Relazione Real Time Systems and Applications

Gruppo 02

Battipaglia Valerio

Caso Antonio

Dell'Orto Giuseppe

De Stefano Sabatino

January 8, 2024

Contents

1 Introduzione

1.1 Contesto realizzativo

Nell'ambito della progettazione di sistemi embedded real-time, uno dei problemi più critici è la gestione efficiente dei task e delle risorse. Questo problema diventa particolarmente complesso quando si tratta di sistemi che richiedono l'esecuzione di task periodici e aperiodici, che devono essere schedulati e coordinati in modo da garantire il rispetto dei vincoli temporali.

In particolare, il problema affrontato in questo progetto riguarda la progettazione e l'implementazione di un sistema di scheduling per un'applicazione basata su FreeRTOS e la libreria ESFree. L'applicazione richiede l'esecuzione di vari task, tra cui il controllo dei sensori, la comunicazione tra schede e la gestione di situazioni di emergenza. Questi task hanno requisiti diversi in termini di periodicità e priorità, e devono essere gestiti in modo da garantire il funzionamento corretto del sistema.

Un aspetto critico del problema è la necessità di sincronizzare i task di comunicazione su due schede diverse. Questi task devono essere in grado di comunicare efficacemente tra loro, il che richiede una gestione accurata del timing e della sincronizzazione.

In sintesi, il problema affrontato in questo progetto riguarda la progettazione e l'implementazione di un sistema di scheduling per un'applicazione real-time complessa, che richiede la gestione di task periodici e aperiodici, la sincronizzazione della comunicazione tra schede, la gestione efficiente delle risorse e la garanzia di sicurezza e affidabilità.

1.2 Strumenti utilizzati

1.2.1 FreeRTOS

FreeRTOS è un sistema operativo real-time (RTOS) open source concepito per microcontrollori e piccoli microprocessori. Scritto in linguaggio C, FreeRTOS si distingue per le sue dimensioni contenute e il basso overhead. Tra le sue funzionalità principali vi sono il multithreading attraverso task, code, semafori, timer software e mutex. L'API standardizzata offerta dal kernel semplifica notevolmente lo sviluppo di componenti software richiedenti funzionalità RTOS, comportando vantaggi sia per gli utenti che per l'industria del software. L'approccio unificato di FreeRTOS riduce gli sforzi di apprendimento e agevola la condivisione di componenti software. Inoltre, i componenti middleware che sfruttano FreeRTOS risultano indipendenti dall'RTOS e quindi più facili da adattare.

Sempre in tema di riduzione delle difficoltà di apprendimento, CMSIS_RTOS API v2, o CMSIS_RTOS2, è un'interfaccia API generica progettata per interfacciarsi con un kernel RTOS. La definizione completa dell'interfaccia API si trova nel

file di intestazione cmsis_os2.h. CMSIS_RTOS2 offre funzionalità di base necessarie in molte applicazioni, contribuendo a ridurre gli sforzi di apprendimento e semplificare la condivisione di componenti software. Analogamente a FreeRTOS, i componenti middleware che utilizzano CMSIS_RTOS2 sono indipendenti dall'RTOS, rendendoli più flessibili nell'adattamento.

Riassumendo, CMSIS_RTOS2 fornisce un'astrazione che semplifica lo sviluppo di applicazioni embedded, consentendo agli sviluppatori di concentrarsi sulle specifiche della loro applicazione senza doversi preoccupare degli aspetti intricati della gestione del sistema operativo in tempo reale.

1.2.2 Polling Server

Nel contesto degli algoritmi di scheduling, il Polling Server è un algoritmo utilizzato per gestire l'esecuzione di task aperiodici in un sistema real-time. L'idea di base del Polling Server è quella di dedicare una certa quantità di capacità di elaborazione a un "server" che esegue task aperiodici quando non ci sono task periodici da eseguire. Esso è un algoritmo fondamentale nei sistemi operativi real-tim, dove un singolo server (un task) serve più clienti (task a loro volta ma aperiodici) in un ordine specifico. L'obiettivo è massimizzare l'efficienza del server minimizzando il tempo di attesa.

Come ogni altro task periodico, un server è caratterizzato da un periodo T_s e un tempo di computazione C_s, detto **Capacità del Server**. Il polling server, in quanto egli stesso task periodico, viene schedulato tramite lo stesso algoritmo (nel nostro caso Rate Monotonic) di ogni altro task periodico. Quando attivo, gestisce le richieste aperiodiche nei limiti del suo budget. L'ordine secondo cui vengono servite le richieste aperiodiche non dipende dall'algoritmo di scheduling usato per i task periodici, e può essere svolto in maniera arbitraria o sulla base dei tempi di arrivo (simulati dalla sequenzialità delle creazioni dei task aperiodici all'interno del task *CheckDanger*, così come realizzato nel progetto in discussione), sui tempi di computazione o sulle deadline.

A intervalli regolari uguali al periodo del Polling Server, il server diventa **attivo** e serve le richieste aperiodiche in attesa di essere servite, rispettando però i limiti imposti dalla sua capacità C_s. Se, allo scattare del periodo T_s, non sono presenti richieste aperiodiche in attesa, il polling server sospende se stesso fino all'inizio del suo prossimo periodo, e il budget originariamente allocato per i task aperiodici viene perso e viene dato spazio ai task periodici.

Questo implica che, se una prima richiesta aperiodica sopraggiunge l'istante immediatamente successivo all'inizio del periodo del Polling Server, essa non beneficerà del budget del polling server, e dovrà aspettare fino all'inizio del prossimo periodo.

Al fine di garantire un set di task periodici hard (schedulati tramite RM) in un contesto di task aperiodici soft, gestiti da un Polling Server, è presente il seguente test di schedulabilità:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} + \frac{C_s}{T_s} \leq (n+1)[2^{1/(n+1)} - 1].$$

Figure 1: Fattore di utilizzo per la schedulabilità del Polling Server.

1.2.3 Mutex

Un mutex, abbreviazione di Mutual Exclusion Object, è un tipo specifico di semaforo binario utilizzato per fornire un meccanismo di blocco. Il mutex è principalmente utilizzato per fornire l'esclusione reciproca a una specifica porzione del codice, in modo che il processo possa eseguire e lavorare con una particolare sezione del codice in un determinato momento. Il mutex utilizza un meccanismo di ereditarietà della priorità per evitare problemi di inversione di priorità. Il meccanismo di ereditarietà della priorità mantiene i processi di priorità superiore nello stato bloccato per il minor tempo possibile.

Nel contesto di FreeRTOS, i mutex sono semafori binari. Mentre i semafori binari sono la scelta migliore per implementare la sincronizzazione (tra task o tra task e un'interruzione), i mutex sono la scelta migliore per implementare una semplice esclusione reciproca. I mutex utilizzano le stesse funzioni API di accesso ai semafori, quindi permettono anche di specificare un tempo di blocco. Il tempo di blocco indica il numero massimo di *tick* che un task dovrebbe attendere nello stato "*Bloccato*" quando tenta di acquisire un mutex se il mutex non è immediatamente disponibile.

1.2.4 ESFree

ESFree è una libreria di scheduling open source che fornisce concetti teorici di tempo reale per FreeRTOS. Poiché ESFree non modifica il kernel di FreeRTOS, la libreria può essere utilizzata su qualsiasi piattaforma in grado di supportare FreeRTOS. Per poter utilizzare questa libreria di efficientamento bisogna semplicemente importare i file `scheduler.c` e `scheduler.h` all'interno della cartella progetto.

La libreria di scheduling proposta fornisce un server di polling che gestisce compiti aperiodici e sporadici, rilevando e gestendo in modo affidabile gli errori temporali, insieme a politiche di scheduling come Rate-Monotonic Scheduling (RMS), Deadline-Monotonic Scheduling (DMS) ed Earliest Deadline First (EDF). Queste politiche mirano a fornire funzionalità teoriche di scheduling in tempo reale, accelerando lo sviluppo di progetti complessi e rendendo FreeRTOS più accessibile agli studenti che hanno appena studiato la pianificazione in tempo reale.

E' importante inoltre notare che ESFree dispone del proprio TCB esteso chiamato SchedTCB, che include informazioni aggiuntive per la gestione di job periodici dallo spazio utente. ESFree ha inoltre definito i suoi Blocchi di Controllo dei

Lavori Aperiodici (AJCB) e dei Lavori Sporadici (SJCB) per gestire rispettivamente lavori aperiodici e sporadici. Alcune delle informazioni aggiuntive presenti in SchedTCB comprendono un puntatore alla funzione del job periodico (definita dall'utente), la priorità del compito, l'handler del compito, il tempo di rilascio, l'ultimo tempo di rilascio, la deadline relativa, la deadline assoluta, il periodo, il WCET e il contatore del tempo di esecuzione.

AJCB contiene le seguenti informazioni sui lavori aperiodici: un puntatore alla funzione del job aperiodico, il nome del job aperiodico, i parametri per la funzione del job, il WCET e il contatore del tempo di esecuzione.

SJCB contiene le seguenti informazioni sui job sporadici: un puntatore alla funzione del job sporadico, il nome del job sporadico, i parametri per la funzione del job, la deadline relativa, la deadline assoluta, il WCET e il contatore del tempo di esecuzione.

1.2.5 Rate Monotonic

Questo algoritmo di scheduling è rivolto a task periodici, e assegna priorità ai task in accordo ai loro periodi; vale a dire, minore sarà il periodo e maggiore sarà la priorità assegnata al task.

Dal momento che i periodi si suppongono invarianti, è possibile definire il RM come un algoritmo "fixed-priority": la priorità assegnata a ciascun task, prima delle sua esecuzione, non cambia nel tempo.

RM inoltre è preemptive: all'arrivo di un nuovo task con periodo più breve, l'esecuzione di un task con periodo maggiore sarà sospesa.

Contestualmente a tutti gli algoritmi "fixed-priority", RM risulta essere ottimale: nessun altro algoritmi a priorità fissa può schedulare un task set che non sia schedulabile da RM.

Un task set è schedulabile tramite RM se e solo se il fattore di utilizzo della cpi sia il seguente, dove n è il numero di task periodici nel set:

$$U = \sum_{i=1}^{n} \frac{C_i}{D_i} <= n\,(2^{\frac{1}{n}} - 1)$$

Figure 2: Fattore di ulizzo per la schedulabilità.

2 Design

2.1 Scheda Master

Nel corso della nostra iniziativa progettuale, si è meticolosamente identificato una serie di task, distinti tra periodici e aperiodici, nel contesto della scheda master. Il task primario, di natura critica, sovrintende la lettura dell'ultrasuono, analizzando con precisione i dati acquisiti per calcolare la distanza da potenziali ostacoli. A seguire, il task *CheckDanger* (periodico) agisce come un monitor costante di tali dati. Nell'eventualità in cui il *CheckDanger* rilevi anomalie, rilevando la presenza di ostacoli, si compie una decisione pronta ed efficace: i task di lettura e comunicazione vengono elusi, catapultando il sistema direttamente nella fase di esecuzione. In questo frangente, si procede alla fermata del sistema e alla disattivazione della seconda scheda. I task non critici, quali la lettura del joystick e dell'accelerometro, sono eseguiti solo in assenza di ostacoli rilevati, conformemente alle esigenze del sistema.

Per quanto riguarda la scelta dello schedulatore, abbiamo adottato un approccio basato su un server di ***Polling*** per la gestione dei task aperiodici. Questa decisione è stata motivata dalla necessità di mantenere un controllo costante e flessibile sui task aperiodici, permettendo la loro esecuzione in base alla disponibilità delle risorse. L'algoritmo di schedulazione sottostante per i task periodici (in cui ricade il server stesso) è il Rate Monotonic, noto per la sua efficienza e la prevedibilità delle priorità assegnate ai task real-time, basate sui rispettivi periodi (utilmente utilizzati come *deadline*). Questa scelta è risultata particolarmente adatta alle caratteristiche specifiche del nostro progetto, data la natura prettamente periodica del *task set*.

Nel contesto dell'implementazione dei task, il periodo del task dell'ultrasuono coincide con quello di un ciclo di controllo, garantendo così una rilevazione accurata e tempestiva degli ostacoli. Il task di polling, al contrario, presenta un periodo pari alla metà di quello del ciclo di controllo. Inizialmente, potrebbe non sarà necessaria l'esecuzione di task aperiodici, ma al termine dell'esecuzione del task *CheckDanger*, il server di polling avrà task aperiodici da gestire. Questa configurazione specifica crea una dipendenza sequenziale tra i task, assicurando che i task aperiodici vengano eseguiti solo quando strettamente necessario.

Queste decisioni progettuali, ponderate e consapevoli, sono state adottate con l'obiettivo di ottimizzare l'efficienza del sistema, garantire la puntualità nell'esecuzione dei task e mantenere un'elevata flessibilità nella gestione dei task aperiodici.

2.2 Scheda Slave

Passando ora alla scheda Slave, notiamo la presenza esclusiva di task periodici con specifici vincoli di schedulabilità. In merito a ciò, si è voluto dare maggiore rilevanza al task gestente l'*encoder*: per poter garantire che esso fosse sempre schedulato trascorso un preciso e sempre uguale periodo di tempo (in virtù

della sua configurazione meccanica-fisica), si è deciso di porlo come primo ad esser eseguito. Successivamente, vengono eseguiti i task di lettura del sensore di batteria e di temperatura. A seguire, si affronta il task di comunicazione, il quale necessita di sincronizzarsi con il corrispondente task di comunicazione sul master. Infine, il ciclo di esecuzione si conclude con il task di esecuzione, impiegato per poter indurre potenza ai motori.

2.3 Task Set

All'interno del modulo ***Master***, la complessità delle operazioni è articolata attraverso una serie di task attentamente progettati e orchestrati. Il primo tra questi, il task di lettura dell'*Ultrasuono*, svolge una analisi dei dati provenienti dal sensore, calcolando la distanza da eventuali ostacoli. Similmente tutti gli altri task del set, per definire le tempistiche richieste da tale operazione **critica** si è valutato sotto il profilo del Worst Case Execution Time (WCET), considerando scenari critici, insieme a tempi associati ai context switch, al fine di garantire la massima affidabilità.

A seguire, il *Task* ***CheckDanger*** opera come un costante monitoraggio dei dati ultrasuono, prontamente reagendo e rilevando eventuali problemi o ostacoli. Esso, ha inoltre il fondamentale compito di creare i job aperiodici sulla base dei valori riscontrati precedentemente. Anche in questo caso, i tempi di esecuzione considerano scenari estremi e context switch per garantire una risposta tempestiva in situazioni critiche.

I task non critici, quali la "Lettura del Joystick e dell'Accelerometro", intervengono solo in assenza di ostacoli rilevati, contribuendo così a ottimizzare l'utilizzo delle risorse. Il *task di* ***Comunicazione*** trasmette accuratamente i dati raccolti, considerando tempi di esecuzione massimi tenendo anche dovuto conto dei tempi richiesti dalle trasmissioni seriali e i necessari context switch.

Il *task di* ***Esecuzione*** implementa le azioni necessarie in base ai dati raccolti e alle decisioni prese, compiendo operazioni cruciali, spaziando dalla disabilitazione dello slave nel caso di ostacoli rilevati, andando fino alla semplice e regolare attivazione dei led oppure direzione dei motori.

Per quanto riguarda gli schedulatori, la scelta del *Rate Monotonic* è stata guidata dalla necessità di assegnare priorità in base ai periodi dei task. Nel caso del ***Server di Polling***, la sua implementazione è stata ponderata per consentire l'esecuzione flessibile dei task aperiodici in linea con la disponibilità delle risorse.

Nel modulo slave, il task dell'*Encoder* effettua la rilevazione della quantità angola trascorsa dalla precedente rilevazione, elaborando le informazioni e gestendole similmente a quanto avviene contestualmente alla scheda master. Seguono i task relativi alla gestione dei sensori rilevanti i livelli di*batteria* e di *temperatura*, attentamente progettati per mantenere una distanza temporale specifica tra successivi *dispatch* del medesimo task, e il task di *Comunicazione* con il Modulo Master, che sincronizza le comunicazioni con il task corrispondente nel modulo

master. Infine, il task di *Esecuzione* completa il ciclo di esecuzione, attuando le azioni necessarie basate sui dati ricevuti.

2.4 Iter progettuale

Inizialmente, l'intento progettuale partiva dal presupposto di voler introdurre un minor numero di task, optando per l'accorpamento di diverse funzionalità in favore di *macro* task; ad esempio, la lettura dei sensori sarebbe stata suddivisa per criticità e importanza di quest'ultimi, creando un task per la gestione dei sensori critici (batteria, temperatura e ultrasuono) e uno per i sensori a inferiore livello di criticità (encoder, accelerometro e controller psx).

Successivamente, l'idea è stata scartata in modo da andare a favorire un maggiore "parallelismo", stante la maggiore suddivisione e particolareggiamento delle singole attività da svolgere, cosa che avrebbe garantito un maggiore controllo e orchestrabilità della gestione dei task. Ovvero, un vantaggio subito riscontrabile consisteva nella inedita possibilità di poter identificare quali task (e quando) causassero problematiche di sforamento di WCET (Worst Case Execution Time) o deadline.

Una volta individuato quantomeno il numero dei task, il problema permaneva riguardo la loro periodicità o meno; vale a dire, se renderli task periodici, aperiodici o financo sporadici. Dapprincipio, si era ipotizzato l'utilizzo di task periodici stante la lapalissiana sequenzialità del ciclo di controllo: tuttavia, si è propeso per la scissione in task periodici e aperiodici sulla base del comportamento del sistema desiderato. E' altresì vero che, alcuni task, in situazioni specifiche, non saranno eseguiti stante il diverso flusso di controllo al quale il sistema viene sottoposto; banalmente, nel caso di rilevazione di una problematica, non si indulgerà nella lettura del controllo o del sensore di accelerazione bensì si procederà immediatamente con il task di esecuzione e dunque con la frenata di emergenza.

In questa fase, si è inoltre considerato l'utilizzo di task *sporadici* al posto di quelli aperiodici (come correntemente implementato), ma ciò non si è rivelato possibile a causa di problematiche riscontrate con la libreria ESFree. In particolare, qualunque parametro utilizzato (tempi di esecuzione bassi, deadline più ampie) comportava che il task non venisse accettato (il valore di ritorno della funzione di creazione del job sporadico corrispondeva ad avvenuto errore, non creando conseguentemente il job). Queste problematiche sono state riscontrate contestualmente all'utilizzo, come algoritmo di scheduling dei task aperiodici, e dunque come tipologia di server, del *Polling Server*, del resto unica opzione messa a disposizione dalla libreria ESFree. D'altro canto, è altresì vero che l'utilizzo di questa tipologia di algoritmo (unitamente al Rate Monotonic per la gestione dei task periodici nel complesso) ben si adatta alla natura del problema, disponendo noi di un numero considerevole di task aperiodici da gestire congiuntamente a quelli periodici.

Si era pensato di introdurre un task atto alla disabilitazione della scheda slave da

parte del master, tuttavia l'introduzione di un ulteriore task avrebbe appesantito lo scheduler e si è deciso dunque di accorpare questa funzionalità nella fase di esecuzione anche perchè logicamente la disabilitazione dello slave fa comunque parte del range di attività compatibili con un task di esecuzione. Una volta creati e definita la totalità dei task, si è proceduto con il ripartirli nelle due schede in accordo alla natura logica e gerarchica delle schede stesse.

Parallelamente a queste considerazioni, si è voluto stabilire una netta gerarchia di subalternità tra le due schede, la quale vede lo Slave incapace di compiere alcuna azione di carattere "emergenziale", e fa si che egli non possa disabilitare il Master. Le capacità dello Slave, dunque, si limitano all'invio dei dati rilevati tramite la sensoristica ad esso connessa (temperatura, batteria, encoder) verso il Master che li utilizzerà all'interno della propria logica di controllo ed esecuzione, e all'esecuzione intesa come semplice gestione dei motori (ed accensione dei led). La divisione dei task fra le schede ha palesato la natura totalmente periodica dei task dello slave (in quanto non vi è nessuna soglia critica che ci permette di disabilitare il master) e dunque si è deciso di schedularli semplicemente con un algoritmo per la schedulazione di task puramente periodici (non misti) come Rate Monotonic.

Ampliando ulteriormente la disamina in merito al rapporto tra le due schede, è opportuno illustrare la necessità del sincronismo tra i due task di comunicazione presenti sulle due schede: va senza dire, il motivo di ciò risiede nella necessità di non generare stati inconsistenti e/o irraggiungibilità della controparte. In particolare, la problematica risiede nella difficoltà di gestione di un protocollo di comunicazione tra i due attori: se le due schede non fossero sincronizzate sulla fase di comunicazione, una scheda potrebbe attendere di ricevere messaggi ma non ricevendoli entro un certo tempo si ridurrebbe a considerare defunto l'altro attore, o comunque irraggiungibile/inaffidabile.

Sarebbe stato possibile prevedere di attendere al più un ciclo di controllo dell'altro attore prima di credere quest'ultimo irraggiungibile, ma ciò avrebbe comportato un notevole dispendio di tempo. Conseguentemente, si è preferito far avvenire le due fasi di comunicazione in una finestra temporale verificantesi in maniera periodica ad istanti prefissati e fissi. La soluzione adottata è stata, in aggiunta a quanto detto, di fare in modo che lo Slave (sebbene in grado di svolgere i task precedenti alla fase di comunicazione più in fretta rispetto al Master) attenda comunque l'inizio della finestra temporale atta alla comunicazione prima di trasmettere i propri dati. Per poter agevolare ciò, i due cicli di controllo sono di medesima durata (iperperiodi uguali).

Sempre nel merito della coordinazione tra più attori, sono stati impiegati anche elementi di sincronizzazione e gestione degli accessi alle risorse: nel dettaglio, sono stati usati mutex per aiutare sia l'utilizzo della printf (funzione bloccante), utilizzata contestualmente alle stampe all'interno del circuito `VERBOSE` sia per coordinare l'utilizzo della struttura dati contenente le informazioni circa la sensoristica. In particolare, si è voluto, nonostante la marcata sequenzialità dei task, dotarsi di maggio protezione in merito all'utilizzo della variabile

`Global_Struct.Control_var` (sulla quale vengono effettuate scritture all'interno del task ultrasuono e lettura all'interno del task *CheckDanger*).

```
struct Struct_s{

    float Distance;
    int64_t Encoder_PastReading;
    int64_t Encoder_Steps;
    int16_t Battery_Sensor;
    int16_t PSX[9];
    int16_t Accelerometer;
    int16_t Temperature_Sensor;
    int8_t Control_Var;
};
```

Figure 3: Struttura contenente dati utili ai sensori.

```
    /* Initializes the mutex */
    MutexHandle = osMutexNew(&Mutex_attributes);
/* Mutex used for checking the accessing of the Control_Var attribute of the Global Struct */
osMutexId_t MutexHandle;
const osMutexAttr_t Mutex_attributes = {
  .name = "Mutex"
};
 /*
    As previously said, checks the mutex before trying to edit the value.
 */
 osMutexAcquire(MutexHandle, osWaitForever);
 if(Global_Struct.Distance < CRITICAL_DISTANCE)
    Global_Struct.Control_Var = 1;
 osMutexRelease(MutexHandle);
```

Figure 4: Esempi di utilizzo del mutex.

In ultima analsi, si vuole introdurre un aspetto che verrà approfondito successivamente: contemporaneamente alla progettazione, si sono effettuate le dovute analisi di schedulabilità intorno agli algoritmi di Rate Monotonic e Polling server utilizzati. Sebbene non si siano riscontrati impedimenti in merito al primo, col secondo non è stato possibile affermare altrettanto: una schedulazione con un budget del Polling server di 22ms e un periodo dello stesso di 50ms, avrebbe consentito di verificare il test di schedulabilità e ottenere una schedulazione **certamente** perseguibile. Per quanto idilliaco, ciò si è reso impossibile a causa di riscontrate incoerenze della libreria, in particolare in merito alla funzione adibita alla gestione del superamento dei WCET (variabili contenenti il tempo da attendere prima che un processo potesse riprendere la propria esecuzione venivano settate e non più utilizzate).

Nella fattispecie, il malfunzionamento del gestore sopracitato faceva in modo

che, per quanto si sforasse debitamente (con appositi valori di test) il WCET, il task non riprendesse alla successiva soglia di ripristino del budget ma riprendesse immediatamente.

```
#if( schedUSE_TIMING_ERROR_DETECTION_EXECUTION_TIME == 1 )

    /* Called if a periodic task has exceeded it's worst-case execution time.
     * The periodic task is blocked until next period. A context switch to
     * the scheduler task occur to block the periodic task. */
    static void prvExecTimeExceedHook( TickType_t xTickCount, SchedTCB_t *pxCurrentTask )
    {
        printf( "\r\nworst case execution time exceeded! %s %d %d\r\n", pxCurrentTask->pcName, pxCurrentTask->xExecTime, xTickCount );

        pxCurrentTask->xMaxExecTimeExceeded = pdTRUE;
        /* Is not suspended yet, but will be suspended by the scheduler later. */
        pxCurrentTask->xSuspended = pdTRUE;
        pxCurrentTask->xAbsoluteUnblockTime = pxCurrentTask->xLastWakeTime + pxCurrentTask->xPeriod;
        pxCurrentTask->xExecTime = 0;
        #if( schedUSE_POLLING_SERVER == 1)
            if( pdTRUE == pxCurrentTask->xIsPeriodicServer )
            {
                pxCurrentTask->xAbsoluteDeadline = pxCurrentTask->xAbsoluteUnblockTime + pxCurrentTask->xRelativeDeadline;
                #if( schedEDF_EFFICIENT == 1 )
                    listSET_LIST_ITEM_VALUE( &pxCurrentTask->xTCBListItem, pxCurrentTask->xAbsoluteDeadline );
                #endif /* schedEDF_EFFICIENT */
            }
        #endif /* schedUSE_POLLING_SERVER */
        BaseType_t xHigherPriorityTaskWoken;
        vTaskNotifyGiveFromISR( xSchedulerHandle, &xHigherPriorityTaskWoken );
        schedYIELD_FROM_ISR( xHigherPriorityTaskWoken );
    }

#endif /* schedUSE_TIMING_ERROR_DETECTION_EXECUTION_TIME */
```

Figure 5: Funzione della libreria ESFree da attenzionare.

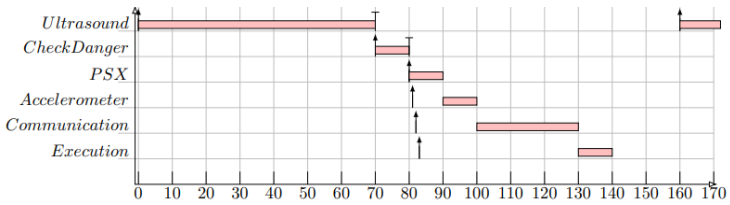

Figure 6: Schedulazione sul master senza anomalie

Sopra viene riportata la logica di schedulazione dei task del master. Sono presenti i task con relative deadline, tempi di arrivo e tempo di esecuzione (WCET).

- Freccia in su: tempo di arrivo.
- T in coda all'esecuzione di un task: deadline.

Per quanto riguarda i task:

- Ultrasound: viene attivato con fase 0, viene eseguito fino a ms 70.
- CheckDanger: arriva a ms 70 e dura 10ms. Ha il compito di controllare se sono successe anomalie; se no, schedula i task aperiodici.
- PSX: la lettura del controller avviene da ms 80 ad ms 90.
- Accelerometro: la lettura dell'accelerometro avviene da ms 90 ad ms 100.
- Communication: la comunicazione si svolge in una finestra da ms 100 a 130.

- Execution: l'esecuzione si svolge da ms 130 ad ms 140.

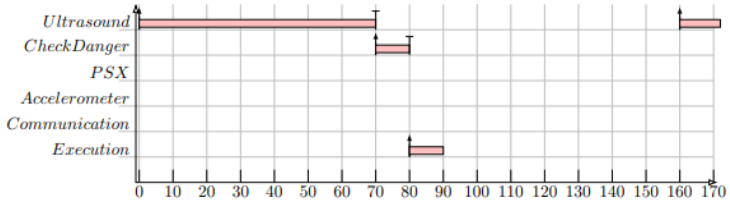

Figure 7: Schedulazione sul master con anomalie

Sopra viene riportata la logica di schedulazione dei task del master in caso di anomalia. Sono presenti i task con relative deadline, tempi di arrivo e tempo di esecuzione (WCET).

- Freccia in su: tempo di arrivo.
- T in coda all'esecuzione di un task: deadline.

Per quanto riguarda i task:

- Ultrasound: viene attivato con fase 0, viene eseguito fino a ms 70.
- CheckDanger: arriva a ms 70 e dura 10ms. In questo caso, rilevando l'anomalia, lancia soltanto il task Execution.
- Execution: l'esecuzione si svolge da ms 80 ad ms 90, fermando il veicolo. A partire da ms 160 si ricontrolla la presenza dell'ostacolo.

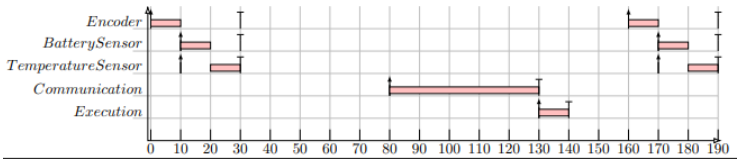

Figure 8: Schedulazione sullo slave.

Sopra viene riportata la logica di schedulazione dei task dello slave. Sono presenti i task con relative deadline, tempi di arrivo e tempo di esecuzione (WCET). Sono presenti 5 task periodici. Per poter decidere l'ordine di schedulazione viene considerata la fase del task.

- Freccia in su: tempo di arrivo.
- T in coda all'esecuzione di un task: deadline.

Per quanto riguarda i task:

- Encoder: necessariamente il primo per questioni di suo funzionamento; deve leggere sempre a cadenza costante per garantire la correttezza della lettura. Durata è ms 10.

- BatterySensor: da ms 10 a ms 20 legge la percentuale attuale di batteria.
- Temperature Sensor: da ms 20 a ms 30.
- Communication: E' presente un gap di ms 50 dal momento che lo slave, avendo finito i propri task da svolgere prima della comunicazione, attenderà il master.
- Execution: da ms 130 a ms 140.

Di seguito vengono inoltre proposti i ***Sequence Diagram***.

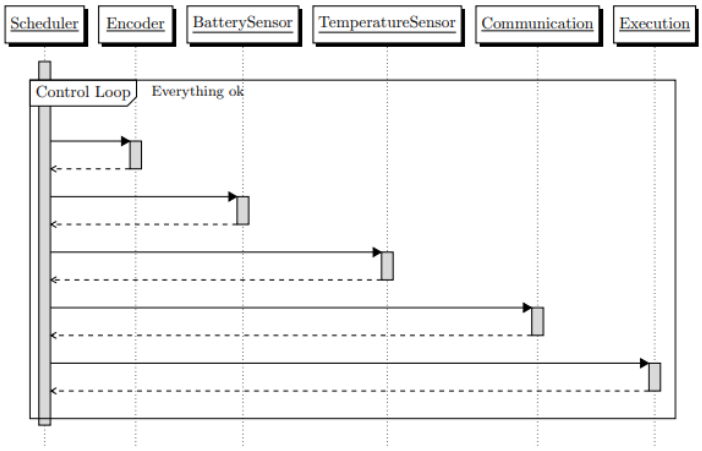

Figure 9: Sequence diagram per lo slave

.

Nel diagramma di sequenza per lo slave, viene rappresentato lo scheduler come sempre attivo (colonna sempre piena). Lo scheduler farà in modo che vengano eseguiti i task in tale ordine: Encoder, Sensory Battery e Temperatura, Comunicazione e Esecuzione.

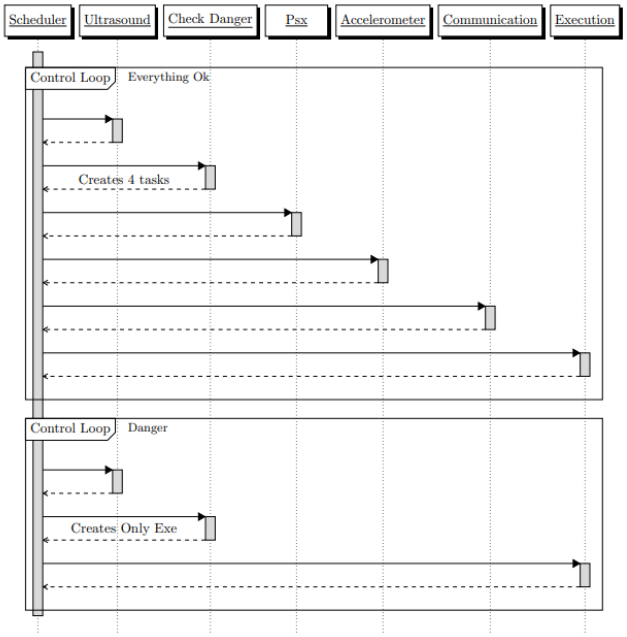

Figure 10: Sequence diagram per il master.

Sono riportati due cicli di controllo: uno dove si verificano anomalie ed uno in cui la schedulazione procede senza intoppi; il flusso cambia a seconda dei due casi. Nel primo ciclo di controllo si nota che dopo aver fatto l'ultrasound che non rileva l'ostacolo, il check danger non rileva problemi e quindi lancia i quattro tasks aperiodici per la normale esecuzione delle attività di locomozione del rover (è bene ricordare che, comunque, durante la comunicazione può sorgere l'emergenza). Nel secondo ciclo, l'ultrasound rileva un ostacolo, il CheckDanger quindi lancia direttamente l'Execution per avviare la procedura di arresto.

3 Implementazione

Questo capitolo è dedicato all'analisi dettagliata dell'implementazione del sistema di schedulazione dei task per il funzionamento ottimale del rover. L'obiettivo principale è quello di fornire una visione chiara e completa del codice che regola il funzionamento del rover, mettendo in evidenza le strategie adottate per garantire l'efficienza e l'efficacia del sistema di schedulazione. Nel corso del capitolo, verranno presentati e discussi vari frammenti di codice, che rappresentano le componenti chiave del sistema di schedulazione e, allo stesso tempo, verranno anche mostrate le scelte che sono state fatte, non solo per rispettare le varie specifiche, ma anche per compensare alcuni limiti riscontrati nella libreria di schedulazione stessa.

3.1 Codice in comune al master e allo slave

La maggior parte del codice per il master e lo slave differisce quasi completamente, in quanto il tipo di task, il loro numero e i compiti di ciascuno di essi cambia. Nonostante ciò è possibile denotare i seguenti elementi in comune.

3.1.1 Struct

```
struct Struct_s{

    float Distance;
    int64_t Encoder_PastReading;
    int64_t Encoder_Steps;
    int16_t Battery_Sensor;
    int16_t PSX[9];
    int16_t Accelerometer;
    int16_t Temperature_Sensor;
    int8_t Control_Var;
};
```

Si è deciso di includere le seguenti variabili utili a descrivere lo stato del sistema.

3.1.2 Policy di scheduling

```
/* The scheduling policy can be chosen from one of these. */
#define schedSCHEDULING_POLICY_MANUAL 0     /* The priority of tasks are set by the user. */
#define schedSCHEDULING_POLICY_RMS 1        /* Rate-monotonic scheduling */
#define schedSCHEDULING_POLICY_DMS 2        /* Deadline-monotonic scheduling */
#define schedSCHEDULING_POLICY_EDF 3        /* Earliest deadline first */

/* Configure scheduling policy by setting this define to the appropriate one. */
#define schedSCHEDULING_POLICY schedSCHEDULING_POLICY_RMS
```

Come detto nei precedenti paragrafi, lo schedulatore utilizzato è RMS, per entrambe le schede.

3.1.3 Error Detection

```
/* Set this define to 1 to enable Timing-Error-Detection for detecting tasks
 * that have missed their deadlines. Tasks that have missed their deadlines
 * will be deleted, recreated and restarted during next period. */
#define schedUSE_TIMING_ERROR_DETECTION_DEADLINE 1

/* Set this define to 1 to enable Timing-Error-Detection for detecting tasks
 * that have exceeded their worst-case execution time. Tasks that have exceeded
 * their worst-case execution time will be preempted until next period. */
#define schedUSE_TIMING_ERROR_DETECTION_EXECUTION_TIME 1
```

Si è deciso di lasciare attivi entrambi i meccanismi di detection degli errori. (deadline e WCET)

3.1.4 Accesso all'area di memoria condivisa Control_Var

```
osMutexAcquire(MutexHandle, osWaitForever);
if(Global_Struct.Distance < CRITICAL_DISTANCE)
    Global_Struct.Control_Var = 1;
osMutexRelease(MutexHandle);
```

Qualora il sistema dovesse accedere, che sia in lettura o scrittura, al valore contenuto all'interno della Control_Var, dovrà dapprima acquisire il mutex relativo.

3.2 Codice del Master

3.2.1 File Scheduler.h

```
/* Maximum number of periodic tasks that can be created. (Scheduler task is
 * not included, but Polling Server is included) */
#define schedMAX_NUMBER_OF_PERIODIC_TASKS 3

/* Set this define to 1 to enable aperiodic jobs. */
#define schedUSE_APERIODIC_JOBS 1
/* Set this define to 1 to enable sporadic jobs. */
#define schedUSE_SPORADIC_JOBS 0
```

Viene impostato il numero di task periodici pari a 3, ovvero:

- Ultrasound
- CheckDanger
- Polling Server

```
#if ( schedUSE_APERIODIC_JOBS == 1 )
    /* Maximum number of aperiodic jobs. */
    #define schedMAX_NUMBER_OF_APERIODIC_JOBS 4
#endif /* schedUSE_APERIODIC_JOBS */
```

Come deciso, vengono attivati solo i task aperiodici, impostando il numero di questi ultimi pari a 4, ovvero:

- PSX
- Accelerometer
- Communication
- Execution

```
#if( schedUSE_POLLING_SERVER == 1 )
    /* The period of the Polling Server. */
    #define schedPOLLING_SERVER_PERIOD pdMS_TO_TICKS( 80 )
    /* Deadline of Polling Server will only be used for setting priority if
     * scheduling policy is DMS or EDF. Polling Server will not be preempted
     * when exceeding deadline if Timing-Error-Detection for deadline is
     * enabled. */
    #define schedPOLLING_SERVER_DEADLINE pdMS_TO_TICKS( 2500 )
    /* Stack size of the Polling Server. */
    #define schedPOLLING_SERVER_STACK_SIZE 2000
    /* Execution budget of the Polling Server. */
    #define schedPOLLING_SERVER_MAX_EXECUTION_TIME pdMS_TO_TICKS( 80 )
    #if( schedSCHEDULING_POLICY == schedSCHEDULING_POLICY_MANUAL )
        /* Priority of the Polling Server if scheduling policy is manual. */
        #define schedPOLLING_SERVER_PRIORITY 5
    #endif /* schedSCHEDULING_POLICY_MANUAL */
#endif /* schedUSE_POLLING_SERVER */
```

Come detto nei precedenti paragrafi, si è deciso di scegliere i seguenti valori:

- Periodo: 80
- Max execution time: 80

Vista la policy RMS, la deadline del polling server non va impostata.

3.2.2 File freertos.c

```
vSchedulerPeriodicTaskCreate(Ultrasound, "Ultrasound", configMINIMAL_STACK_SIZE, NULL, 1, &xHandle1,
        pdMS_TO_TICKS(0),   //phase
        pdMS_TO_TICKS(160), //period
        pdMS_TO_TICKS(70),  //WCET
        pdMS_TO_TICKS(70)); //Deadline
vSchedulerPeriodicTaskCreate(Check_Danger, "Check_Danger", configMINIMAL_STACK_SIZE, NULL, 4, &xHandle4,
        pdMS_TO_TICKS(70),  //phase
        pdMS_TO_TICKS(160), //period
        pdMS_TO_TICKS(10),  //WCET
        pdMS_TO_TICKS(80)); //Deadline
```

Per la creazione dei task periodici (all'interno della MX_FREERTOS_Init()), ci si avvale della funzione vSchedulerPeriodicTaskCreate(), la cui chiamata ha i seguenti parametri:

- la funzione del task
- nome del task
- dimensione dello stack
- parametro da passare al task
- priorità del task, utilizzata solo se la policy di scheduling è impostata su "MANUAL"
- puntatore al task handle
- fase del task periodico

- periodo del task periodico
- WCET del task periodico
- Deadline del task periodico

In particolare, per il task **Ultrasound** si è deciso di impostare i seguenti valori:

- fase 0, è il primo task poiché è l'unico task critico ad essere schedulato
- periodo 160, pari al periodo del ciclo di controllo
- wcet 70, può al massimo impiegare 70 ms, incluso context switch
- deadline 70

Mentre per il task **CheckDanger** si è deciso di impostare:

- fase 70, così che possa venir eseguito subito dopo il task ultrasound,
- periodo 160, pari al periodo del ciclo di controllo
- wcet 70, può al massimo impiegare 70 ms, incluso context switch
- deadline 80

```
osMutexAcquire(MutexHandle, osWaitForever);
if(Global_Struct.Control_Var == 0){
    vSchedulerAperiodicJobCreate(Psx, "Psx", NULL, pdMS_TO_TICKS( 10 ) );
    vSchedulerAperiodicJobCreate(Accelerometer, "Accelerometer", NULL, pdMS_TO_TICKS( 10 ) );
    vSchedulerAperiodicJobCreate(Communication, "Communication", NULL, pdMS_TO_TICKS(30 ) );
    vSchedulerAperiodicJobCreate(Execution, "Execution", NULL, pdMS_TO_TICKS(10 ) );
}
else{
    vSchedulerAperiodicJobCreate(Execution, "Execution", NULL, pdMS_TO_TICKS(10 ) );
}
osMutexRelease(MutexHandle);
```

Il codice qui sopra riportato è la logica utilizzata all'interno del task CheckDanger(); quest'ultimo dapprima acquisisce il mutex utilizzato per l'accesso all'area di memoria condivisa, per poi andare a controllare il valore di control var. Control_var è il parametro utilizzato nella logica implementativa del codice per determinare se il sistema ha riscontrato valori critici dai sensori; in caso negativo, si procede alla normale esecuzione del ciclo di controllo; CheckDanger provvede quindi alla creazione dei 4 task aperiodici, ovvero Lettura del controller PSX, Lettura dell'accelerometro, Comunicazione tra le due schede ed infine attuazione, mettendoli all'interno di una coda. Mentre, in caso negativo CheckDanger provvederà alla sola creazione di esecuzione.

```
if(Global_Struct.Control_Var == 1){
    int8_t led = 1;
    int64_t velocita = 0;
    Global_Struct.Control_Var = 0;
}else{
    int8_t led = Global_Struct.PSX[4];
    int64_t velocita = Global_Struct.PSX[6];
}
```

Il codice qui sopra riportato è relativo alla gestione contenuta all'interno del task Execution(); come detto precedentemente, in base al valore di Control_var, il sistema agirà di conseguenza; nel caso in cui sia stato letto un valore critico dai sensori, la execution provvederà ad accendere i led, per visibilità, impostare la velocità dei motori pari a 0 e a resettare la control var. Nel caso in cui invece non sia stato letto nessun valore critico, la execution procederà ad eseguire i comandi imposti dall'utente attraverso il controller, in particolare imposterà sia i led che la velocità al valore desiderato.

3.3 Codice del Master

3.3.1 File Scheduler.h

```
/* Maximum number of periodic tasks that can be created. (Scheduler task is
 * not included, but Polling Server is included) */
#define schedMAX_NUMBER_OF_PERIODIC_TASKS 5
```

Qui sopra viene definito il numero di task periodici.

```
/* Set this define to 1 to enable aperiodic jobs. */
#define schedUSE_APERIODIC_JOBS 0
/* Set this define to 1 to enable sporadic jobs. */
#define schedUSE_SPORADIC_JOBS 0
```

Disattivato l'utilizzo di task aperiodici e sporadici.

```
#if( schedUSE_APERIODIC_JOBS == 1 || schedUSE_SPORADIC_JOBS == 1 )
    /* Enable Polling Server. */
    #define schedUSE_POLLING_SERVER 1
#else
    /* Disable Polling Server. */
    #define schedUSE_POLLING_SERVER 0
#endif /* schedUSE_APERIODIC_JOBS || schedUSE_SPORADIC_JOBS */
```

Viene così disabilitato anche l'utilizzo del polling server.

3.3.2 File freertos.c

```
vSchedulerPeriodicTaskCreate(Encoder, "Encoder", configMINIMAL_STACK_SIZE, NULL, 1, &xHandle2,
        pdMS_TO_TICKS(0),   //phase
        pdMS_TO_TICKS(160), //period
        pdMS_TO_TICKS(10),  //WCET
        pdMS_TO_TICKS(30)); //Deadline
vSchedulerPeriodicTaskCreate(Battery_Sensor, "Battery_Sensor", configMINIMAL_STACK_SIZE, NULL, 2, &xHandle1,
        pdMS_TO_TICKS(10),  //phase
        pdMS_TO_TICKS(160), //period
        pdMS_TO_TICKS(10),  //WCET
        pdMS_TO_TICKS(30)); //Deadline

vSchedulerPeriodicTaskCreate(Temperature_Sensor, "Temperature_Sensor", configMINIMAL_STACK_SIZE, NULL, 3, &xHandle3,
        pdMS_TO_TICKS(10),  //phase
        pdMS_TO_TICKS(160), //period
        pdMS_TO_TICKS(10),  //WCET
        pdMS_TO_TICKS(30)); //Deadline

vSchedulerPeriodicTaskCreate(Communication, "Communication", configMINIMAL_STACK_SIZE, NULL, 4, &xHandle4,
        pdMS_TO_TICKS(80),  //phase
        pdMS_TO_TICKS(160), //period
        pdMS_TO_TICKS(50),  //WCET
        pdMS_TO_TICKS(130));//Deadline
vSchedulerPeriodicTaskCreate(Execution, "Execution", configMINIMAL_STACK_SIZE, NULL, 5, &xHandle5,
        pdMS_TO_TICKS(130), //phase
        pdMS_TO_TICKS(160), //period
        pdMS_TO_TICKS(10),  //WCET
        pdMS_TO_TICKS(140));//Deadline
```

Come detto in precedenza per il master, ci si avvale della vSchedulerPeriodic-TaskCreate(), all'interno della MX_FREERTO_Init(); di seguito l'elenco delle scelte relative ai parametri passati alla funzione per ciascuno dei 5 task periodici implementati nello slave. Come si può evincere dal periodo impostato per ciascun task, lo slave condivide lo stesso periodo con il master, così che le schede possano essere sincronizzate temporalmente tra di loro.

- Encoder
  - Fase 0, parte come primo, vogliamo che questo accada affinchè il periodo dell'encoder possa essere sempre lo stesso, data la natura di quest'ultimo e la modalità di calcolo della velocità
  - Periodo 160, pari al periodo del ciclo di controllo; //verrà omesso ai prossimi tasks
  - WCET 10, può al massimo impiegare 10 ms, incluso context switch
  - Deadline 30
- Battery Sensor
  - Fase 10, verrà eseguito sequenzialmente subito dopo l'encoder
  - WCET 10, può al massimo impiegare 10 ms, incluso context switch
  - Deadline 30
- Temperature Sensor
  - Fase 10, così come Battery Sensor, verrà eseguito sequenzialmente subito dopo l'encoder; sarà lo schedulatore a decidere quale fra questi due eseguire prima

  - WCET 10, può al massimo impiegare 10 ms, incluso context switch
  - Deadline 30
- Communication
  - Fase 80, quest'ultima andrà implicitamente a determinare l'esecuzione di communication all'istante relativo 80, così che le due schede possano entrambe essere sincronizzate tra di loro, con la garanzia che lo slave inizi ad essere "in ascolto" con circa 20ms di anticipo rispetto alla prima comunicazione da parte del master
  - WCET 50, può al massimo impiegare 50 ms, incluso context switch
  - Deadline 130
- Execution
  - Fase 130, solo dopo la communication, lo slave potrà andare ad eseguire il task Execution
  - WCET 10, può al massimo impiegare 10 ms, incluso context switch
  - Deadline 140

```
if(pdMS_TO_TICKS(xTaskGetTickCount()) % 2 == 0)
    Global_Struct.Control_Var = 0;
else
    Global_Struct.Control_Var = 1;
```

Doveroso è sottolineare come la Control_var dello slave venga modificata solo dalla Communication(), ovvero nel caso in cui il master cessi di comunicare con lo slave.

```
if(Global_Struct.Control_Var == 1){         //Emergency Stop
    int8_t led = 1;
    int64_t velocita = 0;
    Global_Struct.Control_Var = 0;
}
```

Nel caso in cui, per qualsiasi motivo, il master interrompa la comunicazione con lo slave, quest'ultimo prenderà l'iniziativa di arrestare immediatamente il rover, evitando eventuali incidenti indesiderati.

4 Analisi schedulabilità

4.1 Schedulazione Slave

La verifica della schedulabilità concernente le attività dello Slave si è rivelata certamente più semplice rispetto a quella della sua controparte (task esclusiva-

mente periodici sullo Slave). Il test di schedulabilità ha riguardato solamente il Rate Monotonic, fornendo quanto di seguito:

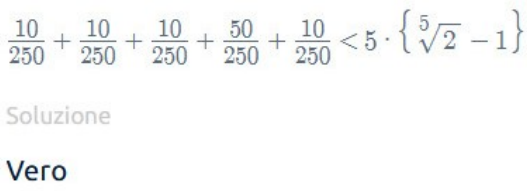

Figure 11: Fattore di utilizzo per Rate Monotonic.

Come è possibile notare, utilizzando un tempo per il ciclo di controllo di 250 ms e 10 ms per tutti i task (fuorchè la comunicazione, per la quale sono stati assegnati 50 ms), si è riusciti a rispettare la condizione in *Figura 2*, ottenendo un fattore di utilizzo del 36%.

4.2 Schedulazione Master

In prima battuta, per poter rimanere saldi nella volontà di ottenere un fattore di utilizzo che garantisse la schedulabilità, si è proceduto seguendo il test presente in *Figura 1*, ottenendo quanto di seguito:

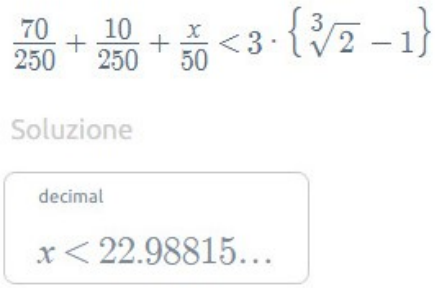

Figure 12: Fattore di utilizzo per Polling Server, iterazione 1.

Vale a dire, si è ottenuto una x, ovvero un budget per il polling server, di 22 ms, su un periodo di 50 ms.

In seconda battuta, invece, per poter ovviare alle problematiche insorte con la libreria ESFree, si è preferito reimpostare un tempo di ciclo di controllo più fedele alle reali tempistiche dei task in esecuzione sul master. In questo caso però, la capacità del Polling Server è stata posta uguale al periodo dello stesso, ovvero 80ms.

5 Conclusioni e indicazioni per l'esecuzione del codice

Concludendo, questo progetto ha richiesto l'impegno di progettare e implementare un sistema di scheduling real-time utilizzando FreeRTOS e la libreria ESFree. Attraverso un'attenta progettazione e implementazione, si è riuscito a gestire efficacemente l'esecuzione di task periodici e aperiodici, garantendo al contempo la sincronizzazione della comunicazione tra schede e la gestione efficiente delle risorse. Nonostante le sfide incontrate, il progetto ha dimostrato che con gli strumenti giusti e un approccio metodico, è possibile sviluppare un sistema di scheduling real-time robusto e affidabile. Tuttavia, ci sono ancora molte aree che potrebbero beneficiare di ulteriori ricerche e sviluppo. Ad esempio, l'implementazione di meccanismi di tolleranza ai guasti potrebbe migliorare ulteriormente la robustezza del sistema. Inoltre, l'ottimizzazione del sistema per ridurre ulteriormente il tempo di attesa dei task aperiodici potrebbe portare a miglioramenti significativi delle prestazioni.

Per quanto concerne l'esecuzione e la verifica del codice prodotto si è deciso, infine, di lasciare la possibilità di stampare i tempi di inizio e di fine task in modo da poter avere un riscontro visivo per una più completa comprensione. Nel caso in cui si vogliano quindi verificare le stampe è necessario impostare ad 1 il valore della define chiamata "VERBOSE", 0 altrimenti.