

## SCALETTA LEZIONE SERT 17.11.2022 (E08)

## 1 Implementazione di un tick periodico Usando circuito "timer, cioè un registro incrementato con una certa frequenza)

non posso  
ripartire da 0 ogni  
tot, deve fare  
"autoreload"

## 1.1 Leggere la documentazione ("ARM 335x TRM", ch. 20)

- 1.1.1 Registro TLDR: contiene il valore ricaricato dopo overflow
- 1.1.2 Registro TCLR: controllo del timer
- 1.1.3 Registro TTGR: per forzare overflow genera interruzione, con autoreload posso reimpostarlo a != 0)
- 1.1.4 Registri IRQ\_ENABLE\_SET/IRQ\_ENABLE\_CLR: abilita/disabilita generazione di IRQ su capture/overflow/match
- 1.1.5 IRQ\_STATUS: notifica ricezione dell'interruzione

## 1.2 Creare il file bbb\_timer.h: Più il tempo è accurato, meno tempo dedico ad altre operazioni. Ogni dispositivo che genera una interruzione, deve sapere che essa è stata vista!

MACRO per ricordare  
le scelte fatte:  
- '32768' è l'incremento  
eseguito ogni secondo.

- Tick frequency = 1000  
implica periodo di 1 ms

- IRQ 66 viene dal manuale.

- Timer0\_IRQ/32 sono il  
numero di registri  
contenenti tale linea.

fa op. enable  
fa op. clean  
"quello che corre"

provoca  
ricaricamento del  
registro

```
#define Timer0_Freq 32768 /* Hz */
#define HZ 1000 /* Tick frequency (Hz) */
#define TICK_TLDR (0xffffffffu- (Timer0_Freq/HZ)+1)
#define Timer0_IRQ 66
#define Timer0_IRQ_Bank (Timer0_IRQ/32)
#define Timer0_IRQ_Bit (Timer0_IRQ%32)
#define Timer0_IRQ_Mask (1u<<Timer0_IRQ_Bit)

#define DMTIMER0_BASE 0x44e05000

iomemdef(DMTIMER0_IRQSTATUS, DMTIMER0_BASE + 0x28);
iomemdef(DMTIMER0_IRQENABLE_SET, DMTIMER0_BASE + 0x2c);
iomemdef(DMTIMER0_IRQENABLE_CLR, DMTIMER0_BASE + 0x30);
iomemdef(DMTIMER0_TCLR, DMTIMER0_BASE + 0x38);
iomemdef(DMTIMER0_TLDR, DMTIMER0_BASE + 0x40);
iomemdef(DMTIMER0_TTGR, DMTIMER0_BASE + 0x44);

#define TCAR_IT_FLAG (1u << 2)
#define OVF_IT_FLAG (1u << 1)
#define MAT_IT_FLAG (1u << 0) } qui non voglio interruzioni.
```

L'operazione TLDR mi dà  
il valore da ricaricare  
automaticamente.  
La parte viola è legata  
all'overflow di 33 bit,  
quella verde è  
l'incremento  
per ogni ms.

- 1.2.1 L'idea è quella di generare un interrupt sull'evento "overflow" del contatore. A seguito dell'overflow il contatore viene ricaricato con il valore contenuto nel registro TLDR. Configurando opportunamente questo registro è possibile indurre un overflow con una certa frequenza, dunque generare interruzioni periodiche
- 1.2.2 Nel nostro caso TLDR viene caricato con il valore: (0xffffffffu- (Timer0\_Freq/HZ)+1)
- 1.2.2.1 Poiché ogni secondo il contatore viene incrementato di 32768, vuol dire che ogni millisecondo viene incrementato di 32768/1000 (== 32)
- 1.2.3 Aggiungere bbb\_timer.h in beagleboneblack.h

## 1.3 Scrivere la funzione init\_ticks() in tick.c

```
void init_ticks(void)
{
    irq_disable(); Quando programmo l'hardware, le interruzioni hw vanno disabilitate.
    if (register_isr(Timer0_IRQ, isr_tick) != 0) {
        irq_enable();
        puts("init_ticks: cannot register isr.\n");
        panic0();
    }
}
```

caso di errore,  
vado in panic0

genero  
interruzioni SOLO  
su overflow,  
prima disabilito  
tutto, poi attivo  
solo overflow.

Registro opera in  
auto reload

faccio passare  
del tempo per  
"assestarsi",  
poi con TTGR =  
1 ricarico veloce

```

iomem(DMTIMER0_TLDR) = TICK_TLDR;
{
    iomem(DMTIMER0_IRQENABLE_CLR) = TCAR_IT_FLAG | MAT_IT_FLAG;
    iomem(DMTIMER0_IRQENABLE_SET) = OVF_IT_FLAG;
    iomem(INTC_ILR_BASE + Timer0_IRQ) = 0x0; PRIORITA' MASSIMA
    iomem(INTC_MIR_CLEAR_BASE + 8 * Timer0_IRQ_Bank) =
        Timer0_IRQ_Mask); 8 = distanza registri ; IRQ_Bank = registro con bit di linea.
    iomem(DMTIMER0_TCLR) = 0x3; /* Auto-reload, start */
    loop_delay(10000);
    iomem(DMTIMER0_TTGR) = 1;
    irq_enable(); ← Alla fine devo sempre riattivare le interruzioni!
}

```

1.3.1 Aggiungere il prototipo di `init_ticks()` in `comm.h`

1.3.2 Invocare `init_ticks()` in `_init()`

1.4 Scrivere la funzione `isr_tick()` in `tick.c`: **INTERRUPT SERVICE ROUTINE**

```

volatile unsigned long ticks = 0;
static void isr_tick(void)
{
    iomem(DMTIMER0_IRQSTATUS) = OVF_IT_FLAG;
    ++ticks;
}

```

E' una funzione cruciale, poichè  
conferma di aver trattato  
l'interruzione. E' così descritta  
nel manuale. "++ticks" perchè  
non ottimizzo gli accessi, i ticks  
aumentano "sotto al naso",  
anche se nessuno li nota.

1.5 Aggiungere '`extern unsigned long ticks;`' a `comm.h`

1.6 Aggiungere la stampa del valore di ticks in `endless_led_blinking()`:

```

static void endless_led_blinking(void)
{
    int state = 1;

    for(;;) {
        loop_delay(10000000);
        leds_off_mask(0xf);
        leds_on_mask(state);
        state = (state+1) & 0xf;
        printf("Ticks: %u\n", ticks);
    }
}

```

1.7 Eseguire per verificare il corretto funzionamento

2 Funzione `mdelay()` basata sul tick periodico

2.1 Scrittura funzione in `delay.c`

```

inline void mdelay(unsigned long msec)
{
    unsigned int tckd = (msec*HZ+999)/1000;
    unsigned long expire = ticks + tckd;
    while (ticks < expire)
        cpu_wait_for_interrupt();
}

```

`mdelay` accetta "msec".  
In "tckd" abbiamo 999 per  
arrotondare all'intero  
superiore, e "tckd" è il  
valore del tick quando il  
ritardo scade.  
In "expire" ho quando  
scade: val. attuale + tick  
che devono passare.  
"cpu\_wait\_for\_interrupt()" spegne la CPU (caso uniprocessore) fino a prossima interruzione (ovvero devo aspettare che i tick passino).

8.1.1 tckd e' la parte intera superiore di  $(msec \cdot HZ) / 1000$

2.2 Definire `cpu_wait_for_interrupt()` in `bbb_cpu.h`

Se arriva altra  
interruzione?  
devo sempre  
controllare  
`ticks < expire`

```
|#define cpu_wait_for_interrupt() __asm__ __volatile__("wfi") |
+-----+
```

2.3 Aggiungere il prototipo di mdelay in comm.h

```
+-----+
| extern void mdelay(unsigned long); |
+-----+
```

2.4 Modificare main.c e sostituire loop\_delay() con mdelay(), impostando un ritardo di 1 secondo con mdelay(1000)

2.5 Problema: cosa accade quando ticks o ticks+tickd vanno in overflow? Il funzionamento non e' corretto!

2.5.1 Il contatore di 32 bit andra' in overflow dopo  $2^{32}/(1000*60*60*24) \sim 50$  giorni

2.5.2 Verificare il bug inizializzando ticks con il valore 0xffffffff-10\*HZ Dopo 10 secondi va in overflow "ticks", sto vedendo un caso di overflow.

2.5.3 Se il sistema embedded e' progettato per funzionare ininterrottamente e' necessario considerare questa possibilita' se ticks crescono, expire diventa più piccola e quindi si genera il bug appena visto.

2.6 Macro per gestire l'overflow del tick (in comm.h):

```
+-----+
|#define time_after(a,b) ((long) ((b) - (a)) < 0)
|#define time_before(a,b) time_after(b,a)
|#define time_after_eq(a,b) ((long) ((a) - (b)) >= 0)
|#define time_before_eq(a,b) time_after_eq(b,a)
+-----+
```

Sono delle macro prese da Kernel Linux, da unsigned long casto a signed long, in quanto solo il segno mi dà info.

2.6.1 Modifica mdelay():

```
+-----+
| [...]
|     while (time_before(ticks, expire))
|         cpu_wait_for_interrupt();
+-----+
```

8.6.2 Verificare che il bug e' stato corretto

3 Definizione di uno scheduler per task periodici a priorita' fissa

3.1 Task periodici caratterizzati da

3.1.1 Periodo

3.1.2 Priorita'

3.1.3 Job (funzione da eseguire ad ogni rilascio)

3.1.4 Nome (per debug)

Parto da cose semplici

3.2 Scheduler a priorita' fissa a livello di task

3.3 Numero massimo di task prefissato (default 32, modificabile)

3.4 Scheduler basato sul tick, ma invocato anche al completamento di un job

3.5 Inizialmente i job saranno non interrompibili

3.6 Osservare che l'implementazione di uno scheduler a priorita' fissa non ha bisogno di memorizzare la scadenza relativa dei task

3.6.1 In fase di creazione del task il progettista puo' scegliere di considerare la scadenza relativa impostando opportunamente la priorita' statica, ad esempio per ottenere Deadline Monotonic

4 Definizione della struttura che descrive un task (in comm.h)

```
+-----+
|#define MAX_NUM_TASKS 32
|typedef void (*job_t)(void *);
|struct task {
|
|} (void*) perchè non so cosa farà ogni task.
```

Nella dimostrazione FIXATA, giunti all'overflow, veniva stampato un valore piccolo (36) e poi ripartiva da 1037, ma il tick funziona correttamente.

```

int valid; task è usato o no?
job_t job; funzione del task
void *arg; argomento del job
unsigned long relesetime;
unsigned long released;
unsigned long period;
unsigned long priority;
const char *name;
};

```

Info per la priorità fissa del task.

- relesetime = ultimo rilascio o prossimo rilascio  
(periodo non basta)

- released (sarebbe un pending) = job rilasciati ma non eseguiti

- name = nome del task

Non mi serve scadenza relativa, riesco a ricavare dalla priorità, in quanto è fissa... altrimenti no!

## 5 Implementazione del task nel file tasks.c

### 5.1 Dichiarazione di un vettore di descrittori di task

```

struct task taskset[MAX_NUM_TASKS];
int num_tasks;

```

### 5.2 Inizializzazione del vettore di descrittori di task:

```

void init_taskset(void)
{
    int i;
    num_tasks = 0;
    for (i=0; i<MAX_NUM_TASKS; ++i)
        taskset[i].valid = 0;
}

```

5.2.1 In realta' questa funzione e' del tutto inutile in quanto sia taskset che num\_tasks sono nella sezione .bss che viene inizializzata a zero durante il bootstrap

### 5.3 Creazione di un task:

```

int create_task(job da eseguire, suo argomento, suo periodo,
                int delay, int priority, const char *name)
{
    delay simile alla fase, coincidono solo a t = 0, priorità, nome

    int i;
    struct task *t;
    for (i=0; i<MAX_NUM_TASKS; ++i)
        if (!taskset[i].valid) se =0 ho trovato "posto libero".
            break;
    if (i == MAX_NUM_TASKS) non posso creare nuovi task, ho già raggiunto
                           la soglia di 32 task.
        return -1;
    t = taskset+i;
    t->job = job;
    t->arg = arg;
    t->name = name;
    t->period = period;
    t->priority = priority;
    t->relesetime = ticks+delay;
    t->released = 0;
    irq_disable();
    ++num_tasks;
    t->valid = 1;
    irq_enable();
    printf(Task %s created, TID=%u\n", name, i);
}

```

Solo con "t->valid = 1" abilito il task per lo scheduler, prima non era abilitato.

```

    return i; stampo il pid (di Linux)
}

```

5.3.1 La sezione critica creata disabilitando le interruzioni serve a garantire che nella funzione `check_periodic_tasks()` (invocata in modo asincrono ad ogni tick) non vengano mai presi in considerazione task che non sono stati completamente inizializzati

5.3.1.1 Secondo la documentazione ARM, l'istruzione "CPSID" e' auto-sincronizzante rispetto al flusso di istruzioni in cui appare e non richiede l'uso di una memory barrier

5.3.2 Aggiungere riferimenti alle variabili globali e ai prototipi delle funzioni in `comm.h`

```

extern int num_tasks;
extern struct task taskset[MAX_NUM_TASKS];
void init_taskset(void);
int create_task(job_t job, void *arg, int period,
               int delay, int priority, const char *name);

```

## 6 Implementazione dello scheduler nel file `sched.c` *considero "job interrompibili"*

### 6.1 Definizione della funzione `check_periodic_tasks()`:

```

volatile unsigned long globalreleases = 0;
void check_periodic_tasks(void)
{
    unsigned long now = ticks; //tick corrente
    struct task *f;
    int i;
    for (i=0, f=taskset; i<num_tasks; ++f) {
        if (!f->valid)
            continue; entro qui se f non è valido. Altrimenti proseguo
        if (time_after_eq(now, f->releasetime)) { Sfrutto le macro del Kernel Linux.
            ++f->released; rilascio job
            f->releasetime += f->period; prossimo rilascio. Se fosse "Now + period" sarebbe sporadico.
            ++globalreleases; n° tot di rilasci.
        }
        ++i; incremento perchè il task era valido.
    }
}

```

Dove la metto?  
Dentro "tick.c",  
in static void `isr_tick(void)`  
metto tale funzione, perchè lo  
schedule è basato su tick, quindi ha  
senso metterlo nel file dei tick.

La funzione è asincrona, invocata per  
ogni tick, e controlla il rilascio del  
task.

### 6.2 Definizione della funzione `select_best_task()`:

```

static inline struct task *select_best_task(void)
{
    unsigned long maxprio;
    struct task *best, *f;
    int i;

    maxprio = MAXUINT;
    best = NULL;
    for (i=0, f=taskset; i < num_tasks; ++f) {
        if (f - taskset >= MAX_NUM_TASKS)
            panic0(); /* Should never happen */
    }
}

```

24/11/2022

```
        if (!f->valid)
            continue;
        ++i;
        if (f->released == 0)
            continue;
        if (f->priority < maxprio) {
            maxprio = f->priority;
            best = f;
        }
    }
    return best;
}
```

### 6.3 Definizione della funzione `run_periodic_tasks()`:

```
void run_periodic_tasks(void)
{
    struct task *best;
    unsigned long state;
    for (;;) {
        state = globalreleases;
        best = select_best_task();
        if (best != NULL && state == globalreleases) {
            best->job(best->arg);
            best->released--;
        }
    }
}
```

#### 6.3.1 Aggiungere definizioni e prototipi in `comm.h`:

```
#define MAXUINT (0xfffffffffu)
void check_periodic_tasks(void);
void run_periodic_tasks(void);
```

#### 6.4 Invocazione di `check_periodic_tasks()` ad ogni tick (in `tick.c`):

```
static void isr_tick(void)
{
    iomem(DMTIMER0_IRQSTATUS) = OVF_IT_FLAG;
    ++ticks;
    check_periodic_tasks();
}
```

### 7 Inizializzazione dello scheduler in `_init()`:

```
[...]
init_taskset();
init_ticks();
[...]
```

## 8 Verifica del funzionamento dello scheduler

### 8.1 Modificare main.c per utilizzare lo scheduler

```
+-----+
static void led_cycle(void *arg __attribute__((unused)))
{
    static int state = 1;
    leds_off_mask(0xf);
    leds_on_mask(state);
    state = (state+1) & 0xf;
}

void main(void)
{
    banner();
    if (create_task(led_cycle, NULL, HZ, 5, HZ, "led_cycle")
        == -1) {
        puts("ERROR: cannot create task led_cycle\n");
        panic1();
    }
    run_periodic_tasks();
}
+-----+
```

Passiamo alle slide E09

### 8.2 Creare il task "show\_ticks":

```
+-----+
static void show_ticks(void *arg __attribute__((unused)))
{
    puts("\nCurrent ticks: ");
    putu(ticks);
    putnl();
}

void main(void)
[...]
```

```
    if (create_task(show_ticks, NULL, 10*HZ, 5, 10*HZ,
        "show_ticks") == -1) {
        puts("ERROR: cannot create task show_ticks\n");
        panic1();
    }
[...]
```

```
+-----+
```

## 9 Esame dello scheduler non interrombibile

### 9.1 Ruolo della funzione check\_periodic\_tasks()

#### 9.1.1 Attivata periodicamente

#### 9.1.2 Rileva i rilasci dei job e aggiorna i task (campi f->released e f->releasetime)

### 9.2 Ruolo della funzione run\_periodic\_tasks()

#### 9.2.1 Cerca il task di priorit  piu' alta con job da eseguire (funzione select\_best\_task())

#### 9.2.2 Esegue il job prescelto

#### 9.2.3 Ripete all'infinito

### 9.3 Race condition: poich  check\_periodic\_tasks()   asincrona rispetto a run\_periodic\_tasks(), le modifiche ai campi released dei task possono avvenire in qualunque momento

- 9.3.1 In particolare `run_periodic_tasks()` potrebbe scegliere erroneamente un job di priorita' piu' bassa rispetto ad un altro se quest'ultimo viene rilasciato dopo che la funzione ha controllato il campo `released` nel ciclo `for`
- 9.3.2 La variabile `globalreleases` serve a mitigare questa race condition: `run_periodic_tasks()` esegue un job solo se dopo una intera scansione del vettore di task nessun nuovo job viene rilasciato
- 9.3.3 Esiste ancora una finestra temporale in cui e' possibile avere una race condition (tra il controllo `'state == globalreleases'` e l'esecuzione del job), ma in questo caso e' piu' piccola ed i tempi di blocco aggiuntivi che comporta sono trascurabili rispetto ai tempi di esecuzione dei job

/\*

vim: tabstop=4 softtabstop=4 expandtab list colorcolumn=74 tw=73

\*/