Università degli Studi di Firenze


Ingegneria Elettrica e dell'Automazione - curriculum
Automazione e Robotica

Elaborato di Software Engineering For Embedded Software

# Modellazione e implementazione di taskset in Linux RTAI

*Studente*
Marco Minarelli

Anno Accademico 2022/2023

# Indice

**1 Introduzione** **2**
1.1 Descrizione Hardware . . . . . 2
1.2 Linux RTAI . . . . . 2
1.3 Descrizione del lavoro . . . . . 2

**2 Premesse** **3**
2.1 Regressione Lineare . . . . . 3
2.2 Busy Sleep . . . . . 4
2.2.1 Implementazione . . . . . 4
2.3 Gestione Task in Linux RTAI . . . . . 4
2.4 Oris2.0 e SIRIO . . . . . 6

**3 Task set** **6**
3.1 Descrizione e modellazione con Sirio . . . . . 6
3.2 Implementazione . . . . . 6
3.3 Verifica del log . . . . . 9
3.4 Sviluppi futuri . . . . . 11
3.5 Conclusioni . . . . . 11

**Riferimenti bibliografici** **11**

# 1 Introduzione

## 1.1 Descrizione Hardware

L'hardware su cui verranno eseguite le prove e che verrà utilizzato come base per l'implementazione della busy_sleep() è un Intel NUC NUC10i3FNHN, che ha un processore Intel i3 di decima generazione con 4MB di cache e una frequenza fino a 4.10 GHz, fino a 10GB di RAM, e porte per le connessioni (USB, HDMI, Ethernet).

## 1.2 Linux RTAI

Linux RTAI (Real Time Application Interface) [MDP00] [But11] è un'estensione di Linux al mondo Real Time.
RTAI offre gli stessi servizi del kernel Linux aggiungendo però le peculiarità di un sistema operativo Real Time. Consiste in un interrupt dispatcher che cattura le interrupt delle periferiche e, se necessario, le reindirizza verso Linux. Grazie al concetto di Hardware Abstraction Layer (HAL), ottiene informazioni da Linux e cattura alcune funzioni fondamentali senza andare a modificare il kernel Linux.
RTAI permette di avere task sia soft- che hard-real time integrando lo scheduling di task RTAI, Linux kernel thread e task nello spazio utente. Fornisce inoltre un middleware basato sul concetto di Remote Procedure Call, che permette di utilizzare le API offerte in maniera distribuita.
Ad ogni task viene assegnato staticamente dall'utente una priorità e sono disponibili nativamente due algoritmi di scheduling: FIFO e Round Robin [1].

## 1.3 Descrizione del lavoro

Gli obiettivi di questo elaborato sono molteplici, sia di natura prettamente didattica che pratica:

1. in primis, si ha la necessità di esplorare le API offert da Linux RTAI e quindi di capire le funzioni che permettono la gestione dei task, dei semafori, delle mailboxes e degli algoritmi di scheduling offerti;

2. un altro obiettivo è l'implementazione di una funzione di busy sleep che permetta di simulare l'utilizzo della CPU per un certo periodo assegnato, così da poter eseguire dei test;

3. infine, un taskset verrà descritto come Rete di Petri tramite il tool Sirio del framework Oris2.0, e verrà implementato tramite le API di

[1] A partire dallo scheduler FIFO è possibile implementare gli algoritmi EDF e Rate Monotonic

RTAI e infine testato sull'hardware di riferimento. A partire dal log ottenuto si verificherà che i vincoli posti siano stati soddisfatti.

# 2 Premesse

## 2.1 Regressione Lineare

La regressione formalizza (e risolve) il problema di una relazione funzionale tra variabili misurate (nel nostro caso, il numero di iterazioni di un dato ciclo e il tempo di utilizzo della CPU).
Si supponga di conoscere alcune coppie di valori $(x_i, y_i)$ rilevati da un qualche esperimento, e di voler determinare una funzione matematica in grado di rappresentare il fenomeno in oggetto. Più nello specifico, si parla di regressione lineare quando si suppone che la relazione voluta sia rappresentabile con una retta. Il modello di regressione lineare è dato da

$$y_i = m \cdot x_i + q \qquad i = 1, \ldots, n \tag{1}$$

dove $y_i$ è la variabile dipendente, $x_i$ è la variabile indipendente, $q$ è l'intercetta della retta di regressione della popolazione mentre $m$ è il coefficiente angolare della retta di regressione della popolazione.
Si definisce

$$r_i = y_i - m \cdot x_i - q \tag{2}$$

residuo, cioé la distanza con segno tra il punto $(x_i, y_i)$ e il punto sulla retta $(x_i, m \cdot x_i + q)$. Si può pensare di annullare la somma dei residui, porre cioé

$$\sum_{i=1}^{n} y_i - m \sum_{i=1}^{n} x_i - nq = 0 \tag{3}$$

Questa equazione però non garantisce che i residui siano piccoli, in quanto residui di segno opposto tendono ad annullarsi nella somma.
È però vero che $0 \leq |r_i| \leq \sqrt{\sum_{j=1}^{n} r_j^2} \forall i = 1, \ldots, n$, quindi se la somma dei quadrati dei residui è minima, allora ogni residuo è vicino a zero.
Si vuole quindi minimizzare la quantità

$$S(m,q) = \sum_{i=1}^{n} (y_i - m \cdot x_i - q)^2 \Rightarrow \begin{cases} \sum_{i=1}^{n} y_i - m \sum_{i=1}^{n} x_i - nq = 0 \\ \sum_{i=1}^{n} x_i y_i - m \sum_{i=1}^{n} x_i^2 - q \sum_{i=1}^{n} x_i = 0 \end{cases} . \tag{4}$$

Questo sistema ammette una ed una sola soluzione:

$$m = \frac{n \sum xy - \sum x \sum y}{n \sum x^2 - (\sum x)^2} \tag{5}$$

$$q = \frac{\sum y - m \sum x}{n} \tag{6}$$

## 2.2 Busy Sleep

La busy_sleep() è una funzione che simula l'utilizzo della CPU per un certo periodo di tempo, passato come input.
La routine consiste in una prima parte dove, a partire dal tempo in ns, si ricava il numero di iterazioni, per poi eseguire un ciclo for per tale numero di iterazioni; l'iterazione del ciclo consiste in una qualche istruzione che effettivamente utilizzi il processore. Lo pseudocodice che rappresenta questo algoritmo è presentato in Algoritmo 1.

**Algoritmo 1** Struttura di busy_sleep()

**Require:** $ex\_time \geq 0$ l'execution time desiderato
**Ensure:** Che la CPU sia utilizzata per $ex\_time$
  $n\_cycles = f(ex\_time)$
  $var = 0$
  **for** $i = 0, \ldots, n\_cycles$ **do**
    $var = var + 1$
  **end for**

### 2.2.1 Implementazione

Per trovare la relazione che lega il tempo di esecuzione di un task con il numero di iterazioni del ciclo for sono state fatte due assunzioni:

1. Le due grandezze sono legate in modo lineare
2. Esiste sempre un overhead non nullo (quindi anche eseguendo 0 iterazioni otterremo un execution time diverso da zero)

A partire da queste due idee, il legame tra $ex\_time$ e $n\_cycles$ è stimato attraverso regressione lineare.
Per ottenere le coppie su cui eseguire la procedura vista in sottosezione 2.1 è stato implementata una procedura sperimentale presentata in Algoritmo 2.

Si è deciso di introdurre il numero di test in modo da poter fare la media per ogni valore di $n\_cycles$ su più test, così da ridurre l'errore commesso.

In Figura 1 possiamo vedere come la retta ottenuta approssimi l'andamento dei dati.

In Figura 2 si può invece vedere di quanto un task termina in anticipo rispetto al tempo desiderato di 20 ms, su 100 prove. Si noti che tale anticipo è nell'ordine delle migliaia di ns, mentre l'obiettivo sono decine di ms, dando quindi un errore trascurabile.

## 2.3 Gestione Task in Linux RTAI

Per scrivere un modulo che può essere caricato el kernel Linux, è necessario implementare due funzioni: `init_module` che fa da entry point e

**Algoritmo 2** Struttura dei test

**Require:** $n\_cycles$ il numero di iterazioni da eseguire
**Require:** $n\_test$ il numero di test da eseguire
  **for** $k = 1, \ldots, n\_test$ **do**
    $t_1 = time()$
    $var = 0$
    **for** $i = 0, \ldots, n\_cycles$ **do**
      $var = i + 1$
    **end for**
    $t_2 = time()$
    $(x_k, y_k) = (n\_cycles, t_2 - t_1)$
  **end for**

`clean_module` che come dice il nome "ripulisce" dopo che il modulo termina.
Linux RTAI fornisce una API in C, la quale permette di inserire nei moduli funzionalità real time. Un task viene definito tramite la funzione `rt_task_init`; si noti che questa funzione non esegue il task. La routine ha come input il descrittore del task, la funzione da eseguire, un intero passato dal task generante al generato, la dimensione dello stack, la priorità, il flag per l'utilizzo della fpu ed infine la funzione per gestire il segnale generato quando il thread entra nello stato di running.
La schedulazione in RTAI può essere periodica o aperiodica; si passa dall'una all'altra con le funzioni `rt_set_oneshot_mode` e `rt_set_periodic_mode`. Una volta settata la modalità, per rendere un processo periodico e farlo eseguire ad un certo istante si utilizza la funzione `rt_task_make_periodic`; per l'esecuzione di un task aperiodico invece si utilizza la `rt_task_resume`. Per la gestione della schedulazione si ha la routine `rt_task_wait`.
È presente una funzione `rt_busy_sleep`, la quale però si limita a controllare che un certo valore di un timer non sia stato superato.
Si può modificare la priorità del task sia in maniera manuale (con `rt_change_prio`) che in maniera automatica per implementare un protocollo di priority inheritance (con `rt_get_inher_prio`).
Si possono rimuovere task tramite la funzione `rt_task_delete`.
Per quanto riguarda lo scheduling, si può settare programmaticamente tramite `rt_set_sched_policy`.
Una mailbox si può creare tramite `rt_mbx_init` e vi si può accedere con `rt_mbx_send` e `rt_mbx_receive`. Esistono tre tipi di semafori: usati per registrare eventi, per gestire eventi binari e per gestire l'accesso a risorse tramite priority inheritance; tutti possono essere creati con la routine `rt_typed_sem_init` e poi si possono utilizzare tramite `rt_sem_wait` e `rt_sem_signal`.

### 2.4 Oris2.0 e SIRIO

ORIS [PBCV21] è un tool per l'analisi di reti di Petri stocastiche e timed. Permette di creare tramite GUI reti di Petri, di effettuare analisi non-deterministica delle suddette reti e di calcolare gli stati transienti di una rete di Petri stocastica.
Grazie alla libreria Sirio, i modelli gengerati dalla GUI possono essere sportati in linguaggio Java per poi venire agilmente analizzati.
Per verificare a correttezza del log, è stato usato un metodo chiamato isLogFeasible, il quale a partire da un file di testo contenente il log e la PTPN restituisce come output se il log contenuto è feasible per il modello fornito o meno. Tale log deve essere nella forma

```
[transition name]
[time to fire]
[transition name]
[time to fire]
...
```

Un esempio di log può essere visto in sottosezione 3.3.

## 3 Task set

### 3.1 Descrizione e modellazione con Sirio

In Figura 3 si può vedere il taskset modellato tramite la GUI di ORIS. Tale taskset è una riduzione del taskset di esempio presentato in [CRV11] e consiste in 3 task periodici, di periodi diversi, sincronizzati da 2 semafori.

I chunk di lavoro sono stati implementati come places seguiti da una transizione temporizzata (che ne indica la fine) che ha come tempi minimo e massimo rispettivamete il Best Case Execution Time e il Worst Case Execution Time del chunk stesso, mentre i due semafori sono stati modellati con i due place $mux_1$ e $mux_2$. Le transizioni $t_{10}$, $t_{20}$ e $t_{30}$ rappresentano i rilasci dei job; le transizioni $t_{21}$, $t_{25}$ e $t_{32}$ rappresentano dei priority boost necessari ad implementare una politica di priority ceiling e sono infatti effettuate prima di operazioni di wait; in maniera complementare, le transizioni $t_{23}$, $t_{27}$ e $t_{34}$ rappresentano siano delle prority deboost che il completamento del chunk di lavoro; infine le transizioni $t_{11}$, $t_{22}$ e $t_{33}$ modellano le operazioni di wait su un semaforo, mentre le $t_{12}$, $t_{23}$ e $t_{34}$ le signal.

### 3.2 Implementazione

Come specificato in sottosezione 2.3, il modulo ha una funzione che fa da entry point denominata init_module(), all'interno della quale sono stati inizializzati i task, i semafori, le code FIFO e viene calibrata la busy_sleep()

(si veda Listato 1).

Listato 1: Entry point del modulo

```
int init_module(void){
        int res;
        res = rtf_create(FIFO_ID_LOG, sizeof(log)*(LOG_VAL));
        if(res == ENODEV){
                rt_printk("FIFO␣ID␣too␣large");
                return -1;
        }
        if (res == ENOMEM){
                rt_printk("FIFO␣size␣too␣large");
                return -2;
        }



        calibrate_busysleep(MAX_VAL, STEP, 3);

        long long start1, start2, start3, time;

        /* IPC initialization */
        rt_typed_sem_init(&mux1, 1, BIN_SEM);
        rt_typed_sem_init(&mux2, 1, BIN_SEM);



        /* Task initialization */
        rt_task_init(&tsk1, (void*)tsk1_job, 0, 10000, 1, 0, 0)
            ↪ ;
        rt_task_init(&tsk2, (void*)tsk2_job, 0, 10000, 2, 0, 0)
            ↪ ;
        rt_task_init(&tsk3, (void*)tsk3_job, 0, 10000, 3, 0, 0)
            ↪ ;




        rt_set_periodic_mode();
        start_rt_timer(nano2count(500000));
        time = rt_get_cpu_time_ns();

        start1 = nano2count(time + 1000*MILLISEC + 80 *
```

```
            ↪ MILLISEC);
        start2 = nano2count(time + 1000*MILLISEC + 100 *
            ↪ MILLISEC);
        start3 = nano2count(time + 1000*MILLISEC + 120 *
            ↪ MILLISEC);


        log_len = 0;

        /* Sending the start-time */
        s = time + 1000*MILLISEC;
        res = rtf_create(FIFO_ID_VALUES, sizeof(long long));
        if(res == ENODEV){
                rt_printk("FIFO␣ID␣too␣large");
                return -1;
        }
        if (res == ENOMEM){
                rt_printk("FIFO␣size␣too␣large");
                return -2;
        }
        rtf_put(FIFO_ID_VALUES, &s, sizeof(long long) );

        /* Task execution */
        rt_task_make_periodic(&tsk1, start1, nano2count(80*
            ↪ MILLISEC));
        rt_task_make_periodic(&tsk2, start2, nano2count(100*
            ↪ MILLISEC));
        rt_task_make_periodic(&tsk3, start3, nano2count(120*
            ↪ MILLISEC));


        return 0;
}
```

Ad ogni task corrisponde una funzione, all'interno della quale viene svolta la logica implementativa. L'esecuzione dei chunk di lavoro è stata simulata tramite la funzione busy_sleep() ed è stato implementato anche una routine per ottenere un numero pseudo casuale tra un minimo e un massimo desiderati, così da poter simulare l'esecuzione di una funzione nel range $[bcet, wcet]$ specificato; si veda come esempio Listato 2

Listato 2: Funzione che implementa il Task 3

```
void tsk3_job(long arg){
        while(1){
```

```
                send_log(3, 30);

                busy_sleep(rand(1, 2) * MILLISEC);
                send_log(3, 31);

                rt_change_prio(rt_whoami(), 3);
                send_log(3, 32);

                rt_sem_wait(&mux1);
                send_log(3, 33);

                busy_sleep(rand(1, 2) * MILLISEC);
                rt_change_prio(rt_whoami(), 5);
                send_log(3, 34);

                rt_sem_signal(&mux1);

                rt_task_wait_period();
        }
}
```

### 3.3 Verifica del log

Il log avviene tramite una coda FIFO, sulla quale i task real time inviano messaggi e, successivamente, un task utente si occupa di leggere tali log e stamparli su file, in quanto quest'ultima operazione non è real-time compliant.
Gli eventi loggati corrispondono alle transizioni sopra descritte. Per motivi di praticità e di debug, sulla coda FIFO vengono condivisi

1. L'id del task che inviava il log (1, 2 o 3)
2. L'id della transizione (ad esempio, 10 27 0 32 )
3. Il tempo assoluto rispetto all'avvio di un timer real-time

Compito del processo real-time è calcolare (e scrivere su file) gli intertempi tra il firing time di una transizione e la successiva.
Per rendere meno impattanti gli errori, i tempi di rilascio dei task sono stati spostati in avanti o all'indietro per fare in modo che si riallineassero al loro tempo periodico di rilascio desiderato.
A titolo di esempio, in Listato 3 vengono riportate alcune linee del log ottenuto da una run.

Listato 3: Esempio di log

```
t10
80
t11
0
t12
2
t13
10
t14
0
t15
2
t20
6
t21
0
t22
0
t23
1
t24
10
t25
0
t26
0
t27
1
t30
8
t31
1
t32
0
t33
0
t34
1
t10
38
```

LogAnalyzer ha fornito come risposta che il log presentato è feasible.

### 3.4 Sviluppi futuri

Possibili sviluppi futuri si hanno andando a modificare una delle tre variabili in gioco:

- Cambiando l'hardware di riferimento si ottengono dei valori numericamente diversi (soprattutto riguardo la funzione busy_sleep);
- Modificando il task set (modificando i task già presenti o aggiungendone altri) potrebbe far vedere in pratica come un task set può non soddisfare la condizione sufficiente di schedulabilità ma essere poi effettivamente schedulabili;
- Modificando l'algoritmo di scheduling utilizzato si potrebbe modificare la sequenza di assegnamento del processore.

### 3.5 Conclusioni

La busy sleep è stata implementata per l'hardware proposto, il task set modellato grazie al tool SIRIO è stato scritto con le API di Linux RTAI ed il log ottenuto dalla sua esecuzione è stato dato in pasto al tool per la verifica. I risultati ottenuti sono stati positivi.

## Riferimenti bibliografici


[But11] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. 2011.

[CRV11] Laura Carnevali, Lorenzo Ridi, and Enrico Vicario. Putting preemptive time petri nets to work in a v-model sw life cycle. *IEEE Transactions on Software Engineering*, 37(6):826–844, 2011.

[MDP00] Paolo Mantegazza, EL Dozio, and Steve Papacharalambous. Rtai: Real time application interface. 2000.

[PBCV21] Marco Paolieri, Marco Biagi, Laura Carnevali, and Enrico Vicario. The ORIS Tool: Quantitative Evaluation of Non-Markovian Systems. *IEEE Trans. Software Eng.*, 47(6):1211–1225, 2021.

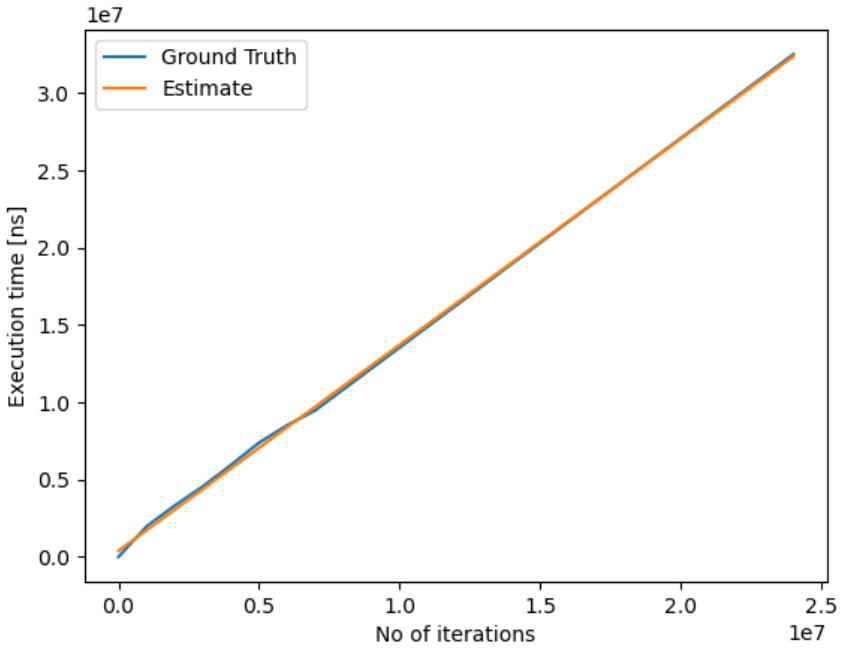


Figura 1: Confronto tra i dati e la retta interpolata

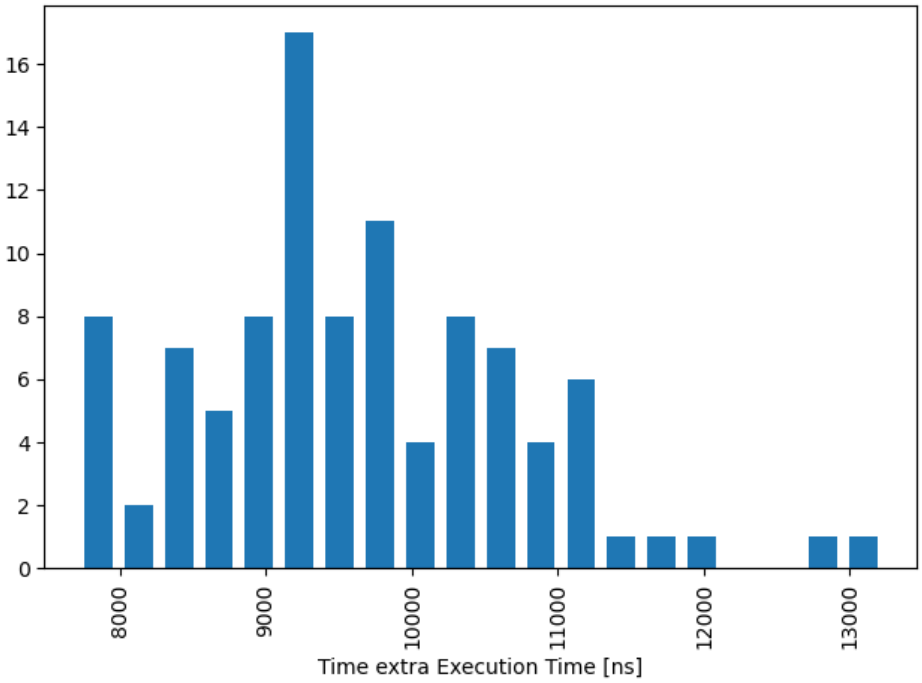


Figura 2: Anticipo della busy sleep rispetto ad un tempo di 20 millisecondi

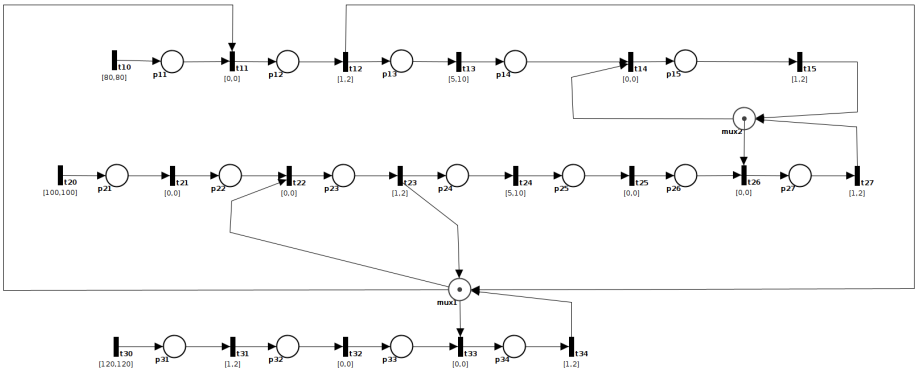


Figura 3: Modellazione del taskset