

+-----+  
| E04 |  
+-----+

#### COMPILAZIONE DEL PROGRAMMA

- Cross-compiler; su BBB si può usare un compilatore Linaro.
- Compilazione: gcc -Wall -Wextra -Os -ffreestanding -mcpu=cortex-a8 -march=armv7-a -mfloat-abi=hard -mfpu=vfpv3 -marm -noop.c
- Caricamento su BBB: ld -nostdlib -e entry\_point -o sert.elf
- Estrazione del binario: objcopy -S -O binary sert.elf sert.bin
- Disassemblaggio del binario: objdump -d sert.elf

#### ESECUZIONE DEL PROGRAMMA

- Per parlare con la BBB: lanciare minicom.
- Per caricare il programma in RAM: loadb 0x80000000; il trasferimento avviene tramite protocollo kermit.
- Per lanciare il programma: go 0x80000000.
- Per stabilire come va posizionato correttamente il codice: creare il file sert.lds (linker) con tutte le sezioni di memoria del programma.

#### ACCENSIONE DEI LED

- Variabile gpio1: entry 0x194 per l'accensione dei led (X); entry 0x134 per impostare i pin relativi ai led come output (!); entry 0x190 per lo spegnimento dei led.
- Variabile cm\_per: entry 0xac per accedere al registro GPIO1; primi due bit (modulemod) per attivare il modulo di gestione del clock (GPIO1); bit 18 (gdbclk) per attivare il circuito.
- Il modulo GPIO1 (General Purpose Input Output 1) è un componente che permette di interagire coi componenti di altre periferiche.

+-----+  
| E05 |  
+-----+

#### LAMPEGGIAMENTO DEI LED

- Cicli for annidati (esterno = loop infinito, interno = attesa attiva) per far lampeggiare i led.
- Problema: tutto il corpo del ciclo for esterno non viene proprio eseguito (X).
- Soluzione: utilizzo di asm("") e volatile per evitare le ottimizzazioni da parte del compilatore (!).

#### RIORGANIZZAZIONE DEL CODICE

- Introduzione di file header e di funzioni per modulare il codice.
- Problema: l'entry point del programma potrebbe non essere più all'indirizzo 0x80000000 (X).
- Soluzione: posizionare l'entry point all'inizio della RAM all'interno di sert.lds (!).
- Ridenominazione della funzione entry point in main() -> definizione di startup.S per linkare il simbolo main() all'entry point.
- Introduzione di macro per definire gli indirizzi di memoria di registri/variabili (come iomem()).

#### INTRODUZIONE DELLO STACK

- Aggiunta della sezione di stack in sert.lds.
- Aggiunta dell'inizializzazione dello stato della CPU in startup.S.

FUNZIONALITÀ DELLE BARRIERE DI MEMORIA: `__asm__ __volatile__ (...)`

- 1) Impedire che il compilatore ottimizzi e riordini la sequenza delle istruzioni a livello di codice sorgente.
- 2) Impedire che il processore riordini nella pipeline la sequenza di istruzioni effettivamente eseguite.
- 3) Impedire che il processore riordini la sequenza di accessi in memoria.
- 4) Impedire che il processore permetta l'esecuzione di un'istruzione prima del completamento di un accesso in memoria.

INIZIALIZZAZIONE DELL'AMBIENTE DI ESECUZIONE: `init.c`

- `init_gpio1()`: inizializzazione di `gpio1/cm_per` come visto precedentemente.
- `fill_bss()`: inizializzazione del `bss` a 0.
- `main()`: esecuzione vera e propria del programma (dove al momento c'è solo il lampeggiamento dei led).

PANIC

- Introduzione delle funzioni `panic()`, il cui argomento è un intero che indica il pattern con cui i led devono lampeggiare.

+-----+

| E06 |

+-----+

ECCEZIONI IN ARM

+-----+-----+-----+-----+			
Prio	Nome	Indirizzo	Modo di esecuzione
+-----+-----+-----+-----+			
1	Reset	0x00000000	Supervisor (codice del kernel)
6	Undef. instruction	0x00000004	Undefined (emulatori di coproc.)
7	Software interrupt	0x00000008	Supervisor (codice del kernel)
5	Prefetch abort	0x0000000C	Abort (gestori di fault)
2	Data abort	0x00000010	Abort (gestori di fault)
-	Hypervisor trap	0x00000014	-
4	IRQ	0x00000018	IRQ (interruzioni normali)
3	FIQ	0x0000001C	FIQ (fast interrupt)
+-----+-----+-----+-----+			

TABELLA DEI VETTORI DI INTERRUZIONE

- Indirizzo della tabella ottenuto leggendo il valore di un apposito registro del coprocessore del sistema ARM.
- 8 entry da 4 byte ciascuna: in ciascuna entry al massimo vi entra l'istruzione `LDR_PC_PC`, che è un incremento del PC di 6 istruzioni. Effetto finale: salto verso la locazione di memoria che dista 8 entry da quella di partenza; tale locazione di memoria conterrà il vero e proprio gestore della rispettiva eccezione.
- Aggiunta di una chiamata a `init_vectors()` all'interno di `init.c`.

PORTA SERIALE

- 1) Programmazione tramite DMA (Direct Memory Access), dove nel trasferimento dati non è coinvolta la CPU.
- 2) Programmazione tramite interruzioni.
- 3) Programmazione tramite CPU polling -> soluzione da noi adottata perché più semplice; due registri usati: THR (Transmit Hold Register) che

mantiene il dato da trasmettere e LSR (Line Status Register) il cui bit 5 indica se è possibile trasmettere dati o meno.

#### FUNZIONI PER SCRIVERE STRINGHE/NUMERI

- `putc()`: singolo carattere
- `puts()`: stringa
- `putnl()`: new line
- `puth()`: numero in esadecimale
- `putu()`: unsigned long
- `putd()`: long
- `putf()`: float -> necessità di abilitare l'accesso ai coprocessori CP10 e CP11 e di abilitare un bit nel registro VFP FPEXC -> aggiunta di una chiamata a `init_vfp()` all'interno di `init.c`.
- `putcn()`: stringa composta da un carattere ripetuto
- `printf()`: come la `printf()` che siamo abituati a conoscere

```
+-----+
| E07 |
+-----+
```

#### GESTIONE DELLE INTERRUZIONI IRQ

- Circuito Interrupt Controller: processa i segnali di interruzione provenienti dalle periferiche
- 128 linee di interruzione

#### REGISTRI PER LE INTERRUZIONI IRQ

- MIR: mascheramento delle linee.
- ILR: tipologia e priorità delle interruzioni.
- ITR: stato delle linee prima del mascheramento.
- Pending IRQ: stato delle linee dopo il mascheramento.
- Threshold: valore di soglia al di sotto del quale le interruzioni non vengono notificate.
- SIR\_IRQ: interruzione pendente più prioritaria.
- Control NewIRQAgr: se 1, consente la gestione di una nuova interruzione.

#### MODI DI ESECUZIONE DELLA CPU: registro cpsr

- SYSTEM: maschera 0x1f
- IRQ: maschera 0x17
- FIQ: maschera 0x11

#### LIVELLI PER I GESTORI DELLE INTERRUZIONI [mike leggi bene!](#)

- Basso: procedura Assembly definita all'offset 0x18 della tabella delle eccezioni che salva e recupera il contesto di esecuzione; i registri `r13=sp`, `r14=lr`, `spsr` sono duplicati per i diversi modi di esecuzione; i registri `r0`, `r1`, `r2`, `r3`, `r12`, `r13=sp`, `r14=lr`, `spsr` non vengono preservati in automatico in un'invocazione di funzione C. [sono clobbered + r13, salvo sullo stack \(diverso tra i task\)](#)
- Medio: procedura C stabilisce il tipo di interruzione e ricava l'indirizzo del gestore di alto livello da eseguire.
- Alto: ISR, ossia funzione C specifica per l'interruzione attivata.

#### ABILITAZIONE/DISABILITAZIONE DELLE INTERRUZIONI

- Il bit 0x80 di `cpsr` disabilita gli IRQ.
- Il bit 0x40 di `cpsr` disabilita i FIQ.

- Aggiunta di una chiamata a `init_intc()` all'interno di `init.c`, che si occupa di mascherare tutte le 128 linee di interruzione, disabilitare l'eventuale threshold e abilitare le interruzioni IRQ. [abilita tutte le interruzioni](#)

```
+-----+
| E08 |
+-----+
```

#### REGISTRI PER IL TICK

- TLDR: valore ricaricato dopo l'overflow.
- TCLR: controllo del timer.
- TTGR: forzatura dell'overflow.
- IRQ\_ENABLE\_SET/IRQ\_ENABLE\_CLR: abilitazione della generazione di IRQ su overflow/capture+match.
- IRQ\_STATUS: notifica della ricezione dell'interruzione.

#### GESTIONE DEL TICK

- Idea: generare un interrupt a ogni tick; generare un ulteriore interrupt quando si ha overflow, in modo tale da aggiornare il contatore al valore di TLDR.
- Tick da 1 millisecondo (1000 Hz), quando il tick hw della BBB ha una frequenza pari a 32768 Hz.
- Aggiunta di una chiamata a `init_ticks()` all'interno di `init.c` + definizione del gestore di alto livello per l'interruzione dei tick, che si occupa di incrementare il contatore dei tick.

#### ATTESA PASSIVA: funzione `mdelay()`

- La funzione prende in input un parametro msec che indica per quanti millisecondi si vuole rimanere in attesa; msec viene convertito in tick e viene sommato al contatore di tick corrente (ottenendo così `expire`). Finché contatore di tick < `expire`, si rimane in attesa passiva.
- Problema: in caso di overflow, `expire` potrebbe diventare negativo e la condizione tick < `expire` potrebbe non verificarsi mai (X).
- Soluzione: definire delle macro riprese da Linux, come `time_after(a,b)`, `time_before(a,b)`, `time_after_eq(a,b)`, `time_before_eq(a,b)`, che confrontano due istanti di tempo visti come signed long (!).

#### SCHEDULER PER TASK PERIODICI A PRIORITÀ FISSA

- Numero massimo di task prefissato (default 32).
- Scheduler basato su tick ma invocato anche al completamento di un job.
- Inizialmente i job saranno non interrompibili.
- Struttura che descrive un task: `valid`, `job`, `arg`, `releasetime`, `relased`, `period`, `priority`, `name`.
- Aggiunta di una chiamata a `init_taskset()` all'interno di `init.c`, in modo da definire l'array di 32 task prefissati.

#### FUNZIONI PER LO SCHEDULER

- `create_task()`: ricerca di una entry libera (i.e. not valid) all'interno della lista dei task del sistema; se esiste, inizializzazione dei campi della relativa struttura task; con le interruzioni disabilitate, incremento della variabile globale `num_tasks` & `t->valid=1`.
- `check_periodic_tasks()`: ciclo su tutti i task (validi) per verificare se, per ogni task, è stato rilasciato un nuovo job; in tal caso è previsto l'aggiornamento dei campi `released` e `releasetime` del task.

L'invocazione a questa funzione viene aggiunta nel gestore di alto livello per l'interruzione dei tick.

- `select_best_task()`: ciclo su tutti i task (validi) per stabilire qual è quello con job rilasciati ma non ancora completati che ha maggiore priorità.

- `run_periodic_tasks()`: ciclo infinito in cui viene invocato `select_best_task()`, viene lanciata la funzione `job()` del task e viene decrementato il campo `released` del task. NB: poiché `check_periodic_tasks()` è asincrona rispetto a `run_periodic_tasks()`, le modifiche al campo `released` dei task possono avvenire in qualunque momento: in particolare, `run_periodic_tasks()` potrebbe scegliere erroneamente un job di priorità più bassa rispetto a un altro se quest'ultimo viene rilasciato dopo che la funzione ha controllato il campo `released` nel ciclo `for`. Per mitigare: variabile `globalreleases` in `select_best_task()`.

```
+-----+
| E10 |
+-----+
```

#### SCHEDULER PER JOB INTERROMPIBILI

- Introduzione della variabile `current` che punta al descrittore di task in esecuzione.

- Introduzione dell'`idle` task.

- Ogni task ha un proprio stack.

- Cambio di contesto = posizionare `sp` in modo tale che faccia riferimento allo stack di un task differente. Qui è necessario salvare nel descrittore del task i registri `non clobbered` (`r4-r11`, `r13=sp`) -> necessità di aggiungere i campi `sp`, `regs[8]` nella struttura del task.

- Il cambio di contesto avviene solo se si sta per tornare all'esecuzione di un job; viene effettuato nel gestore di basso livello subito dopo la terminazione del gestore di medio livello.

#### FUNZIONI C PER LO SCHEDULER PER JOB INTERROMPIBILI

- `_switch_to()`: salvataggio dei registri `non clobbered` nel descrittore del task che sta rilasciando la CPU; recupero dei registri `non clobbered` dal descrittore del task che sta per essere schedulato in CPU; spostamento di `sp` verso lo stack del task che sta per essere schedulato in CPU; `current = task` che sta per essere schedulato in CPU; `naked` return.

- `init_task_context()`: posizionamento di `sp` nello stack corretto; inizializzazione dei registri `non clobbered` diversi da `r13=sp` (`r4-r11`) a 0; posizionamento sullo stack dei registri `clobbered` (`r0=puntatore al descrittore del task`, `r1=0`, `r2=0`, `r3=0`, `r12=0`, `r14=lr=0`, `r15=pc=task_entry_point()`, `spsr=SYS_MODE`): in tal modo, lo stack viene inizializzato in modo da essere `analogo allo stack` di un task che era in esecuzione ma è stato interrotto; `lr=0` perché `task_entry_point()` è senza ritorno.

- `task_entry_point()`: ciclo infinito in cui si esegue la funzione `job()` del task con le interruzioni abilitate, si decrementa il campo `released` del task e si invoca `_sys_schedule()` con le interruzioni disabilitate.

#### FUNZIONI C MODIFICATE RISPETTO AL CASO DI JOB NON INTERROMPIBILI

- `create_task()`: il controllo sulle entry libere nell'array di task parte dall'indice 1 e non più dall'indice 0 (a 0 c'è l'idle task); aggiunta dell'invocazione a `init_task_context()`.
- `check_periodic_tasks()`: la scansione dei task per stabilire chi è stato rilasciato parte dall'indice 1 e non dall'indice 0 (l'idle task non è periodico); aggiunta di `trigger_schedule=1` se è stato rilasciato almeno un job, dove `trigger_schedule` è una nuova variabile globale che indica se lo scheduler deve essere invocato appena possibile per stabilire quale task deve essere mandato in esecuzione.
- `select_best_task()`: inizialmente il task migliore è l'idle task e la ricerca del task migliore parte dall'indice 1 dell'array di task; solo se non è stato trovato alcun task periodico da schedulare, il task scelto sarà effettivamente l'idle task.

#### PRIMA VERSIONE DELLO SCHEDULER

- `schedule()`: funzione che sostituisce `run_periodic_tasks()` e che, invocando `select_best_task()`, restituisce l'indirizzo del descrittore del task da eseguire, oppure NULL se il task corrente è già il migliore; `trigger_schedule=0` alla fine di questa funzione.
- `_irq_schedule()`: funzione Assembly che invoca `schedule()` e verifica il valore restituito: se 0 (NULL), **ripristina** lo stato dei registri clobbered del task corrente e scrive `spsr` in `cpsr`; altrimenti, invoca `_switch_to()`. È eseguita in modalità SYSTEM.
- `_sys_schedule()`: funzione Assembly che salva i registri clobbered sullo stack (in modo tale che poi all'invocazione di `_irq_schedule()` la struttura dello stack sia identica a quella che si ha a seguito di un'interruzione) e invoca `_irq_schedule()`. È eseguita in modalità SYSTEM quando il vecchio job termina.
- `_irq_handler()`: gestore delle interruzioni di basso livello; rispetto al caso di job non interrompibili, aggiunge un'invocazione a `_irq_schedule()` subito dopo `_bsp_irq()`, che verrà effettuata se l'interruzione che si sta gestendo non è annidata e se `trigger_schedule==1` (si incrementa e decrementa il livello di annidamento nel gestore di medio livello `_bsp_irq()`).

#### SECONDA VERSIONE DELLO SCHEDULER

- Problema: `schedule()` invoca `select_best_task()` con le interruzioni abilitate (è una funzione lenta, per cui non vogliamo eseguirla con le interruzioni disabilitate); di conseguenza, un interrupt può sopraggiungere durante l'esecuzione di `select_best_task()` e può provocare una nuova invocazione a `schedule()` innestata rispetto all'invocazione originale -> `schedule()` è detta funzione rientrante; ciò è un problema nel momento in cui `schedule()` aggiorna variabili globali, causando inconsistenze nelle variabili stesse (X).
- Soluzione: impedire invocazioni innestate a `schedule()` sfruttando come meccanismo di sincronizzazione una variabile 'sentinella' `do_not_enter (!)`.

```
+-----+
| E11 |
+-----+
```

DESCRITTORE DI TASK NELLO SCHEDULER IBRIDO

- Campo priority: se priorità fissa, è uguale a priorità del task; se EDF, è uguale a priorità del job in esecuzione/primo job pendente e indica la scadenza assoluta di quel job.
- Campo deadline (NEW): se priorità fissa, vale 0; se EDF, è uguale alla scadenza relativa del task.

#### FUNZIONI C MODIFICATE RISPETTO AL CASO DI SOLI TASK A PRIORITÀ FISSA

- create\_task(): aggiunta di un parametro type per indicare se il task da creare è FPR o EDF; modifica dell'inizializzazione del campo priority del nuovo task + aggiunta dell'inizializzazione del campo deadline.
- select\_best\_task(): aggiunta di due flag fpr, edf che indicano rispettivamente se è stato selezionato un task di tipo fpr e se è stato selezionato un task di tipo edf. All'interno del ciclo su tutti i task: se `fpr==1`, si guardano solo task di tipo fpr; altrimenti, si cercano prima i task di tipo fpr e, in mancanza di essi, si cercano i task di tipo edf.
- task\_entry\_point() - al completamento di un job: aggiunta di un controllo sul rispetto della scadenza per i task edf; aggiornamento del campo priority per fissare la scadenza assoluta del prossimo job (che sarà pari a vecchia scadenza assoluta + periodo).

#### WATCHDOG

- È un timer che, quando scade (tipicamente ogni minuto), provoca il reset del processore; il relativo contatore si trova nel registro WDT\_WTGR (X).
- Soluzione: definire un nuovo task periodico che sovrascrive WDT\_WTGR (!).

#### HEARTBEAT

- Lampeggiamento del led 3 mentre un task diverso dall'idle task è in esecuzione. Se esegue l'idle task, il led rimane spento.

```
+-----+
| E12 |
+-----+
```

#### SERVER CBS

- Numero massimo di task aperiodici prefissato (default 8).
- Definizione di struct cbs\_queue coi seguenti campi: task (di fatto il server CBS è modellato come task), num\_workers (= #task aperiodici presenti), workers (= array delle funzioni dei task aperiodici presenti), args (= argomenti delle funzioni dei task aperiodici presenti), pending (= array che indica per ogni task aperiodico quanti sono i job rilasciati ma non ancora completati).

#### DESCRITTORE DI TASK NELLO SCHEDULER CON CBS

- Campo budget (NEW): se CBS, è uguale al budget corrente del server (sempre >0); altrimenti vale 0.
- Campo period: se CBS, vale p\_s.
- Campo deadline: messo in union con budget\_max; se CBS viene usato budget\_max (=e\_s), altrimenti viene usato deadline.
- Campo priority: se CBS, vale d\_s.

#### IMPLEMENTAZIONE DEL SERVER CBS

- `cbs_server()`: ciclo sulle entry (valide) dell'array di task aperiodici; se si trova un task `T_i` con job pending, si esegue quello e, disattivando le interruzioni, si decrementa `pending[i]`. Notare che viene selezionato sempre il task aperiodico con indice più basso (i task aperiodici hanno priorità).
- `irqsafe_activate_cbs_worker()`: rilascio di un nuovo job di uno specifico task aperiodico; esecuzione di opportuni check per stabilire se `d_s` deve essere ritardato.
- `activate_cbs_worker()`: disabilitazione delle interruzioni; invocazione di `irqsafe_activate_cbs_worker()`; riabilitazione delle interruzioni.
- `decrease_cbs_budget()`: decremento del budget che viene subito ricaricato se diventa nullo; ritardo di `d_s`. Questa funzione viene invocata all'interno di `isr_tick()` (a patto che si tratti di un task CBS).
- `init_cbs()`: inizializzazione del server CBS, con creazione del primo task aperiodico.
- `add_cbs_worker()`: aggiunta di un nuovo task aperiodico nel server CBS.
- Modifica di `create_task()`, `check_periodic_tasks()` e `task_entry_point()` per tener conto anche dell'esistenza dei task CBS.