioni periodiche o misurare il tempo tra eventi.
- **system\_stm32f3xx.c**: È un file di sistema essenziale per l'utilizzo dei microcontrollori STM32F3xxx. Fornisce l'implementazione di funzioni e definizioni di base richieste per il corretto funzionamento del microcontrollore e la configurazione del sistema, come la gestione degli interrupt e l'inizializzazione del clock di sistema.
- **sysmem.c** e **syscall.c**: sono funzioni utili per la gestione della memoria e responsabili alle chiamate di sistema.

## Tool per il debug

La fase di debug rappresenta una delle operazioni più importanti all'interno dello sviluppo di una qualsiasi applicazione SW, non solo nell'ambito embedded, infatti essa ricopre circa il 20% dell'intero ciclo di sviluppo. Il debug consente di individuare e risolvere errori, anomalie o problemi che si verificano durante l'esecuzione del software su un sistema embedded. Infatti, tale fase viene frequentemente utilizzata per i seguenti fini:

- Identificazione degli errori;
- Analisi del comportamento;
- Ottimizzazione delle prestazioni;
- Verifica della correttezza.

Per il debug dell'applicazione sviluppata su STM32F303VC, sono disponibili diversi tool tra cui la toolchain GNU ARM e altri strumenti. Alcuni degli strumenti più comunemente utilizzati sono:

- **GNU ARM Embedded Toolchain:** La GNU ARM Embedded Toolchain è una suite di strumenti di compilazione, assemblaggio e debug fornita da GNU. Include il compilatore GCC per ARM, il debugger GDB e altre utility di supporto;
- **STM32CubeIDE:** STM32CubeIDE è un ambiente di sviluppo integrato (IDE) fornito da STMicroelectronics specificamente per la programmazione e il debug dei microcontrollori STM32;
- **STM32CubeProgrammer:** STM32CubeProgrammer è uno strumento di programmazione e debug dedicato ai microcontrollori STM32. Supporta il debug tramite la connessione di un debugger esterno, come ST-Link, J-Link o altri, consentendo l'esecuzione passo-passo, l'analisi dei registri, il monitoraggio delle variabili e altre funzionalità di debug avanzate;
- **Segger J-Link:** J-Link è un popolare debugger hardware prodotto da Segger. Supporta una vasta gamma di microcontrollori ARM, offre funzionalità avanzate di debug, tra cui il tracciamento degli eventi, la visualizzazione dei registri, il debug a basso livello e altro ancora;
- **Eclipse con plugin GNU ARM:** L'IDE Eclipse può essere esteso con plugin specifici per lo sviluppo embedded, come il plugin GNU ARM Eclipse. Questo plugin fornisce un'interfaccia per l'utilizzo della GNU ARM Embedded Toolchain all'interno di Eclipse.

La scelta del tool per il debug dipenderà dalle preferenze personali, dalle esigenze specifiche del progetto e dalle funzionalità richieste per il debug e lo sviluppo.

#### **Toolchain ARM: GDB**

Pertanto si è scelto di usare GDB che offre le seguenti funzionalità:

- **Avvio e stop del programma:** GDB consente di avviare l'esecuzione del programma sotto controllo del debugger e di interromperlo in punti specifici del codice.
- **Breakpoints:** È possibile impostare punti di interruzione nel codice sorgente o all'indirizzo di un'istruzione specifica, consentendo di fermare l'esecuzione del programma in quei punti per ispezionare lo stato.
- **Stepping:** GDB permette di eseguire il programma istruzione per istruzione, consentendo di controllare l'esecuzione passo dopo passo e di analizzare lo stato delle variabili e dei registri durante l'esecuzione.
- **Ispezione delle variabili e dei registri:** È possibile esaminare il valore delle variabili e dei registri in qualsiasi punto durante l'esecuzione del programma, consentendo di individuare errori o comportamenti anomali.
- **Visualizzazione dello stack:** GDB fornisce informazioni sullo stack di chiamate, consentendo di tracciare il flusso di esecuzione e identificare possibili problemi di stack overflow o di chiamate errate.
- **Analisi dei core dump:** GDB può essere utilizzato per analizzare i core dump generati in caso di crash del programma, consentendo di identificare la causa del crash e di esaminare lo stato del programma al momento del crash.

Applichiamo queste funzionalità con l'utilizzo di OpenOCD.

#### **OpenOCD**

OpenOCD (Open On-Chip Debugger) è un framework open source utilizzato per il debug e la programmazione di microcontrollori e processori embedded. Quando combinato con il GNU Debugger (GDB), OpenOCD offre diverse funzionalità utili durante il processo di sviluppo e debug, quali:

- **Debug basato su JTAG/SWD:** OpenOCD consente di comunicare con il microcontrollore o il processore target tramite le interfacce di debug JTAG (Joint Test Action Group) o SWD (Serial Wire Debug);
- **Flashing del firmware:** OpenOCD supporta la programmazione del firmware nel microcontrollore o nel processore target. Può scrivere il firmware nell'area di memoria appropriata del dispositivo target utilizzando file di immagine binaria o altri formati supportati;
- **Monitoraggio dei registri e della memoria:** OpenOCD consente di accedere e monitorare i registri interni del dispositivo target, come registri di controllo, registri di stato, registri di periferiche, etc;
- **Breakpoint e trace:** OpenOCD supporta la configurazione e l'utilizzo dei breakpoint, che consentono di interrompere l'esecuzione del codice in un punto specifico per il debug. Inoltre, offre anche funzionalità di trace che consentono di registrare e visualizzare le informazioni sull'esecuzione del codice, consentendo una maggiore comprensione del flusso di esecuzione del programma;
- **Supporto multipli processori e microcontrollori:** OpenOCD supporta una vasta gamma di microcontrollori e processori embedded provenienti da diversi produttori. È in grado di gestire le differenze di architettura, protocollo di debug e configurazioni specifiche per supportare una varietà di dispositivi embedded;
- **Integrazione con GDB:** OpenOCD funziona come un server di debug per GDB, consentendo a GDB di comunicare con il dispositivo target tramite OpenOCD usando una TCP socket. Questa integrazione consente a GDB di sfruttare tutte le funzionalità di debug fornite da OpenOCD, come il monitoraggio dei registri, i breakpoint, il flashing del firmware e altro ancora.

Per il caso in esame si è optato per la configurazione che prevede l'**uso di OpenOCD (server) in combinazione con GDB (client)**, in quanto tali strumenti risultano open source e altamente flessibili rispetto alle esigenze specifiche del progetto.

```

/c/Users/giuse/Documents/EmbeddedSystems/ProgettoSTM32F3_FreeRTOS
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=i686-w64-mingw32 --target=arm-n
one-eabi".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from freertosproject.elf...
Remote debugging using localhost:3333
Nebulizer_Task (argument=<optimized out>) at ./src/main.c:180
180      __asm("bkpt 4");
semihosting is enabled
Unable to match requested speed 1000 kHz, using 950 kHz
unable to match requested speed 1000 kHz, using 950 kHz
[stm32f3x.cpu] halted due to debug-request, current mode: Thread
XPSR: 0x01000000 pc: 0x080004d0 msp: 0x2000a000, semihosting
Loading section .isr_vector, size 0x188 lma 0x80000000
Loading section .text, size 0x4414 lma 0x80000190
Loading section .rodata, size 0x64 lma 0x80045a4
Loading section .init_array, size 0x8 lma 0x8004608
Loading section .fini_array, size 0x4 lma 0x8004610
Loading section .data, size 0x43c lma 0x8004614
Loading section .bss, size 0x100 lma 0x8004618
Transfer rate: 12 kB/sec, 2716 bytes/write.
unable to match requested speed 1000 kHz, using 950 kHz
unable to match requested speed 1000 kHz, using 950 kHz
[stm32f3x.cpu] halted due to debug-request, current mode: Thread
XPSR: 0x01000000 pc: 0x080004d0 msp: 0x2000a000, semihosting
unable to match requested speed 8000 kHz, using 4000 kHz
unable to match requested speed 8000 kHz, using 4000 kHz
(gdb) |

```

```

/c/Users/giuse/Documents/EmbeddedSystems
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : clock speed 1000 kHz
Info : STLINK V2J39M27 (API v2) VID:PID 0483:3748
Info : Target voltage: 2.892163
Info : [stm32f3x.cpu] cortex-M4 r0p1 processor detected
Info : [stm32f3x.cpu] target has 6 breakpoints, 4 watchpoints
Info : starting gdb server for stm32f3x.cpu on 3333
Info : Listening on port 3333 for gdb connections
[stm32f3x.cpu] halted due to breakpoint, current mode: Thread
XPSR: 0x21000000 pc: 0x080026f6 psp: 0x200013e8
Info : accepting 'gdb' connection on tcp/3333
Error: attempted 'gdb' connection rejected
Info : accepting 'gdb' connection on tcp/3333
Info : device id = 0x10036422
Info : flash size = 256 kB
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : Unable to match requested speed 1000 kHz, using 950 kHz
[stm32f3x.cpu] halted due to debug-request, current mode: Thread
XPSR: 0x01000000 pc: 0x080004d0 msp: 0x2000a000, semihosting
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : Unable to match requested speed 1000 kHz, using 950 kHz
[stm32f3x.cpu] halted due to debug-request, current mode: Thread
XPSR: 0x01000000 pc: 0x080004d0 msp: 0x2000a000, semihosting
Info : Unable to match requested speed 8000 kHz, using 4000 kHz
Info : Unable to match requested speed 8000 kHz, using 4000 kHz
Info : Padding image section 0 at 0x08000188 with 8 bytes
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : Unable to match requested speed 1000 kHz, using 950 kHz
[stm32f3x.cpu] halted due to debug-request, current mode: Thread
XPSR: 0x01000000 pc: 0x080004d0 msp: 0x2000a000, semihosting
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : Unable to match requested speed 1000 kHz, using 950 kHz
[stm32f3x.cpu] halted due to debug-request, current mode: Thread
XPSR: 0x01000000 pc: 0x080004d0 msp: 0x2000a000, semihosting
Info : Unable to match requested speed 8000 kHz, using 4000 kHz
Info : Unable to match requested speed 8000 kHz, using 4000 kHz

```

GDB e OpenOCD

## Verifica della gestione delle priorità e sincronizzazione

Tramite la fase di debug, si è verificato l'ordine di esecuzione dei task al fine di determinare se l'RTOS gestisce le priorità dei task ed effettui la sincronizzazione nel modo corretto. In particolare, all'interno delle funzioni relative ai 4 task sono stati inseriti dei **breakpoint SW** tramite la direttiva `__asm("bkpt")` che permette di scrivere istruzioni di basso livello assembler direttamente nel codice C.

Attraverso GDB, si procede all'esecuzione step-by-step dell'applicazione, con il comando “c” andiamo ad eseguire tutte le istruzioni fino al successivo breakpoint, come è possibile vedere dalla Figura successiva, i task del sistema si alternano nello stato Running in modo coerente con quanto atteso dal diagramma illustrato in precedenza. Possiamo, dunque, evincere da questo risultato che il kernel FreeRTOS gestisce in maniera corretta le priorità e la sincronizzazione tra i task.

```

(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
DHT11_Task (argument=<optimized out>) at ./src/main.c:132
132      __asm("bkpt 1");
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
waterLevel_Task (argument=<optimized out>) at ./src/main.c:75
75      __asm("bkpt 2");
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
HandNeb_Task (argument=<optimized out>) at ./src/main.c:156
156      __asm("bkpt 3");
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
Nebulizer_Task (argument=<optimized out>) at ./src/main.c:180
180      __asm("bkpt 4");

```

Verifica della gestione delle priorità e sincronizzazione attraverso il debugging

## Verifica dello scambio di messaggi tramite coda

A questo punto, dopo aver verificato la corretta schedulazione dei task, si passa alla verifica del corretto scambio di messaggi tra i task tramite le opportune code create in precedenza. Tale verifica è stata effettuata ponendo il nebulizzatore in *due possibili configurazioni, acceso e spento*. In particolare, tramite GDB con il comando “p” è possibile stampare il valore corrente della variabile desiderata, nel caso in esame si osservano le variabili *Temperature*, *Humidity* e *ADC\_VAL* (*livello dell'acqua*). Dalla Figura successiva è possibile notare che l'umidità è superiore al 60%, quindi, il nebulizzatore non deve accendersi motivo per cui all'Handler Nebulizer arriva sulla **xQueue1** il bit **DHT11\_EN** pari a 0 ed al Nebulizer arriva sulla **xQueue3** il bit **NEBULIZER\_EN** pari a 0.

```
(gdb) p Temperature
$1 = 29
(gdb) p Humidity
$2 = 61
(gdb) p DHT11_EN
$3 = 0 '\000'
(gdb) p ADC_VAL
$4 = 783
(gdb) p WATERLEVEL_EN
$5 = 1 '\001'
(gdb) p NEBULIZER_EN
$6 = 0 '\000'
```

Verifica dello scambio di messaggi tramite coda - Prima configurazione

Nella *seconda configurazione*, tramite il medesimo comando GDB visto in precedenza si può notare che l'umidità in questo caso è inferiore al 60%, quindi, il nebulizzatore deve essere acceso motivo per cui all'Handler Nebulizer arriva sulla **xQueue1** il bit **DHT11\_EN pari a 1** ed al **Nebulizer** arriva sulla **xQueue3** il bit **NEBULIZER\_EN pari a 1**.

```
(gdb) p Temperature
$1 = 29
(gdb) p Humidity
$2 = 58
(gdb) p DHT11_EN
$3 = 1 '\001'
(gdb) p ADC_VAL
$4 = 862
(gdb) p WATERLEVEL_EN
$5 = 1 '\001'
(gdb) p NEBULIZER_EN
$6 = 1 '\001'
```

Verifica dello scambio di messaggi tramite coda - Seconda configurazione

Quindi, possiamo affermare che lo scambio di messaggi tramite la coda avviene correttamente tra i task ed il nebulizzatore si comporta come atteso.

## Confronto dell'applicazione nella versione con FreeRTOS e Bare Metal

Si fa un confronto qualitativo e non tra l'applicazione nella versione con FreeRTOS e senza per valutarne le differenze essenziali ed i vantaggi dell'una rispetto all'altra.

```
/c/Users/giuse/Documents/EmbeddedSystems/ProgettoSTM32F3_noFreeRTOS
one-eabi".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
 <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from nofreertosproject.elf...
Remote debugging using localhost:3333
0x08001bc2 in HAL_GetTick O at ./src/stm32f3xx_hal.c:291
291         return uwTick;
semihosting is enabled
unable to match requested speed 1000 kHz, using 950 kHz
unable to match requested speed 1000 kHz, using 950 kHz
[stm32f3x.cpu] halted due to debug-request, current mode: Thread
XPSR: 0x01000000 pc: 0x08001ae8 msp: 0x2000a000, semihosting
Loading section .isr_vector, size 0x188 lma 0x8000000
Loading section .text, size 0x9c10 lma 0x80001c0
Loading section .rodata, size 0x4a8 lma 0x8009dd0
Loading section .ARM, size 0x8 lma 0x800a278
Loading section .init_array, size 0x8 lma 0x800a280
Loading section .fini_array, size 0x4 lma 0x800a288
Loading section .data, size 0x9c8 lma 0x800a28c
start address 0x08001ae8, load size 44060
Transfer rate: 17 KB/sec, 4895 bytes/write.
unable to match requested speed 1000 kHz, using 950 kHz
unable to match requested speed 1000 kHz, using 950 kHz
[stm32f3x.cpu] halted due to debug-request, current mode: Thread
XPSR: 0x01000000 pc: 0x08001ae8 msp: 0x2000a000, semihosting
unable to match requested speed 8000 kHz, using 4000 kHz
unable to match requested speed 8000 kHz, using 4000 kHz
(gdb) c
Continuing.

/c/Users/giuse/Documents/EmbeddedSystems
Info : Padding image section 0 at 0x08000188 with 56 bytes
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : Unable to match requested speed 1000 kHz, using 950 kHz
[stm32f3x.cpu] halted due to debug-request, current mode: Thread
XPSR: 0x01000000 pc: 0x08001ae8 msp: 0x2000a000, semihosting
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : Unable to match requested speed 1000 kHz, using 950 kHz
[stm32f3x.cpu] halted due to debug-request, current mode: Thread
XPSR: 0x01000000 pc: 0x08001ae8 msp: 0x2000a000, semihosting
Info : Unable to match requested speed 8000 kHz, using 4000 kHz
Info : Unable to match requested speed 8000 kHz, using 4000 kHz
Temperature: 29   Humidity: 58
Acqua: 1
Nebulizzatore acceso!
```

Esecuzione Bare Metal (No FreeRTOS)

### Tempi di esecuzione

Si valutano le differenze sui tempi di esecuzione (Execution Time ET) dei rispettivi task tra FreeRTOS e Bare Metal:

FreeRTOS

```
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
DHT11_Task (argument=<optimized out>) at ./src/main.c:132
132      __asm__("bkpt 1");
(gdb) p Task_ET[0]
$28 = 500
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
waterLevel_Task (argument=<optimized out>) at ./src/main.c:75
75      __asm__("bkpt 2");
(gdb) p Task_ET[1]
$29 = 0
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
HandNeb_Task (argument=<optimized out>) at ./src/main.c:156
156      __asm__("bkpt 3");
(gdb) p Task_ET[2]
$30 = 250
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
Nebulizer_Task (argument=<optimized out>) at ./src/main.c:180
180      __asm__("bkpt 4");
(gdb) p Task_ET[3]
$31 = 228
```

Tempi di esecuzione FreeRTOS

#### Bare Metal

```
(gdb) p Task_ET[0]
$5 = 22
(gdb) p Task_ET[1]
$6 = 0
(gdb) p Task_ET[2]
$7 = 0
```

Tempi di esecuzione Bare Metal (No FreeRTOS)

Dai risultati si nota che le medesime funzioni che gestiscono i sensori ed il nebulizzatore nella versione Bare Metal sono risultate molto più veloci rispetto alla configurazione FreeRTOS. Questo può dipendere da vari fattori, uno possibile è l'overhead del sistema operativo FreeRTOS che introduce vari meccanismi per la gestione dei task, la sincronizzazione e la comunicazione tra i task e altre funzionalità del kernel per aumentare la predicitività e affidabilità del sistema. Se questo overhead può influire sul tempo di esecuzione dei singoli task in maniera negativa, d'altro canto risulta una soluzione più adatta in contesti real-time rispetto alla versione bare metal, la quale non ha nessun tipo di controllo.

**Si noti:** Quando abbiamo un tempo di esecuzione inferiore ad 1 ms viene mostrato un valore pari a 0, essendo che la risoluzione del clock (1 MHz) non permette di misurare tale tempo.

#### Utilizzo della memoria

Per valutare la differenza nell'utilizzo della memoria tra la versione FreeRTOS dell'applicazione e non, nel makefile, si aggiunge il seguente flag:

- `--print-memory-usage`: Il seguente comando stampa informazioni sulla memoria, inclusa la dimensione utilizzata e totale delle regioni, utile per valutare la disponibilità di memoria su target embedded.

Memory Region	Used Size	Region Size	%age used
CCMRAM:	0 GB	8 KB	0.00%
RAM:	4520 B	40 KB	11.04%
FLASH:	44116 B	256 KB	16.83%

[OBJDUMP] nofreertosproject.lst

arm-none-eabi-objdump -st nofreertosproject.elf >nofreertosproject.lst

[SIZE] nofreertosproject.elf

arm-none-eabi-size nofreertosproject.elf

text data bss dec hex filename

41544 2516 2016 46076 b3fc nofreertosproject.elf

[OBJCOPY] nofreertosproject.bin

arm-none-eabi-objcopy -O binary nofreertosproject.elf nofreertosproject.bin

Memory Region	Used Size	Region Size	%age used
CCMRAM:	0 GB	8 KB	0.00%
RAM:	7048 B	40 KB	17.21%
FLASH:	19024 B	256 KB	7.26%

[OBJDUMP] freertosproject.lst

arm-none-eabi-objdump -st freertosproject.elf >freertosproject.lst

[SIZE] freertosproject.elf

arm-none-eabi-size freertosproject.elf

text data bss dec hex filename

17920 1096 5964 24980 6194 freertosproject.elf

[OBJCOPY] freertosproject.bin

arm-none-eabi-objcopy -O binary freertosproject.elf freertosproject.bin

Utilizzo della memoria a confronto (Screen 1 - Bare Metal) e (Screen 2 - FreeRTOS)

Nel makefile:

```
# Linker flags
LDFLAGS = -Wl,--print-memory-usage -Wl,--gc-sections
          -Wl,-Map=$(TARGET).map $(LIBS) -Tsrc/linkerScript.ld
```

La memoria occupata in RAM è maggiore nella versione FreeRTOS, sempre per l'overhead aggiuntivo introdotto dall' RTOS, come la gestione dei task, dei semafori, delle code e altre funzionalità offerte dal kernel. Questo è un trade-off che si deve tenere in considerazione quando si decide di utilizzare FreeRTOS: l'efficienza e la flessibilità offerte dal sistema operativo vengono a discapito di un maggior utilizzo di risorse, come la memoria.

## Tempi di compilazione

Per valutare i tempi di compilazione si fa uso, nel makefile, del seguente flag:

- ftime-report: è utilizzato per generare un report dettagliato sui tempi di compilazione delle singole fasi del processo di compilazione.

```
generating "obj/main.o" from "./src/main.c"
arm-none-eabi-gcc -Wall -g -std=c99 -Os -mlittle-endian -mcpu=cortex-m4 -march=armv7e-m -mthumb -mfpu=fpv4-sp-d16 -mfloat-abi=hard -ffunction-sections -fdata-sections -ftime-report -Isrc -I/c/Users/giuse/Documents/EmbeddedSystems/STM32CubeF3/Drivers/CMSIS/Include -I/c/Users/giuse/Documents/EmbeddedSystems/STM32CubeF3/Drivers/STM32F3xx_HAL_Driver/Inc -I./FreeRTOS/Source/include -I./FreeRTOS/Source/portable/GCC/ARM_CM4F -DSTM32F303XC -DUSE_HAL_DRIVER -DUSE_DBPRINTF -c -o obj/main.o ./src/main.c -MMD -MP -MF dep/main.d
```

Time Variable	usr	sys	wall	GGC
phase setup	: 0.00 ( 0%)	2090 kB ( 23%)		
phase parsing	: 0.03 ( 39%)	5331 kB ( 59%)		
phase opt and generate	: 0.05 ( 61%)	1511 kB ( 17%)		
callgraph functions expansion	: 0.03 ( 39%)	1030 kB ( 11%)		
callgraph ipa passes	: 0.02 ( 22%)	216 kB ( 2%)		
register scan	: 0.02 ( 20%)	0 kB ( 0%)		
preprocessing	: 0.01 ( 19%)	3690 kB ( 41%)		
inline parameters	: 0.02 ( 20%)	22 kB ( 0%)		
final	: 0.02 ( 19%)	30 kB ( 0%)		
symout	: 0.02 ( 20%)	642 kB ( 7%)		
TOTAL	: 0.08	8993 kB		

```
generating "obj/main.o" from "./src/main.c"
arm-none-eabi-gcc -Wall -g -std=c99 -Os -mlittle-endian -mcpu=cortex-m4 -march=armv7e-m -mthumb -mfpu=fpv4-sp-d16 -mfloat-abi=hard -ffunction-sections -fdata-sections -ftime-report -Isrc -I/c/Users/giuse/Documents/EmbeddedSystems/STM32CubeF3/Drivers/CMSIS/Include -I/c/Users/giuse/Documents/EmbeddedSystems/STM32CubeF3/Drivers/STM32F3xx_HAL_Driver/Inc -DSTM32F303XC -DUSE_HAL_DRIVER -DUSE_DBPRINTF -c -o obj/main.o ./src/main.c -MMD -MF dep/main.d
```

Time Variable	usr	sys	wall	GGC
phase setup	: 0.01 ( 7%)	2090 kB ( 24%)		
phase parsing	: 0.04 ( 49%)	5441 kB ( 63%)		
phase opt and generate	: 0.03 ( 38%)	1041 kB ( 12%)		
callgraph functions expansion	: 0.03 ( 33%)	808 kB ( 9%)		
preprocessing	: 0.03 ( 32%)	3742 kB ( 43%)		
TOTAL	: 0.08	8620 kB		

Utilizzo della memoria a confronto (Screen 1 - FreeRTOS) e (Screen 2 - Bare Metal)

Quando si utilizza il flag "-ftime-report" durante la compilazione di un programma, il compilatore GNU fornisce informazioni sul tempo impiegato per eseguire le diverse fasi della compilazione, come la fase di preprocessing, la fase di analisi lessicale e sintattica, la fase di ottimizzazione e la fase di generazione del codice.

Il report generato include il tempo impiegato da ciascuna fase, suddiviso per singoli file sorgente, e può fornire utili informazioni per identificare le parti del codice che richiedono più tempo di compilazione.

Nel makefile:

```
# Compiler flags
...
CFLAGS += -ffunction-sections -fdata-sections -ftime-report
...
```

È evidente che, sia nel caso dell'utilizzo di FreeRTOS che nel caso di un'applicazione Bare Metal, i tempi di compilazione sono simili, con l'eccezione delle fasi di compilazione aggiuntive necessarie per gestire il sistema operativo in tempo reale in caso di FreeRTOS.

## Analisi qualitativa

L'analisi qualitativa dei vantaggi di utilizzare il kernel FreeRTOS rispetto a un'applicazione senza kernel può essere suddivisa in diversi aspetti chiave.

- **Gestione delle risorse:** FreeRTOS, come si è visto, fornisce un sistema di gestione delle risorse che permette di definire e allocare in modo efficiente le risorse hardware disponibili tra le diverse attività. Questo riduce la complessità e il rischio di errori rispetto alla gestione manuale delle risorse in un'applicazione senza kernel.
- **Multithreading:** FreeRTOS supporta il multithreading, consentendo l'esecuzione simultanea di più task indipendenti. Questa caratteristica facilita lo sviluppo di sistemi embedded complessi in cui diversi task devono essere eseguiti in parallelo. Al contrario, un'applicazione senza FreeRTOS potrebbe basarsi su un modello sequenziale, eseguendo una sola attività alla volta, limitando la flessibilità e l'efficienza complessiva.
- **Sincronizzazione e comunicazione:** FreeRTOS fornisce meccanismi di sincronizzazione e comunicazione tra le attività, come semafori, code, mutex, ecc. Questi strumenti consentono alle attività di cooperare fra loro in modo sicuro e coordinato. In un'applicazione senza FreeRTOS, sarebbe necessario implementare manualmente tali meccanismi, aumentando la complessità del codice e la possibilità di errori.
- **Scalabilità:** FreeRTOS è progettato per essere altamente scalabile, consentendo di gestire sistemi embedded di diverse dimensioni e complessità. Può essere utilizzato su microcontrollori con pochi kilobyte di memoria RAM fino a dispositivi più potenti con centinaia di kilobyte o addirittura megabyte di memoria. Un'applicazione senza FreeRTOS potrebbe non essere altrettanto flessibile o adattabile a diverse piattaforme.

## Conclusioni

In conclusione, l'utilizzo di FreeRTOS offre numerosi vantaggi, come una gestione efficiente delle risorse, supporto al multithreading, meccanismi di sincronizzazione e comunicazione predefiniti e la scalabilità del sistema il tutto personalizzabile. Questi vantaggi semplificano lo sviluppo di sistemi embedded complessi, migliorando l'efficienza e l'efficacia riducendo la possibilità di errori nella gestione delle risorse e nella sincronizzazione delle attività.

## Bibliografia

1. FreeRTOS Real Time Kernel (RTOS) (<https://www.freertos.org/RTOS.html>)
2. GCC online documentation - Option Summary (<https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html>)
3. GNU Toolchain | Arm Developer (<https://developer.arm.com/Tools%20and%20Software/GNU%20Toolchain>)
4. ES\_AirHumidifier\_FreeRTOS - GitHub Repository degli Autori: Antonio Romano - Giuseppe Riccio ([https://github.com/giuseppericcio/ES\\_AirHumidifier\\_FreeRTOS](https://github.com/giuseppericcio/ES_AirHumidifier_FreeRTOS))
5. Binutils - ld Documentation: Memory Usage (<https://sourceware.org/binutils/docs-2.40/ld.html#index-memory-usage>)
6. GCC online documentation - Developer Options (<https://gcc.gnu.org/onlinedocs/gcc/Developer-Options.html>)

Retrieved from "[http://www.naplespu.com/es/index.php?title=Progetto\\_di\\_un%27applicazione\\_basata\\_su\\_kernel\\_FreeRTOS\\_per\\_il\\_controlloAutomatico\\_di\\_un\\_umidificatore\\_e\\_confronto\\_con\\_versione\\_Bare\\_Metal&oldid=7542](http://www.naplespu.com/es/index.php?title=Progetto_di_un%27applicazione_basata_su_kernel_FreeRTOS_per_il_controlloAutomatico_di_un_umidificatore_e_confronto_con_versione_Bare_Metal&oldid=7542)"

This page was last edited on 25 July 2023, at 16:13.