UNIVERSITÁ POLITECNICA DELLE MARCHE

FACOLTÁ DI INGEGNERIA

Corso di
Sistemi Operativi Dedicati

Introduzione e confronto tra FreeRTOS ed Erika Enterprise

Vitali Cristian

Anno Accademico 2020-2021

Indice

Capitolo 1

Introduzione

1.1 Obiettivo

Con la seguente documentazione si vuole presentare un'introduzione ai sistemi Real Time (RT) ed una panoramica sul funzionamento di due popolari Real Time Operating System (RTOS), FreeRTOS ed Erika Enterprise, cercando di evidenziarne le caratteristiche principali. Inoltre, poiché molti RTOS offrono soltanto semplici Fixed Priority Scheduling Policy, l'obiettivo sarà quello di studiare come algoritmi più complessi (e.g Earlier Deadline First) possano essere implementati al fine di creare sistemi più avanzati o avvicinare gli studenti, che hanno appena appreso concetti della moderna teoria RT, a questi due Operating Sistem (OS).

Si procederà come segue:

- nella prima parte si farà una panoramica sui sistemi operativi Real Time;
- nella seconda parte ci si concentrerà sul funzionamento di FreeRTOS, analizzando la struttura e come avviene la schedulazione dei task, terminando con una semplice implementazione dell'EDF su STM32 Nucleo Board.
- nella terza parte viene studiato il sistema operativo Erika Enterprise, il linguaggio OIL e lo standard OSEK/VDX. Si termina il capitolo con una breve configurazione della schedulazione EDF con Erika Enterprise.
- si chiuderà la documentazione con le conclusioni.

1.2 Sistemi Operativi Real Time

I sistemi real time sono definiti come quei sistemi in cui la correttezza del sistema non dipende solo dal risultato logico del calcolo, ma anche dal momento in cui i risultati vengono prodotti. Pertanto, è essenziale che i vincoli temporali (deadline) del sistema siano garantiti affinché non si verifichino danni o malfunzionamenti. I sistemi RT vengono utilizzati sia su microcontrollori molto semplici sia su sistemi altamente sofisticati. I task RT possono essere classificati in tre diverse categorie a seconda delle conseguenze di uno sforamento della deadline:

- Hard real-time: si hanno delle conseguenze dannose o tragiche sul sistema che si sta controllando;
- Firm real-time: generare l'output dopo la deadline crea un malfunzionamento, ma non causa alcun danno;
- Soft real-time: il superamento della deadline causa solamente un calo delle prestazioni, talvolta impercettibile.

Un sistema RT, quindi, deve soddisfare molte richieste entro un tempo limitato. L'importanza di quest'ultime può dipendere dalla loro natura (una richiesta relativa alla sicurezza può essere più importante di una semplice registrazione di un dato), dalla loro frequenza o dalla loro deadline. Pertanto, l'allocazione delle risorse di sistema deve essere gestita in modo tale da garantire il soddisfacimento di tutte le richieste in base alla loro importanza.

Generalmente, si utilizza uno schedulatore che implementa una determinata policy la quale deciderà come le risorse del sistema vengono allocate alle richieste.

Fondamentalmente, il problema è quello di determinare uno scheduling per l'esecuzione dei tasks al fine di soddisfare i requisiti temporali.

Per utilizzare un RTOS è necessario disporre di alcune informazioni relative ai tasks che si vogliono programmare, come: la deadline, il tempo di attivazione, il Worst Case Execution Time (WCET), il periodo o la priorità, in base a quale algoritmo di schedulazione si vuole utilizzare. La maggior parte di questi sistemi presume che molte delle informazioni siano disponibili a priori e ciò rende molto complessa la programmazione RT (un esempio è la nota difficoltà a stabilire a priori il WCET).

Un algoritmo di scheduling RT può essere classificato in base a diverse caratteristiche:

- esecuzione preemptive/non-preemptive: l'esecuzione preemptive di un task è quando esso viene temporaneamente interrotto, al fine di permettere l'esecuzione di un altro task a maggiore priorità. Il task interrotto viene in genere ripristinato una volta che quello a priorità maggiore ha terminato il proprio lavoro. Al contrario, l'esecuzione non-preemptive è quando sarà completato senza alcuna interruzione;

- task periodici/aperiodici/sporadici: i task periodici vengono eseguiti regolarmente a intervalli predefiniti. La maggior parte dell'elaborazione sensoriale è di natura periodica. I task aperiodici sono completamente irregolari mentre quelli sporadici vengono attivati in modo irregolare ma con un periodo minimo che non può essere anticipato;

- priorità statica/dinamica: nello scheduling basata su priorità, viene assegnata una priorità a ciascun task. L'assegnazione può essere eseguita alla creazione dei task, cioè in modo statico, o mentre il sistema è in esecuzione, quindi in modo dianamico.

Capitolo 2

FreeRTOS

FreeRTOS è un sistema operativo progettato per essere abbastanza leggero da funzionare su microcontrollore, anche se il suo utilizzo non è limitato a tale applicazione. FreeRTOS fornisce molteplici funzionalità e primitive che rendono molto agevole lo sviluppo di applicazioni RT. Tra le principali si hanno: scheduling RT, comunicazione tra task, primitive di temporizzazione e sincronizzazione.

Lo scheduler di FreeRTOS è preemtive e basato su priorità fissa. Quando viene inizializzato un task, gli viene assegnata una priorità e se più task hanno la stessa priorità, viene utilizzato il Round-Robin tra loro.

2.1 Struttura del Kernel

FreeRTOS fa della compattezza e semplicità le sue caratteristiche principali. Fondamentalmente il kernel è costituito da soli tre file:

- task.c: contiene tutte le primitive per la creazione e la gestione dei task. Alcuni esempi sono xtaskCreate(), vtaskDelay();

- queue.c: in questo file vengono definite le strutture utilizzate per la comunicazione e la sincronizzazione dei task e degli interrupt. Essi comunicano tra loro utilizzando code per lo scambio di messaggi mentre per la sincronizzazione e la condivisione di risorse critiche, utilizzano semafori e mutex.

- list.c: in questo file vengono definite la struttura dati delle liste e le primitive che permettono l'interazione con esse. Queste strutture vengono utilizzate sia dai task che dalle code.

2.2 Task

I task sono essenzialmente funzioni, piccole operazioni a sé stanti, alla quale viene assegnata una priorità. Ad ogni istante il sistema operativo selezionerà il task che sarà elaborato, in base a quest'ultima. Ognuno di essi viene eseguito nel proprio contesto, indipendentemente da quello degli altri.

Ad ogni task, FreeRTOS associa una struttura dati chiamata task Control Block (TCB).

Il TCB contiene informazioni generali, tra cui:

- stack pointers:
  *pxStackpoint, puntatore all'inizio dello stack appartenente al task;
  *pxTopOfStackpoints, alla parte superiore corrente dello stack;
  *pxEndOfStack, che punta alla fine dello stack, utilizzato per il controllo dello stackoverflow;
- uxPriorityVariable: contiene la priorità del task;
- ListIteamObjects, xGenericListItem e xEventListItem: le liste Ready, Blocked, Suspended non contengono un semplice puntatore a TCB, ma un puntatore a un ListItemObject. L'utilizzo di elementi ListIteam garantisce alle liste di essere più intelligenti e di elaborare operazioni con una minore complessità computazionale;
- pctaskName: vettore di char contenente il nome del task.

Un task può esistere in uno dei seguenti stati:

- Running: è lo stato del task indicato dal puntatore *pcCurrentTCBsystem ed è quello associato al task che è attualmente in esecuzione;
- Ready: a questa lista appartengono quelli pronti, in attesa di essere schedulati;
- Blocked: i task in questa lista non possono essere schedulati perché in attesa di un evento esterno o di un evento temporale. Ad esempio, un task in esecuzione che chiama il metodo vtaskDelay() si bloccherà ed entrerà nello stato Bloccato, in attesa di essere risvegliato;
- Suspended: si può entrare o uscire da questo stato solo chiamando esplicitamente i metodi vtaskSuspend() e xtaskResume(). I task sospesi non sono disponibili per lo scheduling.

La struttura TCB non contiene una variabile che rappresenta lo stato. Sarà FreeRTOS che implicitamente glielo attribuirà inserendolo nella lista appropriata (Ready, Blocked e Suspended).

La creazione di un task avviene invocando il metodo xtaskCreate(), contenuto in task.c: ne crea uno con una priorità assegnata e lo aggiunge alla lista Ready. Da questo momento esso è in attesa di essere eseguito dallo scheduler.

Un task può raggiungere lo stato Blocked chiamando la funzione vtaskDelayUntil(). Tale funzione può essere utilizzata per implementare task periodici poiché definisce una frequenza che ne permetterà l'esecuzione ad intervalli regolari. vtaskDelayUntil(), sposta il task nella lista Blocked, dove attende un intervallo di tempo predeterminato prima di essere nuovamente spostato nella lista Ready. Il funzionamento di questi cambi di contesto saranno riassunti nel prossimo paragrafo.

2.3 Context Switch

Il Context Switch è una particolare operazione del sistema operativo che permette di conservare lo stato di un task, per poi essere ripreso in un secondo momento. Prevede la disattivazione del task in esecuzione e il salvataggio del contesto di esecuzione nel proprio stack, pronto per essere riattivato quando il task sarà eseguito nuovamente. Un task non sa quando verrà sospeso o riattivato dal sistema, esso continua il suo flusso di esecuzione come se non si fosse verificato alcun cambio di contesto.

Questa attività permette a più processi di condividere la CPU ed è una caratteristica essenziale per i sistemi operativi multitasking e Real time. La funzione per effetuare il salvataggio è portSAVECONTEXT() mentre quella per il ripristino è portRESTORECONTEXT().

2.4 System Tick

Come visto in precedenza, quando viene chiamata la funzione xtaskDelayUntil(), il task chiamante specifica un tempo dopo il quale è necessario risvegliarsi. Il tempo è misurato da FreeRTOS ulizzando una variabile di sistema detta tick.

Un tick corrisponde ad un intervallo temporale definito dall'utente. Allo scadere di questo intervallo viene attivata un'Interrupt Service Routine (ISR) che incrementa la variabile tick con elevata precisione, consentendo al kernel RT di misurare il tempo alla frequenza scelta.

Ogni volta che il numero di tick viene incrementato, il sistema operativo deve verificare se è il momento di riattivare un task. È possibile che un task risvegliato durante la ISR abbia una priorità superiore a quella del task interrotto. In tal caso, avverrà un cambio di contesto e quello appena risvegliato verrà eseguito immediatamente. Come si è detto precedentemente, un comportamento di questo genere è detto preemtive.

2.5 EDF con Libreria ESFree

Diversi RTOS commerciali, compreso FreeRTOS, offrono solo semplici criteri di scheduling a priorità fissa. Per rendere il sistema operativo utilizzabile a scopi didattici si vuole implementare un algoritmo a priorità dinamica basato su deadline: l'Earlier Dedline First (EDF).

Poiché FreeRTOS non offre questa funzionalità, per integrare questi concetti si utilizzerà una libreria esterna chiamata ESFree. Utilizzare una libreria esterna implica che il kernel di FreeRTOS non venga modificato mantenendo i livelli di affidabilità e sicurezza invariati. In effetti, la libreria di scheduling proposta viene eseguita come una normale applicazione. In questo modo è possibile decidere se fare uso delle funzionalità teoriche messe a disposizione da ESFree come EDF, RM, ecc. o attenersi alle API standard di FreeRTOS.

ESFree è costituito dal file scheduler.c e la sua intestazione scheduler.h. La libreria fornisce:

- task periodici con parametri aggiuntivi che non sono inclusi nel TCB di FreeRTOS come: Phase, Period, deadline, Worst Case Execution Time (WCET);
- Rilevamento e gestione degli errori dovuti al WCET superato;
- Rilevamento e gestione degli errori dovuti allo sforamento della deadline;
- Rate-Monotonic Scheduling (RMS) policy;
- deadline-Monotonic Scheduling (DMS) policy;
- EDF policy;
- Aperiodic jobs;
- Sporadic jobs;
- Polling server.

ESFree ha il proprio TCB esteso chiamato SchedTCB, che include informazioni aggiuntive per la gestione dei task periodici. Anche in questo caso non viene modificato il TCB originale, ma viene aggiunta una nuova struttura utilizzata soltanto dallo scheduler aggiuntivo di ESFree.

Di seguito ci si concentrerà sulla EDF scheduling policy. In questo caso, ESFree utilizza delle Sorted Linked List per la gestione degli SchedTCB relativi ai task implementati. Fare uso di Sorted Linked list permette di ordinare i task in modo efficiente e in base alla propria deadline.

Lo scheduler di ESFree è visto da FreeRTOS come un task a massima priorità e la sua funzione include l'esecuzione dell'EDF e il rilevamento degli errori di temporizzazione.

Questa libreria fa uso delle notifiche dei task per gestire il blocco e il rilascio dello scheduler ESFree. La notifica dei task è una funzionalità che in alcuni casi può sostituire semafori e code ed è più efficiente. La libreria utilizza la funzionalità di tracciamento trace macros di FreeRTOS, per essere avvisati quando i task diventano Ready, Blocked o Suspended.

Come anticipato, lo sceduler di FreeRTOS continua a funzionare normalmente, ma si usano solo due priorità di esso per task periodici: Running e not-Running. A tutti i task viene inizialmente assegnata la priorità not-Running. Quando viene attivato lo scheduler ESFree, esso si assicura che il task periodico con la minore deadline assoluta in stato Ready abbia la priorità Running, ogni volta che:

- un task entra in stato Bloccato;
- un task si auto-sospende;
- un task entra in stato Ready mentre non ci sono altri task nello stato Running;
- un task con deadline assoluta minore del task in Running diventa Ready.

I task aperiodici e sporadici sono implementati come strutture dati chiamate rispettivamente Aperiodic Job Control Block (AJCB) e Sporadic Job Control Block (SJCB). Una volta creati sono inseriti in due code FIFO diverse e viene utilizzato il Polling Server per eseguirli. La struttura dati SJCB contiene la deadline a differenza di AJCB.

Il Polling Server è implementato come un task periodico con le seguenti istruzioni:

```
1. Se la coda FIFO degli SJCB non è vuota, vai a 2), altrimenti vai a 3)
2. Eseguire la funzione del primo sporadic job in coda, poi tornare a 1)
3. Se la coda FIFO degli AJCB non è vuota, vai a 4), altrimenti return
4. Eseguire la funzione del primo task aperiodico in coda, poi tornare a 1)
```

ESFree fornisce due tipi di rilevamento e gestione degli errori di temporizzazione: sforamento WCET da parte di task periodici e sforamento della deadline assoluta. Il rilevamento del WCET superato è implementato all'interno della funzione tick ISR (invocata ad ogni incremento della variabile tick). La gestione di questo errore è implementata all'interno del task scheduler ESFree che viene attivato nel caso sia stato rilevato.

A differenza del WCET, il rilevamento della mancata deadline assoluta richiede la scansione di tutti i task periodici e dei job sporadici. Questo causa un sovraccarico maggiore rispetto al rilevamento WCET che deve solo controllare il

task periodico in esecuzione. Quindi sia il rilevamento che la gestione dell'errore della mancata deadline assoluta sono implementati all'interno della funzione dello scheduler di ESFree. Di conseguenza, il rilevamento e la gestione dell'errore relativo al WCET superato, viene effettuato con la frequenza di tick, mentre il rilevamento e la gestione dell'errore relativo alla deadline assoluta superata, viene effettuato secondo il periodo del task dello scheduler ESFree.

2.5.1 Strumenti utilizzati

La schedulazione EDF con FreeRTOS è stata implementata e testata su STM32 Nucleo Board. Il codice implementato prevede quattro task periodici chiamati t1, t2, t3, t4. Ognuno di essi effettua un ciclo for vuoto con diverso numero di iterazioni e pubblica alcune informazioni nel log che sarà mostrato successivamente.

Oltre ai task periodici, ne vengono generati quattro aperiodici e quattro sporadici. Quelli sporadici vengono accettati solo se superano un test di schedulabilità. Questi 8 nuovi task sono gestiti da un Polling Server considerato come quinto task periodico.

Come IDE per lo sviluppo del progetto è stato utilizzato STM32CubeIDE che è una piattaforma di sviluppo C/C++ basata su framework Eclipse che permette di generare e compilare il codice per microcontrollori STM32.

STM32CubeIDE integra STM32CubeMX, che permette di configurare il sistema operativo e le periferiche della STM32 board da Interfaccia grafica generando automaticamente tutto il codice necessario.

Visualizzare il log o effettuare il debug sull'Host PC, non è immediato. Un'opzione è usare il semihosting con STM32CubeIDE che, però, può essere estremamente lento e quindi non adatto allo scopo del progetto. L'altra opzione è l'utilizzo della porta seriale (UART). Piu precisamente la porta che si utilizza su STM32 Nucleo Board è la UART2 che fornisce una porta COM virtuale via USB.

Per trasmettere informazioni si possono utilizzare le funzioni della libreria HAL come ad esempio HAL_UART_Transmit, oppure usare le classiche funzioni printf e scanf.

In questo caso si è scelto di utilizzare le funzioni dello stdio.h, ma per far questo vanno sovrascritte alcune funzioni utilizzate da esse, per indirizzare l'input e l'output sulla seriale.

Il codice è stato preso da Mastering STM32 di Carmine Noviello. Si vedrà di seguito i passaggi che sono stati fatti:

Il primo passo è escludere dal build il file syscalls.c che definisce molte delle stesse funzioni che vanno realizzate o sovrascritte. Quindi clik con il tasto destro sul file syscalls.c e poi Proprietà. In C/C++ Build > Settings, va selezionato "Exclude resource from build". Successivamente è stato creato un nuovo file nella directory

"Inc" chiamato retarget.h. Il codice si trova in https://github.com/cnoviello/mastering-stm32/blob /master/nucleo-f030R8/system/include/retarget/retarget.h.

```
#ifndef _RETARGET_H__
#define _RETARGET_H__

#include "stm32l4xx_hal.h"
#include <sys / stat.h>

void RetargetInit (UART_HandleTypeDef * huart);
int _isatty (int fd);
int _write (int fd, char * ptr, int len);
int _close (int fd);
int _lseek (int fd, int ptr, int dir);
int _read (int fd, char * ptr, int len);
int _fstat (int fd, struct stat * st);

#endif // # ifndef _RETARGET_H__
```

Va creato un secondo file nella directory "Src" chiamato retarget.c contenente il seguente codice:

```
#include <_ansi.h>
#include <_syslist.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/times.h>
#include <limits.h>
#include <signal.h>
#include <../Inc/retarget.h>
#include <stdint.h>
#include <stdio.h>

#if !defined(OS_USE_SEMIHOSTING)

#define STDIN_FILENO  0
#define STDOUT_FILENO 1
#define STDERR_FILENO 2

UART_HandleTypeDef *gHuart;

void RetargetInit(UART_HandleTypeDef *huart) {
  gHuart = huart;

  /* Disable I/O buffering for STDOUT stream, so that
   * chars are sent out as soon as they are printed. */
  setvbuf(stdout, NULL, _IONBF, 0);
}

int _isatty(int fd) {
  if (fd >= STDIN_FILENO && fd <= STDERR_FILENO)
    return 1;

  errno = EBADF;
  return 0;
}

int _write(int fd, char* ptr, int len) {
  HAL_StatusTypeDef hstatus;

  if (fd == STDOUT_FILENO || fd == STDERR_FILENO) {
    hstatus = HAL_UART_Transmit(gHuart, (uint8_t *) ptr, len, HAL_MAX_DELAY);
    if (hstatus == HAL_OK)
      return len;
```

```
43      else
44        return EIO;
45    }
46    errno = EBADF;
47    return -1;
48  }
49
50  int _close(int fd) {
51    if (fd >= STDIN_FILENO && fd <= STDERR_FILENO)
52      return 0;
53
54    errno = EBADF;
55    return -1;
56  }
57
58  int _lseek(int fd, int ptr, int dir) {
59    (void) fd;
60    (void) ptr;
61    (void) dir;
62
63    errno = EBADF;
64    return -1;
65  }
66
67  int _read(int fd, char* ptr, int len) {
68    HAL_StatusTypeDef hstatus;
69
70    if (fd == STDIN_FILENO) {
71      hstatus = HAL_UART_Receive(gHuart, (uint8_t *) ptr, 1, HAL_MAX_DELAY);
72      if (hstatus == HAL_OK)
73        return 1;
74      else
75        return EIO;
76    }
77    errno = EBADF;
78    return -1;
79  }
80
81  int _fstat(int fd, struct stat* st) {
82    if (fd >= STDIN_FILENO && fd <= STDERR_FILENO) {
83      st->st_mode = S_IFCHR;
84      return 0;
85    }
86
87    errno = EBADF;
88    return 0;
89  }
90
91  #endif //#if !defined(OS_USE_SEMIHOSTING)
```

Per utilizzare printf e scanf è sufficiente inizializzare il sistema con la funzione retargetInit(&huart2) e includere le librerie stdio.h e retarget.h.

Per visualizzare o inviare le informazioni attraverso la seriale va aperto il terminale che si preferisce e connetterlo alla porta COM in cui è collegata la Nucleo. La velocità di trasmissione dovrà essere impostata su 115200, 8-N-1. Il terminale che è stato utilizzato è GTKTerm, un semplice emulatore grafico di porta seriale per Linux. Può essere utilizzato per comunicare con tutti i tipi di dispositivi con interfaccia seriale come, appunto, questo microcontrollore.

2.5.2 Log ottenuto

Il log creato contiene diverse informazioni:

- il task quando viene eseguito stampa la sua firma contenente il proprio nome e l'istante della propria esecuzione (es. per il task n: "TICK <istante evento>: Periodic <n> Execution"). Si è aggiunto anche una serie di "#" che hanno solo uno scopo grafico, per rendere l'esecuzione del task distinguibile dalle altre tipologie;
- si stampa un log quando un task viene preso in carico dalla CPU (task in) mostrando anche la deadline assoluta e il momento in cui si è risvegliato (entrata nella Ready List);
- si stampa un log quando il task termina la sua operazione (task out) ed entra nello stato blocked fino al prossimo risveglio. In questo caso è mostrato anche il tempo di esecuzione.

Nel log di Fig. 2.1 il test di garanzia dell'EDF viene calcolato a priori, quindi non avverranno sforamenti di deadline a patto che il WCET sia rispettato.

Nel log di Fig. 2.2 viene modificato il task numero tre affinché ci sia uno sforamento del WCET e della deadline per testare il rilevamento degli errori. Nel log è quindi possibile notare ad esempio al tick 243 che il WCET è stato superato ed è stato gestito l'errore con una stampa a video. Al tick 317 ad esempio si ha il primo superamento della deadline sempre dal task numero 3.

Figura 2.1: Log ottenuto eseguendo l'applicazione creata

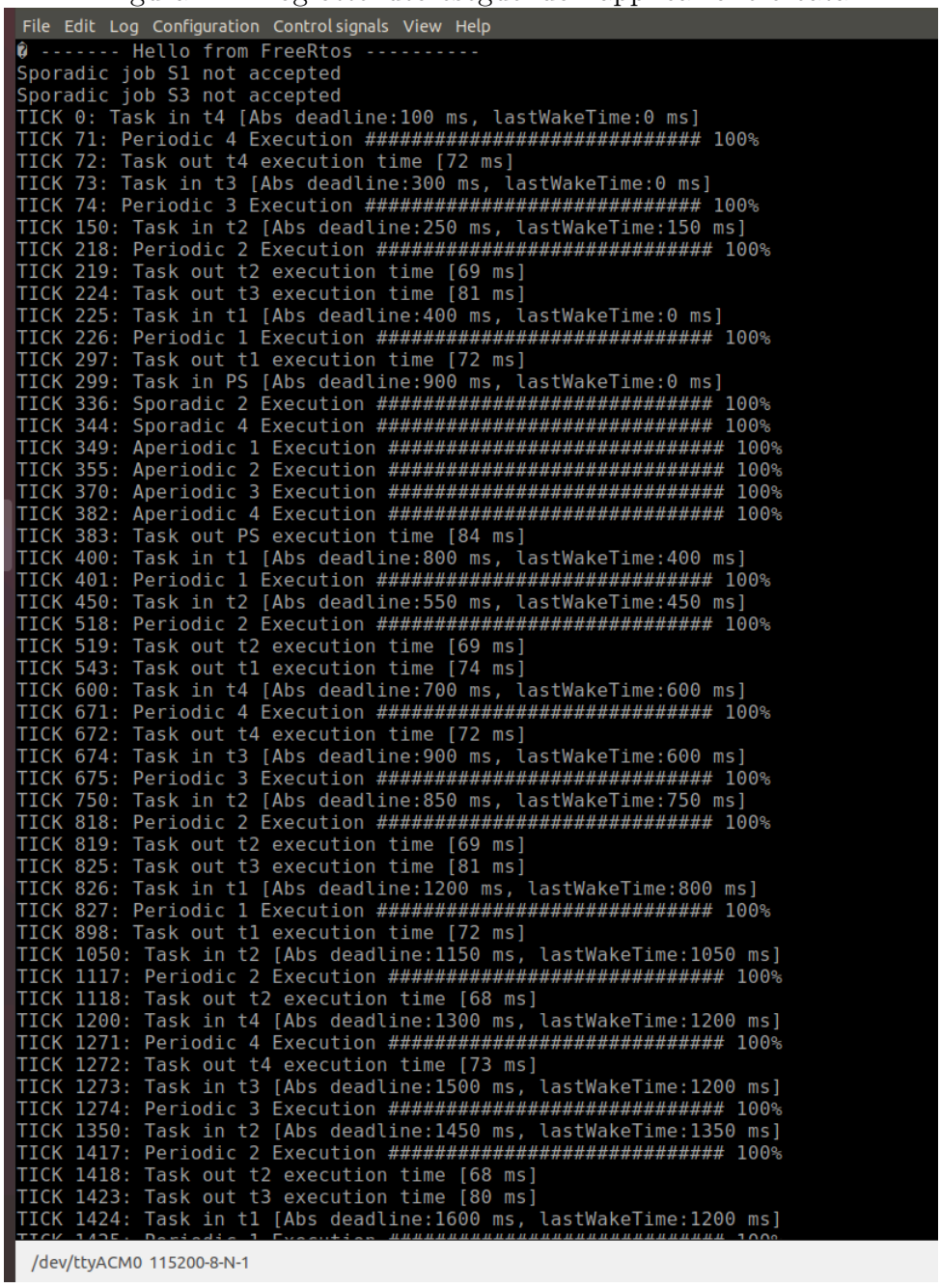
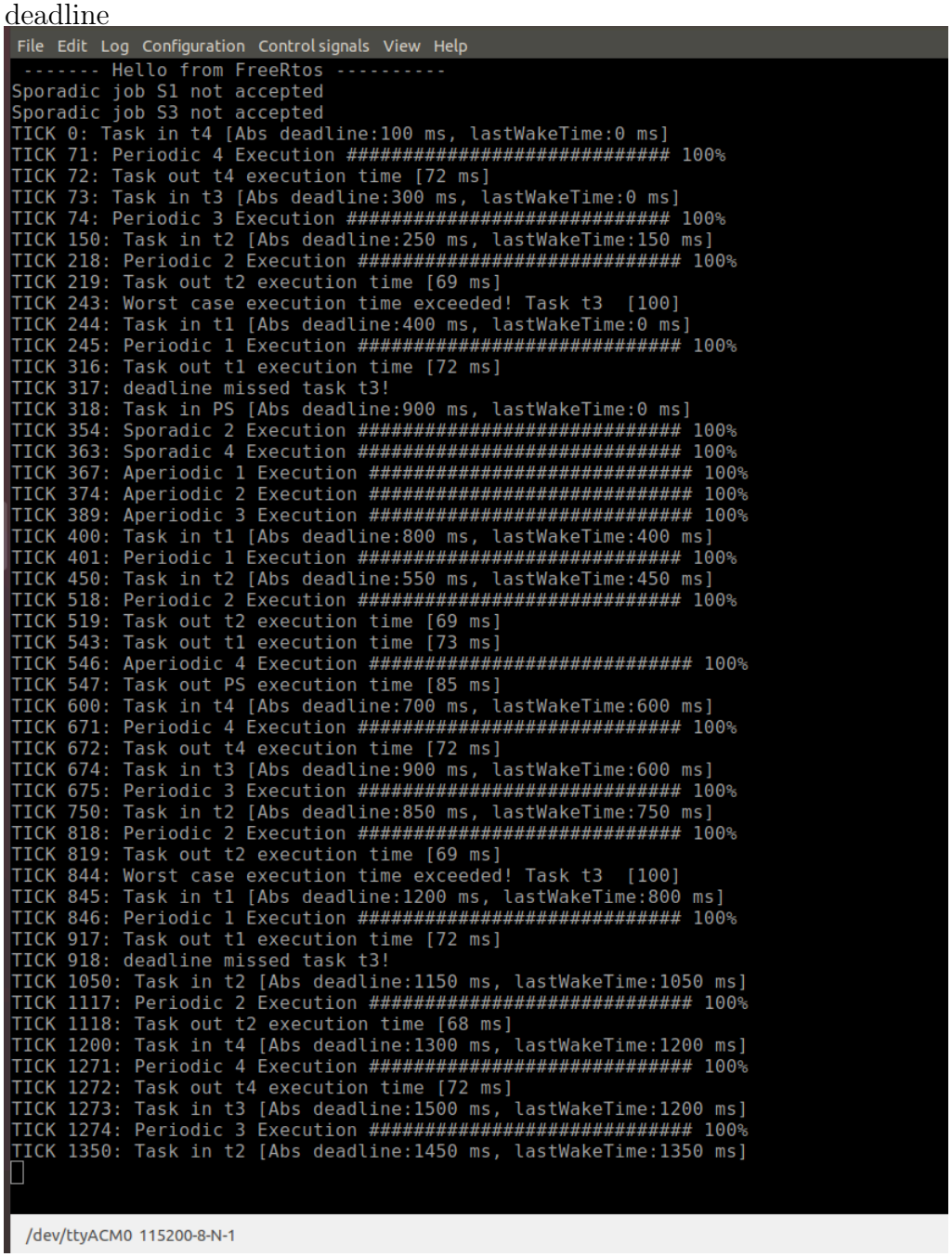


Figura 2.2: Log ottenuto modificando il task numero 3 affinché sfori il WCET e la deadline



Capitolo 3

Erika Enterprise

Erika Enterprise è un sistema operativo RT per microcontrollori che fornisce un ambiente di esecuzione multithreading semplice e compatto, con supporto di algoritmi di scheduling RT avanzati. Il kernel è stato sviluppato in modo tale da fornire un insieme minimale di primitive che possono essere utilizzate per implementare agevolmente un sistema multithreading. Erika E. è certificata OSEK/VDX e quindi implementa l'API basata su questo standard.

La sopra citata API offre diverse funzionalità come l'attivazione di task, la mutua esclusione, gli Alarms, i Counters, Semaphores e altre. Questi oggetti, OSEK/VDX e il relativo OSEK Implementation Language (OIL), saranno introdotti nei prossimi paragrafi.

Alcune delle caratteristiche fornite da Erika E., tipiche di un sistema RT, sono le seguenti:

- supporto per il multitasking sia di tipo preemptive che non-preemptive;
- supporto per algoritmi di schedulazione a priorità fissa;
- supporto per risorse condivise, con implementazione dell'Immediate Priority Ceiling (IPC);
- supporto per l'attivazione periodica di task per mezzo degli Alarms.

Altre caratteristiche innovative rispetto ad altri sistemi operativi RT sono:

- tecniche per lo stack sharing: tutti i task, sotto certe condizioni, possono condividere un unico stack;
- modelli di task one-shot al fine di ridurre l'utilizzo della RAM;
- supporto per schedulazione a priorità dinamica (es. EDF);
- supporto per la gestione centralizzata degli errori;

- supporto per funzioni hook prima e dopo ogni context switch.

Lo scopo di questo capitolo è soltanto quello di introdurre Erika E. e alcune delle sue API cercando di mostrare una panoramica a chi si avvicina a questo OS. Presso il sito web di Evidence è disponibile documentazione più dettagliata.

3.1 Standard OSEK/VDX

Negli anni, la quantità di microcontrollori utilizzati nell'industria automobilistica è cresciuta velocemente. Attualmente le auto hanno centinaia di sistemi real-time basati su tali componenti che necessitano di elevati requisiti di qualità, efficienza e sicurezza.

Al fine di migliorare l'efficienza dello sviluppo software, accelerare questo processo e mantenere gli alti requisiti di sicurezza, è stata fatta un'analisi che ha evidenziato quali sono i punti più impegnativi durante la progettazione di questi sistemi. Tale studio, portato avanti da Francia e Germania, ha evidenziato che la maggior parte dello sforzo è fatto nella fase di sviluppo e debug del sistema operativo e delle interfacce I/O.

Da questo processo di analisi sono nati due consorzi: il Vehicle Distributed eXecutive (VDX) realizzato dalle case automobilistiche francesi e l'Offene Systeme und deren Schnittstellen fur die Elektronik imKraftfahrzeug (OSEK) dalle case automobilistiche tedesche. Questi due consorzi sono cresciuti molto velocemente concentrando i loro sforzi sulla creazione di un API comune e di processi di sviluppo standardizzati per aumentare l'efficienza di sviluppo tramite riutilizzo di codice.

Successivamente, si sono fusi insieme dando vita allo standard OSEK/VDX chiamato anche standard ISO17356.

3.2 OIL: OSEK Implementation Language

Il consorzio OSEK/VDX ha messo a punto il linguaggio OIL (OSEK Implementation Language) come standard per la configurazione delle applicazioni. OIL viene utilizzato per la definizione statica di una serie di componenti e funzionalità da istanziare nell'applicazione, in modo da garantire la portabilità del software definita da OSEK/VDX.

Erika E. supporta appieno il linguaggio OIL per la configurazione di applicazioni RT. Per far fronte alla complessità che deriva dalla manipolazione di un file di configurazione scritto in linguaggio OIL, l'azienda Evidence ha sviluppato un tool di configurazione e profiling apposito chiamato RT-Druid. Esso permette di configurare un'applicazione in ogni sua componente, impostando i parametri in modo semplice attraverso una interfaccia visuale che automaticamente genera il codice di configurazione in linguaggio OIL.

Il tipico flusso di design di un'applicazione include la definizione di un file di configurazione in OIL, che definisce gli oggetti e i componenti che sono utilizzati all'interno dell'applicazione real-time. In questa fase, RT-Druid aiuta lo sviluppatore ad impostare i parametri dei singoli oggetti e provvede a:

- generare relativo file di configurazione in OIL;
- generare codice sorgente;
- generare make files richiesti per la compilazione dell'applicazione.

Grazie a questo processo di generazione del codice, lo sviluppatore dovrà concentrarsi soltanto sulla progettazione dei task e delle altre funzionalità dell'applicazione.

3.3 Task

In Erika E. esistono due diverse tipologie di task:

- Basic Tasks: è l'implementazione più semplice, esso inizia ad essere eseguito e termina. La CPU viene rilasciata quando il task termina la sua esecuzione, quando lo scheduler decide di scambiarlo con uno a priorità più alta oppure se deve essere eseguita una ISR. In questo caso i stati che possono essere assunti dal task sono Running, Ready o Suspended;
- Extended Tasks: si differenzia dal precedente poiché i Basic Tasks non possono sincronizzarsi tra loro né attendere altri eventi per poi continuare. Funzionalità come queste invece, vengono introdotte negli Extended Tasks; Viene quindi aggiunto lo stato Waiting tra quelli in cui un Task può trovarsi. Essi raggiungeranno tale stato dallo stato di Running e ci rimarranno fino a quando non si presenterà l'evento di cui è in attesa. Gestire lo stato di Waiting aumenta la complessità al sistema operativo, che deve utilizzare oggetti più complessi per la sincronizzazione, come ad esempio i semafori.

I task sono tipicamente implementati nella forma Basic e sono delle normali funzioni che eseguono il relativo codice e poi terminano. Ciascuna di queste esecuzioni è chiamata istanza del task. Al termine di un Basic task, lo stack relativo al task viene liberato.

Utilizzare i Basic task permette di implementare la condivisione dello stack tra i diversi task creati, così da ridurre l'utilizzo di RAM da parte dell'applicazione. In questo scenario le primitive bloccanti potranno essere usate soltanto dai task ai quali è stato assegnato uno stack privato.

La scheduling policy utilizzata di base in Erika E. è la Fixed Priority con Immediate Priority Ceiling per risolvere l'inversione di priorità.

Inoltre è possibile implementare:

- task Full Preemptive: task che può essere interrotto in qualsiasi istante da quello a priorità più elevata;
- task Non Preemptive: task che non può essere interrotto da nessun altro, ma solo da eventuali interrupt;
- task Mixed Preemptive: task che viene eseguito ad una priorità maggiore di quella assegnata quando viene inserito nella coda Ready (tecnica chiamata Preemption Thresholds). In tal modo il numero di preemption viene ridotto risparmiando considerevolmente la quantità di RAM utilizzata.

I task vengono attivati per mezzo della primitiva ActivateTask. Nel caso in cui un task in esecuzione o in attesa di essere eseguito, venga nuovamente attivato, l'attivazione viene comunque salvata e considerata come pendente (pending activation). Il numero massimo di pending activation viene specificata dall'utente.

Per utilizzare altri tipi di schedulazione dovranno essere configurate delle Classi di conformità, che si vedranno nel prossimo paragrafo.

3.4 Classi di conformità

Le classi di conformità mappano i servizi di cui necessita il sistema. In questo modo il sistema operativo risulta scalabile: in base alle esigenze si utilizzeranno classi diverse che consentono di soddisfare i requisiti dell'applicazione che si vuole sviluppare;

Di base sono fornite 4 diverse classi di conformità:

- BCC1 e BCC2 consentono al sistema operativo di supportare solo i Basic Task;
- ECC1 ed ECC2 che supportano solo Extended Tasks.

Le classi di conformità che terminano con uno (BCC1, ECC1), non possono memorizzare le pending activation. Al contrario, quelle che terminano con due (BCC2, ECC2), possono farlo e la capacità massima di attivazioni è specificata nel campo ACTIVATION, all'interno del campo Task nel file OIL.

Il numero di diverse priorità di task consentite in BCC2 è 8 mentre in ECC2 fino a 16. Utilizzando classi di conformità che terminano con uno, possono esistere n priorità diverse, dove n è il parallelismo permesso dall'architettura del microcontrollore.

Oltre a queste classi di conformità standard, ERIKA E. ne fornisce altre personalizzate come EDF o FRSH. La classe di conformità EDF include il supporto

per uno scheduler EDF dove ogni task ha una deadline relativa che viene calcolata nel momento in cui viene effettuata l'attivazione del task. La priorità maggiore l'avrà il task con la deadline assoluta minore.

Di seguito la modalità di configurazione della classe di conformità EDF:

```
KERNEL_TYPE = EDF {
    NESTED_IRQ = TRUE;
    TICK_TIME = "10,5 ns";
    REL_DEADLINES_IN_RAM = TRUE;
};
```

```
CPU test_application {
    OS myOS {
        ...
        KERNEL = EDF;
    };
    ...
    TASK myTask1 {
        PRIORITÀ = 3;
        REL_DEADLINE = "10ms";
        ...
    };
}
```

3.5 Events, Counters e Alarms

In questo paragrafo saranno presentati alcuni degli oggetti principali di questo sistema operativo.

Lo standard OSEK offre la possibilità di dichiarare e gestire gli eventi (Events). Ogni trigger dell'applicazione può generne di diversi: ad esempio la terminazione di un task. Nelle implementazioni preemptive del sistema operativo, essi attivano lo scheduler, al fine di valutare quale task deve essere eseguito considerando quel particolare evento.

Essi possono essere gestiti soltanto dalle classi di conformità dei task Extended, ECC1 ed ECC2. Quest'ultime, infatti, possono attendere un evento, impostare uno, e così via.

Tipicamente, gli eventi sono legati ai Counters: per implementare un'esecuzione periodica di un task, può essere sfruttato un contatore per scandire il tempo fino all'istante corrispondente al risveglio. Un Counter è semplicemente un valore intero che viene incrementato di un tick utilizzando la primitiva IncrementCounter.

Un Alarm è una notifica che viene collegata ad uno specifico Counter. Il collegamento tra i due viene specificato al momento della compilazione all'interno del file di configurazione OIL. Un Alarm può essere impostato per essere generato ad uno specifico valore di tick utilizzando le primitive SetRelAlarm e SetAbsAlarm e possono essere cancellati per mezzo della primitiva CancelAlarm.

Erika E. supporta un meccanismo di notifica basato su Counters e Alarms. Quando un Alarm viene attivato, ha luogo una notifica che può essere impostata per effettuare una delle seguenti azioni:

- task activation : in questo caso viene attivato un task quando il corrispondente Alarm viene generato;
- alarm callback: in questo caso viene chiamata semplicemente una funzione callback;
- impostare un evento.

I Counters, gli Alarms e le relative azioni di notifica sono specificate all'interno del file di configurazione OIL.

3.6 EDF in Erika Enterprise

Come visto in precedenza oltre alle classi di conformità OSEK/VDX classiche, Erika E. offre la classe EDF che rappresenta un'estensione allo standard OSEK/VDX. Per implementare una schedulazione Earlier Deadline First è sufficiente dichiararlo nel parametro KERNEL_TYPE all'interno del file di configurazione OIL. Quindi per configurare questa classe va definito:

```
1 KERNEL_TYPE = EDF {
2     NESTED_IRQ = TRUE;
3     TICK_TIME = "10,5 ns";
4     REL_DEADLINES_IN_RAM = TRUE;
5 };
```

Per il kernel EDF, è possibile specificare la lunghezza del tick in TICK_TIME. Il tick è l'unità di misura con cui viene calcolato il tempo. Il kernel EDF ha il parametro REL_DEADLINES_IN_RAM, che consente di specificare se le deadline relative debbano essere memorizzate nella RAM al posto della Flash, per consentirne la modifica in fase di esecuzione.

Il seguente esempio mostra come dichiarare un task, specificando il livello di priorità e la relativa deadline.

```
CPU test_application {
    OS myOS {
        ...
        KERNEL = EDF;
    };
    ...
    TASK myTask1 {
        PRIORITÀ = 3;
        REL_DEADLINE = "10ms";
        ...
    };
    ...
    TASK myTask2 {
        PRIORITÀ = 4;
        REL_DEADLINE = "20ms";
        ...
    };
    ...
}
```

Nel kernel EDF, il valore PRIORITÀ specifica il livello di preemtion del Task. Il valore viene utilizzato da RT-Druid come ordine relativo delle priorità e non come valore di priorità assoluta. Valori più alti corrispondono a priorità più alte.

L'attributo REL_DEADLINE specifica la Deadline relativa del task. Il valore può essere espresso in secondi (s), millisecondi (ms), microsecondi (us) o nanosecondi (ns).

Esso viene diviso per l'attributo TICK_TIME specificato all'interno dell'attributo KERNEL_TYPE, per ottenere il valore di tick che viene utilizzato dal programma.

Capitolo 4

Conclusioni

In questa documentazioni sono stati studiati due sistemi operativi real time con l'obiettivo di evidenziarne le caratteristiche essenziali e vedere come sia possibile utilizzare questi, per motivi didattici o per creare sistemi piu complessi, implementando algoritimi appartenti alla moderna teoria RT.

Si è visto come FreeRTOS abbia funzionalità minime per quanto riguarda la schedulazione, implementando di default soltanto una Fixed Priority. Per utilizzare uno schedulatore a priorità dinamica come l'EDF, è stata utilizzata una libreria esterna chiamata ESFree. Questo OS risulta ben documentato, ha esempi, guide e vanta di una community molto attiva con cui scambiare soluzioni e idee. Inoltre, FreeRTOS ha interi libri dedicati come quello utilizzato in questa relazione, Mastering STM32 di Carmine Noviello, nel quale viene analizzato nei minimi dettagli l'intero sistema oprativo FreeRTOS su STM32 Nucleo. Ciò permette uno sviluppo rapido e semplice anche per chi si è appena avvicinato al mondo del RT.

Erika Enterprise è un sistema operativo italiano e viene portato avanti dalla Evidence, azienda nata come spin-off del laboratorio RETIS dell'Istituto Superiore Sant'Anna di Pisa, ed è stato costruito sulla base della grande esperienza maturata sul kernel real time di ricerca SHARK. Esso ha funzionalità complesse nonostante il suo footprint sia molto basso. Vanta ad esempio dell'implementazione di base dell'algoritmo EDF e ha una modalità di gestione dello stack molto avanzata permettendo un bassissimo consumo della RAM. Come si è visto, permette a tutti i Task di condividere un unico stack in determinate condizioni, a differenza di FreeRTOS in cui è necessario uno stack separato per ogni Task (più RAM utilizzata e cambio di contesto più lento). Attualmente però, risulta piu' difficile accedere a guide, video o spiegazioni dettagliate e si hanno soltanto manuali pubblicati dell'azienda produttrice, generando una maggiore complessità per un principiante che vuole apprendere come avvengono determinati processi interni, come la schedulazione.

La principale differenza tra i due sistemi operativi è che Erika Enterprise è certificata OSEK/VDX, uno standard industriale riconosciuto nel settore automotive dagli anni novanta. E' da preferire quindi per progetti che necessitano di questa certificazione e in generale in tutto l'ambito automotive. Di seguito si cerca di riassumere e confrontare alcune caratteristiche:

- Standardizzazione API;
- Maturità;
- Hardware supportato;
- Opzioni di scheduling;
- Memoria utilizzata;
- Licenza.

OS	FreeRTOS	Erika Enterprise
Standardizzazione		
POSIX	parziale	no
OSEK/VDX	no	si
Maturità		
First release	2003	2000
Last release	2020	2019
Community	open source	open source
Hardware		
STM32 NUCLEO (Cortex-M4)	si	no
STM32 DISCOVERY (Cortex-M4)	si	si
NVIDIA JATSON TX1/TX2	no	si
Scheduling		
Priority-based	si	si
Round-Robin	si	no
Sporadic job Server	no	no
Rate Monotonic	no	no
Semaphore /Mutex	si	si
PIP	si	no
PCP	no	si
EDF	no	si
Memoria		
RAM	236B scheduler + 64B/task	67B Kernel Global Func. + 90B/task
ROM	5-10 kB	1-6 kB
Licenza	MIT e Commerciale	GPL