Progettazione e Implementazione di un Taskset Real-Time per Rover

Software Engineering for Embedded Systems

Angelo D'Amante
angelo.damante@stud.unifi.it

Kevin Maggi
kevin.maggi@stud.unifi.it

A.A. 2020/2021

Sommario

In questo lavoro presentiamo un robot *rover-like* basato su Raspberry Pi 3B+ equipaggiato con sistema operativo Real-Time VxWorks su cui viene eseguito un taskset, progettato, analizzato e validato tramite il tool Oris 1.0, per compiere una missione. La realizzazione sfrutta il formalismo delle PTPN che, integrandosi nel modello a V, supporta varie fasi di quest'ultimo.

Introduzione

Lo sviluppo di sistemi Real-Time si basa sulla verifica della correttezza temporale degli eventi, intesa anche come istante temporale nel quale il risultato della computazione viene prodotto.

In letteratura esistono molti modi per studiare la schedulabilità di un *taskset*, tra cui il formalismo delle *Preemptive Time Petri Net* (PTPN) che, come descritto in [4], può essere integrato nel *modello a V* del software *life-cycle*.

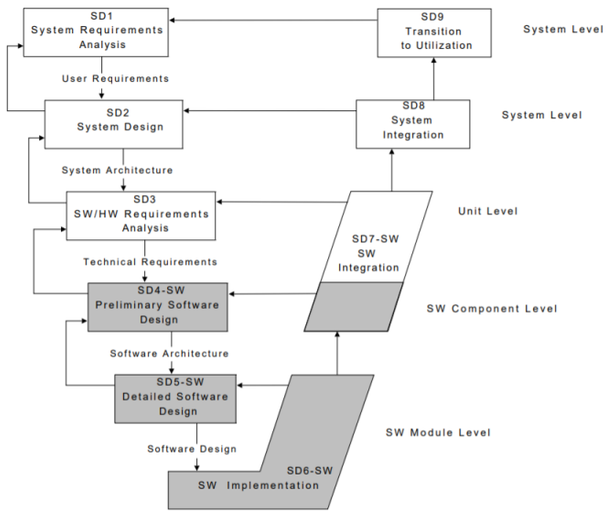

Figura 1: Modello a V

In questo lavoro abbiamo realizzato un robot *rover-like* basato su Raspberry Pi 3B+ equipaggiato con sistema operativo Real-Time VxWorks e un taskset per compiere delle missioni. Inizialmente abbiamo svolto un'analisi dei requisiti:

- Funzionali: il sistema deve fornire i servizi e le funzionalità di un tipico rover. Quindi far muovere il robot per raggiungere un determinato obiettivo seguendo delle direttive ben precise fornite dall'utente e interagire con tutta la sensoristica di bordo;

- Non Funzionali: il sistema deve rispettare i vincoli temporali tipici di un sistema Hard Real-Time. Quindi far rispettare il robot le sue deadline coincidenti con i tempi minimi di inter-arrivo. Inoltre, abbiamo stabilito dei requisiti energetici per non sovraccaricare la scheda Raspberry e dei vincoli sul processo di sviluppo che consistono nell'uso del software Workbench 4 di Wind River con tutte le impostazioni del Build Source Project (BSP).

La progettazione, analisi, implementazione e validazione hanno seguito il suddetto modello a V, articolandosi in varie fasi (come in figura 2); dopo una fase iniziale in cui abbiamo ideato i requisiti, ovvero i compiti che il rover avrebbe dovuto svolgere

(quello che nel modello a V costituisce approssimativamente le fasi SD1, SD2 ed SD3), ci siamo occupati di:

- *Definizione del taskset*: sono stati elencati i task che avrebbero permesso al rover di assolvere ai suoi compiti, assegnando ad ognuno il *release time*, il periodo nel caso dei task periodici e il tempo minimo di inter-arrivo per i task sporadici (non sono previsti task con jitter sul tempo di inter-arrivo), e la *deadline*.

- *Stesura della timeline*: è stato rappresentato il taskset in forma di *timeline*, si tratta di una specifica semiformale che colma il gap tra la pratica industriale e l'uso di sistemi Real-Time. In questa fase viene fatto un design preliminare del software, dividendo i task in chunk di computazione, ognuno con un suo BCET e WCET, e allocando le risorse ad ogni blocco computazionale;

- *Traduzione in PTPN e analisi*: la timeline può essere tradotta algoritmicamente in PTPN (come descritto in [2], [4], [5] e [8]). Grazie a questo formalismo è possibile analizzare il taskset per studiarne la schedulabilità;

- *Implementazione*: è stato implementato il taskset in VxWorks, effettuando preliminarmente una ricerca sulle modalità di interazione tra il bus GPIO e l'RTOS;

- *Execution Time profiling*: instrumentando il codice tramite opportune funzioni di log, è possibile verificare l'effettivo rispetto delle deadline da parte del taskset in fase di esecuzione.

Tutti questi passi appartengono alle fasi SD4, SD5 ed SD6 del modello a V.

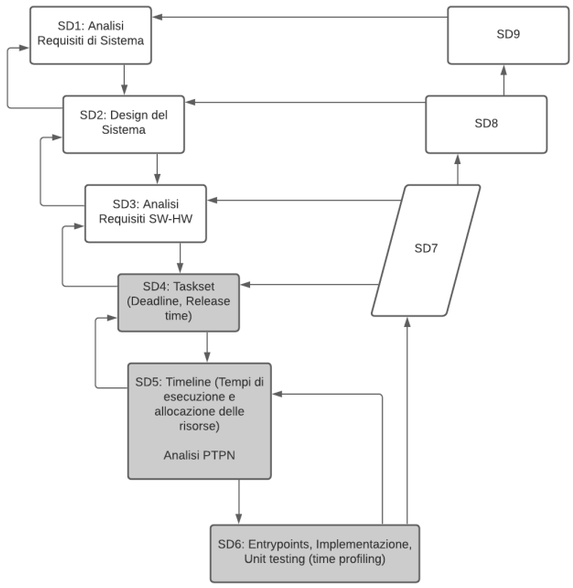

Figura 2: Percorso seguito

Le fasi di analisi della PTPN e di analisi dell'Execution Time profile sono stati eseguiti tramite il tool Oris 1.0 [3][1].

Il rover, come detto, si basa su Raspberry Pi 3B+ ed è dotato di un modulo integrante motori (quindi si muove davvero); invece la parte sensoristica è stata simulata sostituendo ad ogni sensore un LED montato su una breadboard[1].

[1]La difficoltà incontrata per interfacciare il SO VxWorks con il bus GPIO e la sofferta riuscita finale ha assegnato al rover il nome *Resilience*

Questo lavoro si basa su un lavoro precedemente svolto da altri colleghi [9], i quali hanno sviluppato una funzione di BusySleep e degli strumenti appositi per studiarne il comportamento. È possibile trovare tutto il materiale di questo progetto nella repository GitHub [7].

Figura 3: Workflow

Progettazione del Taskset

Definizione del Taskset

Resilience lo abbiamo ideato con lo scopo di muoversi e raccogliere dati e materiali scientifici di vario tipo, quindi deve assolvere a più compiti, riassunti di seguito:

- *Gestione*: comunicazione con il satellite;
- *Navigazione*: calcolo della direzione da seguire, movimento;
- *Raccolta dati e materiali scientifici*: scatti fotografici, raccolta di campioni geologici (di superficie e non), memorizzazione della pressione atmosferica, calcolo dell'altitudine (sulla sola base della pressione atmosferica), memorizzazione della temperatura, rilevazione di tempeste di sabbia.

Ognuno di questi compiti viene assolto con un task; sono tutti di tipo periodico (quindi *time driven*) tranne la comunicazione col satellite che avviene su richiesta dell'utente e il calcolo dell'altitudine che deve avvenire successivamente alla misurazione della pressione atmoserica (dunque sono task di tipo *event driven*).

La comunicazione col satellite deve avere un tempo minimo di inter-arrivo di 10 secondi, i task riguardanti la navigazione hanno periodo di 5 secondi, infine i task rigurdanti la raccolta di dati e materiali scientifici hanno periodo di 10 secondi, tranne la raccolta di campioni geologici che avviene più raramente (periodo di 20 secondi).

È stato deciso che le deadline corrispondano ai periodi per i task periodici e al tempo minimo di inter-arrivo per il task sporadico; per il task event driven del calcolo dell'altitudine, non potendo definire un periodo minimo di inter-arrivo, è stata fissata a 10 secondi.

Il task del movimento ha un *offset* di 2.5 secondi. Questa scelta è stata fatta in un secondo momento, infatti si è reso necessario tornare indietro nel modello a V. La prima versione del taskset, quella senza l'offset, in fase di analisi del taskset (tramite il formalismo delle PTPN e il tool Oris 1.0, come descritto nelle prossime sezioni), è risultato non schedulabile perché il task del movimento in alcune occasioni sforava la deadline. Infatti i job di questo task venivano rilasciati insieme ai job del task per il calcolo della direzione (avendo lo stesso periodo), ma essendo vincolati ad essi da vincoli di precedenza (in particolare come vedremo dopo da uno scambio di messaggi) venivano ritardati e quindi portati a sforare facilmente la deadline.

In tabella 1 sono riportati in maniera sintetica tutti i parametri dei task.

Timeline

Le *timeline* sono una specifica semiformale che colma il gap tra le pratiche industriali consolidate nel campo dello sviluppo di software Real-Time e le specifiche formali, come le PTPN, che permettono di analizzare un taskset.

Task	Period	Offset	Deadline
Satellite comunication	$[10s, \infty]$	-	$10s$
Direction	$2.5s$	-	$2.5s$
Movement	$2.5s$	$2.5s$	$2.5s$
Photograph	$10s$	-	$10s$
Geological Samples Collection	$20s$	-	$20s$
Atmospheric Pressure	$10s$	-	$10s$
Altitude Record	-	-	$10s$
Temperature Record	$10s$	-	$10s$
Sand Storm Detection	$10s$	-	$10s$

Tabella 1: Taskset

In fase di stesura della timeline i job vengono suddivisi in blocchi computazionali detti *chunk* a cui vengono assegnati un entrypoint e quindi un *BCET* e un *WCET*, le risorse di cui fanno uso (non solo intese come CPU, ma anche come risorse a cui accedono in sezione critica, quindi i semafori utilizzati) e la priorità.

Il taskset di *Resilience*, espresso in forma di timeline, è quello raffigurato in figura 4.

I task sono stati suddivisi in 4 livelli di priorità: il task per la comunicazione col satellite ha la priorità massima (1), al livello subito sotto (2) si trovano quelli per il movimento e per lo scatto delle fotografie, poi a seguire (priorità 3) la raccolta di campioni geologici e infine gli altri di rilevazione dei dati a priorità minima (4).

Tutti i chunk eseguono sulla stessa CPU, ovvero il sistema sarà single-core.

Come anticipato nella precedente sezione i task della direzione e del movimento sono legati da un vincolo di precedenza, ovvero uno scambio di messaggi tramite Inter-Process Comunication. Analogo meccanismo vincola anche i task di misurazione della pressione atmosferica e quello di calcolo dell'altitudine con la sola differenza che in questo caso l'invio del messaggio nella mailbox funziona anche da evento di attivazione del task event driven.

C'è anche un vincolo di risorsa condivisa tra i task di movimento, fotografia e raccolta di campioni geologici. In particolare le azioni di movimento, scatto fotografico, trivellazione e raccolta dei campioni non possono interrompersi a vicenda per ovvi motivi. Da notare che il task di raccolta dei campioni geologici acquisisce il mutex all'inizio del primo chunk e lo rilascia al termine del secondo (come descritto in [5]).

Essendo i sensori solamente simulati non abbiamo avuto dei veri requisiti temporali riguardo ai BCET e WCET dei vari chunk, che abbiamo quindi impostato liberamente cercando di renderli verosimili: per esempio i chunk che vanno a leggere un sensore hanno tempi di esecuzioni più lunghi dei chunk che invece computano solamente. L'unica eccezione è data dal chunk C32 che effettivamente attiva i motori: abbiamo deciso che questo chunk dovesse ad ogni istanza del job ruotare il rover di 90° o farlo muovere di una ventina di centimetri. Del tutto empiricamente abbiamo trovato il tempo necessario a compiere le due azioni e abbiamo impostato il BCET e WCET di conseguenza.

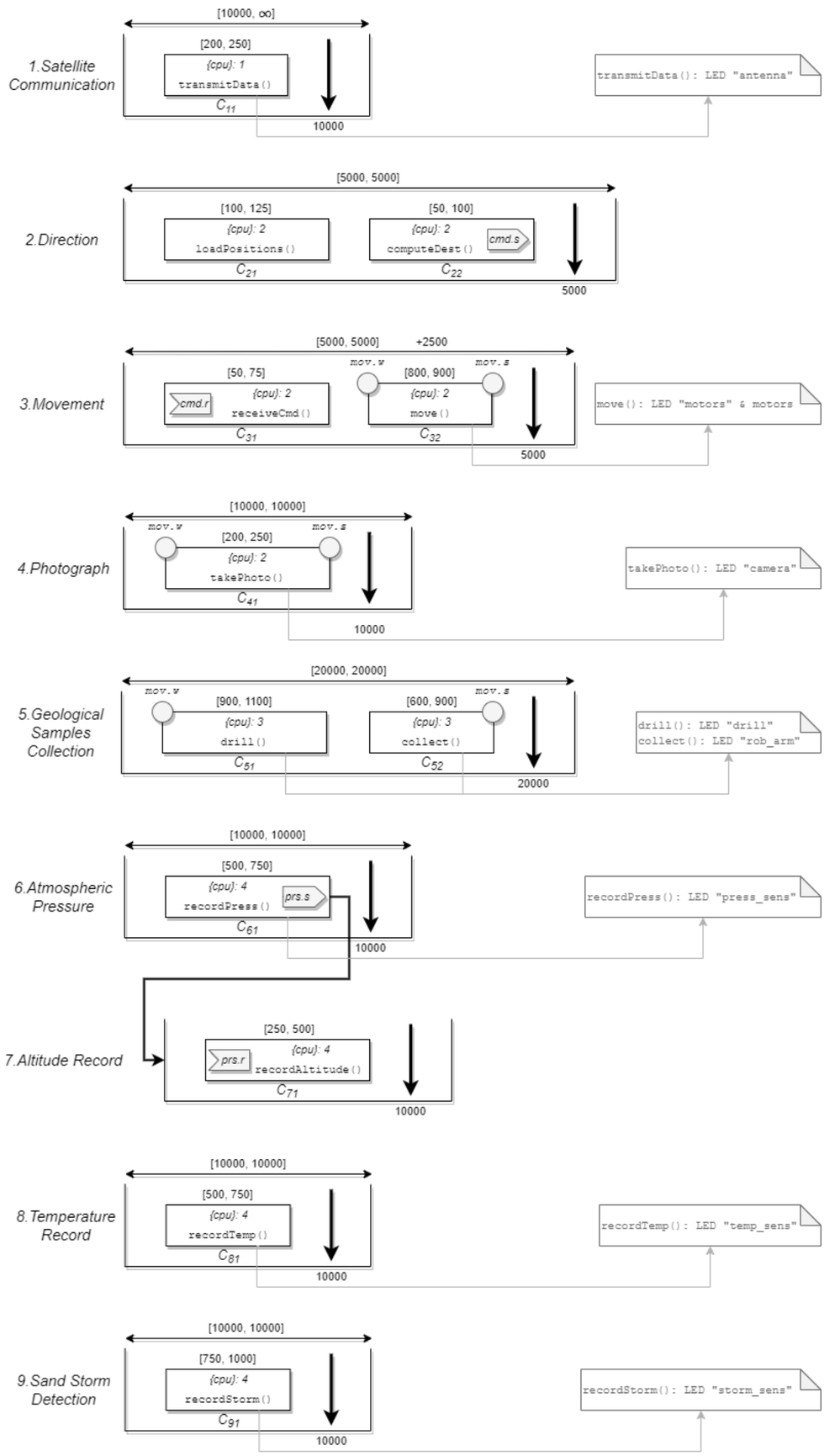

Figura 4: Timeline del taskset

PTPN

Traduzione

La timeline è stata tradotta in PTPN seguendo i metodi descritti in [2], [4], [5] e [8], ottenendo la PTPN raffigurata in figura 7.

In particolare per evitare il fenomeno dell'inversione di priorità viene adottato il protocollo di *Priority Ceiling*, quindi il task dei campioni geologici prima di acquisire il semaforo subisce un *boost* venendo portato dalla priorità 2 alla priorità 3[2].

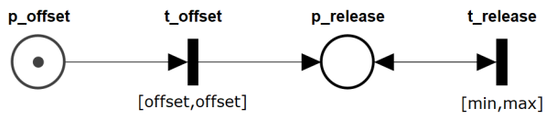

Figura 5: Traduzione in PTPN dell'offset

L'offset è modellato anteponendo alla transizione di rilascio una transizione che si trova abilitata una volta sola (ha come precondizione un posto con un gettone che non è postcondizione di nessuna transizione) il cui *firing* abilita la transizione di rilascio (che si riabilita autonomamente, quindi è sempre abilitata, come tutte le transizioni di release). La figura 5 illustra meglio il concetto.

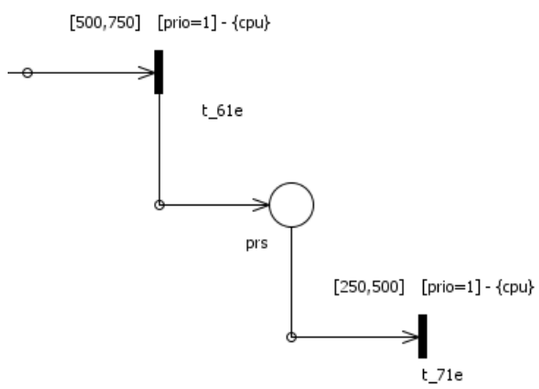

Figura 6: Traduzione del task event driven

Interessante vedere anche la traduzione del task event driven per l'altitudine. Il task consiste solamente di un chunk che riceve un messaggio. Normalmente si tradurrebbe con un "blocco" di ricezione con una transizione immediata e un "blocco" di esecuzione. In questo caso però il firing della transizione che invia il messaggio funge da transizione di rilascio per il job del task event driven. Quindi il posto che rappresenta la mailbox avrà la funzione di posto del "blocco" di esecuzione del chunk e a questo basta aggiungere una transione che modella l'esecuzione del chunk. Nel nostro caso (in figura 6) `t_61e` è il termine dell'esecuzione del chunk che invia il messaggio alla mailbox `prs`; l'esecuzione del task event driven consiste nella sola transizione `t_71e`: `t_61e` sostituisce il rilascio del job (un'ipotetica transizione `t_7r` secondo la nomenclatura usata in figura 7) e `prs` sostituisce il posto di esecuzione (un ipotetico posto `p_71e`).

Analisi

La PTPN ottenuta è stata analizzata con il plugin TCAnalyzer del tool Oris ottenendo uno spazio degli stati di quasi 30.000 stati. Dopodiché grazie al plugin TraceExtractor è stato possibile estrarre tutte le possibili tracce di esecuzione per ognuno dei task.

[2]Da notare che in Oris, a differenza della timeline, la priorità maggiore corrisponde al valore assoluto più alto

Data la complessità del taskset il numero di tracce per ogni task è abbastanza elevato (e anche variegato: si passa dalle poche centinaia ad alcuni milioni). Per questo motivo per verificare che il tempo di esecuzione massimo di ogni traccia non sforasse la deadline, abbiamo scritto un semplice script Python[3] che presi i file delle tracce (e la deadline di ciascun task) ne verificasse la possibilità di schedularlo.

È stato in questa fase che è emerso che la prima versione del taskset non soddisfaceva i requisiti: infatti il task relativo al movimento sforava la deadline in alcuni casi. Come detto, è stato sufficiente aggiungere un offset iniziale a quel task per rendere il taskset schedulabile.

[3]Si trova nella repo [7]

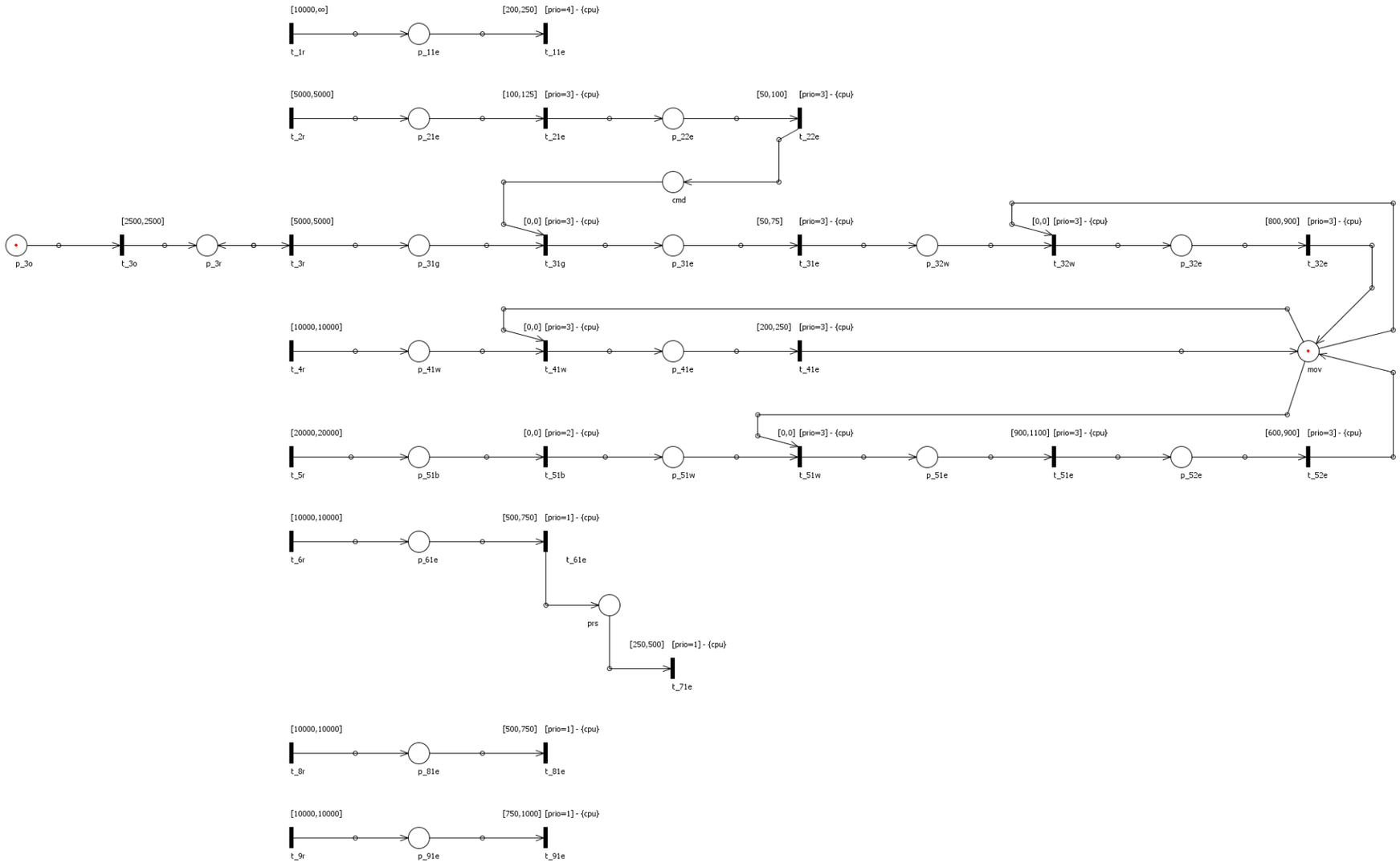

Figura 7: PTPN del taskset

Implementazione del taskset

Sviluppo su VxWorks

VxWorks è un sistema operativo Real-Time di tipo Unix-like sviluppato da Wind River Systems e include un kernel multitasking con uno scheduler di tipo preemptive e una gestione rapida degli interrupt, estesi meccanismi di inter-process communication e funzionalità di sincronizzazione, nonché un file system [11].

Lo sviluppo software è svolto su un computer *"host"* con sistema operativo Unix o Windows e viene successivamente cross compilato per l'architettura *"target"*, tramite una toolchain definita dagli sviluppatori.

Target

Resilience è equipaggiato con una scheda Raspberry Pi 3B+, che presenta, i seguenti componenti:

- CPU Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit SoC @ 1.4GHz;
- 1GB LPDDR2 SDRAM;
- 2.4GHz e 5GHz IEEE 802.11.b/g/n/ac wireless LAN, Bluetooth 4.2, BLE;
- Gigabit Ethernet over USB 2.0 (max throughput 300 Mbps);
- Extended 40-pin General Purpose Input Output (GPIO) header,
- Porta micro SD per caricare il sistema operativo e memorizzare dati;

dove, nella micro SD abbiamo caricato VxWorks e il bus GPIO è stato utilizzato per interfacciarci ai sensori simulati, nonchè ai motori, di cui ne parleremo meglio dopo.

Host

Per lo sviluppo abbiamo usato Workbench 4 [12] creando un Build Source Project (BSP) e selezionando il target di interesse. A quel punto, risulta possibile sviluppare un Downloadable Kernel Module (DKM) sulla base del BSP appena creato e buildato.

Un'alternativa molto comoda per la compilazione di codice su CLI, che non utilizza l'IDE appena citato, è quello di usare il comando `source` per assegnare l'SDK al compilatore `gcc` dell'host e compilare direttamente da terminale [6].

Per poter comunicare con il target e passare i moduli compilati si avvia una connessione FTP con `pyftpdlib` e si instaura una comunicazione nel target con `netDevCreate()`. Un'idea del workflow seguito in fase di implementazione è rappresentata dalla figura 8.

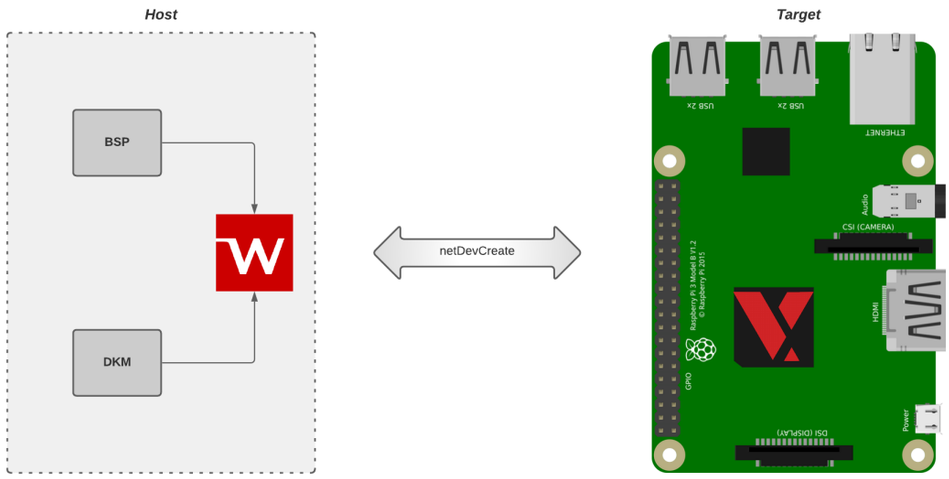

Figura 8: Host e Target

Hardware di Resilience

Riferendoci ad un vero rover dotato di una componentistica avanzata, abbiamo supposto che Resilience sia dotato di sensori e parti hardware, che chiaramente non abbiamo a disposizione, e, per sopperire a tale mancanza, essi vengono simulati da LED che si accendono all'acquisizione del task e si spengono al rilascio.

Per quanto riguarda il movimento, invece, Resilience è dotato di quattro motori CC connessi ad una *shield* controllabile dal bus GPIO. Anche per i motori è presente un sensore che indichi l'attivazione e spegnimento di essi simulato da un LED.

Inoltre, come mostrato nella fase di progettazione, `move()`, `takePhoto()`, `drill()` e `collect()` condividono un semaforo mutex, e abbiamo ritenuto interessante montare un LED che segnali l'acquisizione del mutex.

Il rover-like sviluppato in questo progetto, chiamato Resilience, è mostrato in figura 9.

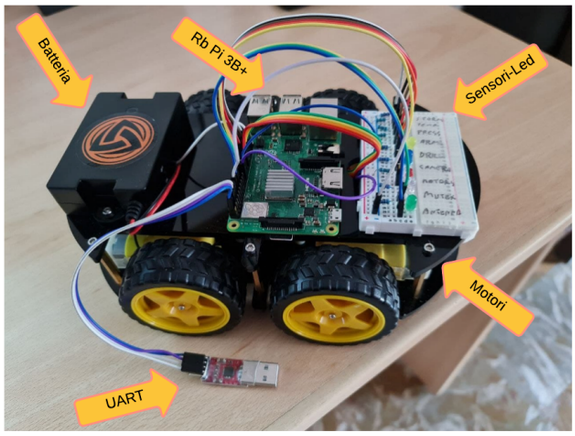

Figura 9: Resilience

Sensori LED

I sensori sono rappresentati da:

- un LED che simula il sensore che rileva tempeste di sabbia, chiamato **Storm**;
- un LED che simula il sensore di temperatura, chiamato **Temp**;
- un LED che simula il sensore che rileva la pressione atmosferica, chiamato **Press**;
- un LED che simula un braccio robotico per la raccolta di campioni geologici, chiamato **Arm**;
- un LED che simula una trivella per poter scavare, chiamato **Drill**;
- un LED che simula una videocamera, chiamato **Camera**;
- un LED che simula l'interazione con il satellite, chiamato **Antenna**;
- un LED che simula l'attivazione del movimento, chiamato **Motors**;
- un LED che segna l'acquisizione del mutex, chiamato chiaramente **Mutex**;

Motor Shield

Per le ruote, sono stati utilizzati quattro motori in CC che non è possibile attaccare direttamente al bus GPIO, perchè altrimenti ci sarebbe un sovraccarico di tensione sul bus. Per tale motivo, abbiamo utilizzato il motor shield `l298n` mostrato in figura 10.

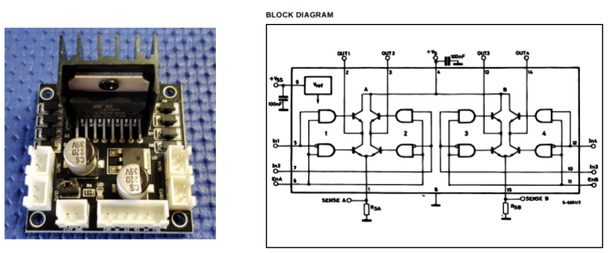

Figura 10: Modulo l298n

Possiamo notare che montare quattro motori in questa shield non vuol dire dotare Resilience di 4 Gradi di libertà (DOF), infatti, il chip embedded l298n presenta solo due `pin_enabLED` che evidenziano i motori di destra e sinistra rispettivamente. Quindi è possibile comandare due motori di destra e due motori di sinistra in maniera sincrona. Ovvero, entrambi i motori di uno stesso lato seguono la stessa direttiva di movimento. Quindi, in definitiva, Resilience presenta 2 DOF.

Circuito

Il circuito finale mostrato in figura 11 presenta la configurazione completa dei collegamenti al bus GPIO per i LED, i motori e la motor shield. Notiamo che quest'ultima è collegata con massa comune al bus e un'alimentazione esterna di 12V.

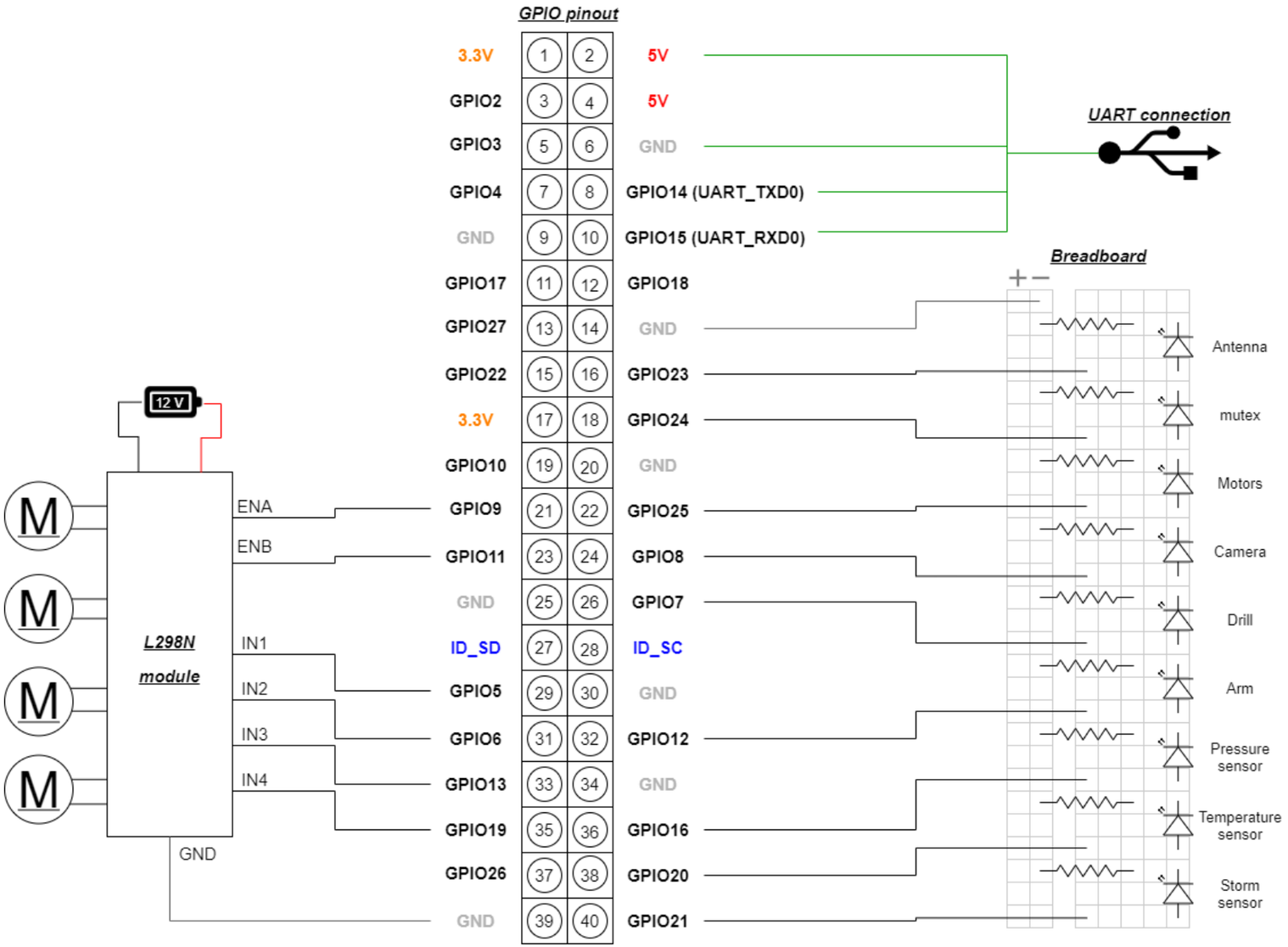

Figura 11: Configurazione GPIO

Implementazione

Una delle parti più complicate e difficili di questo progetto è stata l'integrazione con l'hardware e l'interfacciamento con il bus GPIO da parte di VxWorks, necessario per poter ottenere una piattaforma *rover-like*.

I primi tentativi ci hanno portato a studiare le librerie `pigpio`, `WiringPi`, `bcm2837` trovate in [10]. Ma tutte e tre presentano il problema di accedere al file `/dev/mem` non accessibile sul Raspberry da VxWorks. Abbiamo tentato l'accesso diretto ai registri di sistema per controllare il bus [10] e anche in questo caso, il risultato è stato deludente.

Libreria gpioLib

L'utilizzo della libreria `vxbGpioLib` è stata la soluzione al nostro problema. Per semplificare l'interazione con questa libreria di sistema, abbiamo scritto un'interfaccia per l'accesso ai vari pin del bus.

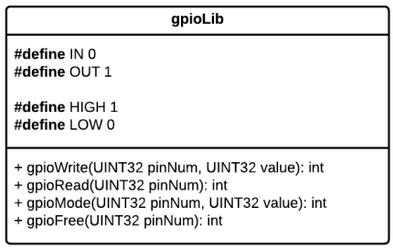

Figura 12: gpioLib.h

Libreria l298n

Per gestire la shield abbiamo usato la libreria appena descritta, mandando i segnali ai pin corrispondenti della scheda. Per semplicità non è stato implementato nessun meccanismo di controllo sulla velocità, questo perchè con la libreria `vxbGpioLib` non è possibile inviare segnali PWM, ma solo segnali digitali a duty cycle fisso pari a 100%.

La logica che sta dietro al modulo l298n consiste, come già anticipato, nel controllare i motori montati su uno stesso lato (destra o sinistra) in modo sincrono ed è mostrata in figura 13.

ENA	IN1	IN2	DC Status
LOW	x	x	STOP
HIGH	LOW	LOW	BRAKING
HIGH	HIGH	LOW	FORWARD
HIGH	LOW	HIGH	BACKWARD
HIGH	HIGH	HIGH	BRAKING

ENA	IN3	IN4	DC Status
LOW	x	x	STOP
HIGH	LOW	LOW	BRAKING
HIGH	HIGH	LOW	FORWARD
HIGH	LOW	HIGH	BACKWARD
HIGH	HIGH	HIGH	BRAKING

Figura 13: Logica del modulo

Per semplificare questa logica e l'interazione con la libreria `gpioLib.h`, anche in questo caso abbiamo scritto un'interfaccia per gestire il modulo l298n mostrata in figura 14.

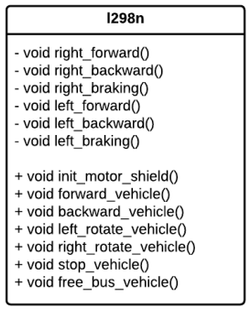

Figura 14: l298n.h

Codice di Resilience

Tutto il codice sviluppato per questo progetto si trova in [7]. Giusto per orientarci mostriamo preliminarmente il `tree` del workspace di VxWorks.

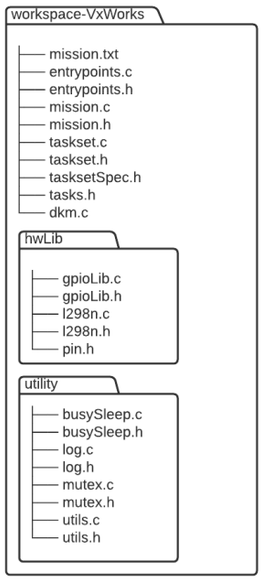

Figura 15: Layout del Workspace

In `hwLib` sono presenti le librerie per gestire l'hardware di Resilience e in `utility` si trovano i moduli per simulare la computazione, gestire il mutex e le funzioni di log [9].

Esaminiamo nel dettaglio ogni file presente nella directory principale così da dare un'idea della toolchain di sviluppo.

taskSetSpec.h

Il file `taskSetSpec.h` definisce le caratteristiche del Taskset, come le priorità, i periodi (ove previsti), i BCET e WCET di ogni chunk. Riportiamo a titolo di esempio, le caratteristiche del task *Movement*.

```
#define T3 5000
#define P3 2
#define PHASE 2500
#define C31_BCET 50
#define C31_WCET 75
#define C32_BCET 800
#define C32_WCET 900
```

mission.h

Resilience compie la missione definita dall'utente nel file `mission.txt`, essa viene caricata dalla funzione `load_mission_file()` definita in `mission.h`. La missione viene dunque caricata ed eseguita dal chunk `move()` una direttiva alla volta (figura 16) e da li in poi, dopo il completamento, il rover segnalerà solo il sensore dei motori ma resterà fermo.

Definiamo sessione la fase che va dallo `start()` allo `stop()`. Il caricamento della missione e l'inizializzazione dei motori avviene all'inizio della sessione. Tra una sessione ed un'altra è chiaramente possibile aggiornare la missione da parte dell'utente.

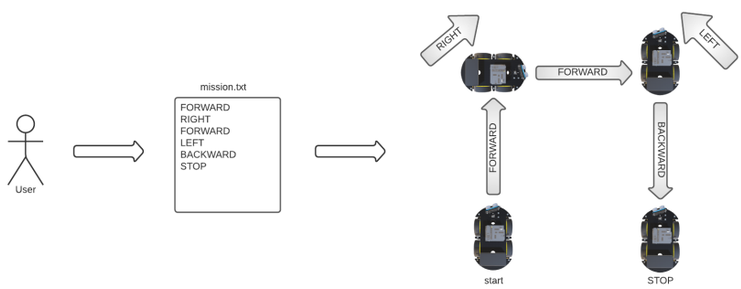

Figura 16: Missione compiuta di Resilience

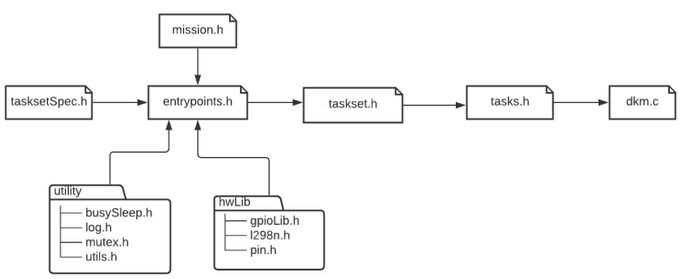

Figura 17: Toolchain Headers

dkm.c

Tutta la routine, parte effettuando lo spawn della funzione `start()` ed è possibile stopparla con lo spawn della funzione `stop()`. Dopo aver descritto tutti i file principali, la toolchain è mostrata in figura 17.

In particolare, come già anticipato sopra, il sistema è single-core e quindi è stato effettuata l'assegnazione ad un singolo core per evitare l'esecuzione parallela. Assegnazione che si riflette a cascata su tutti i task rilasciati da questo.

```
void start(){
    load_mission_file();
    init_motor_shield();

    cpuset_t affinity;
    CPUSET_ZERO(affinity);
    CPUSET_SET(affinity, 1);
    taskCpuAffinitySet(taskIdSelf(), affinity);

    mov = newMutex();
    cmd = msgQCreate();
    prs = msgQCreate();

    spawnTaskGenerator();
}
```

dove per motivi di spazio la funzione fittizia `spawnTaskGenerator()` si occupa dello spawn degli 8 task periodici.

tasks.h

Il file `tasks.h` contiene i generatori dei task periodici, in quanto, l'utilizzo della funzione `period()` sarebbe non funzionale per la necessità di gestire i log e la durata dei periodi non multipli del secondo e quindi si utilizzano dei task *generatori* con priorità massima.

```
void spawnSatelliteTask();
void generatorDirectionTask();
void generatorMovementTask();
void generatorPhotographTask();
void generatorGeologicalSampleTask();
void generatorAtmosphericPressureTask();
void generatorAltitudeRecordTask();
void generatorTemperatureRecordTask();
void generatorSandStormDetectionTask();
```

A titolo di esempio, mostriamo nel dettaglio qualche generatore:

- Il `generatorDirectionTask()` effettua lo spawn di del task periodico reso possibile con l'utilizzo di `taskDelay()` e la funzione che converte i millisecondi in tick.

```
void generatorDirectionTask(){
    while (true){
        taskDelay(msToTick(T2));
        taskDirection = taskSpawn("Direction", P2, jobDirection);
    }
}
```

- Il `generatorMovementTask()` presenta un offset, quindi si ha un ulteriore `taskDelay()`.

```
void generatorMovementTask(){
    taskDelay(msToTick(PHASE));
    while (true){
        taskDelay(msToTick(T3));
        taskMovement = taskSpawn("Movement", P3, jobMovement);
    }
}
```

- Il metodo `spawnSatelliteTask()` presenta delle particolarità interessanti. Infatti, essendo sporadico, non fa uso di un `taskDelay()`, ma al momento in cui l'utente effettua lo spawn, potrebbe non essere gestito dalla stessa CPU che gestisce gli altri task, perché il suo spawn non viene gestito da un task a cui è stata assegnata una CPU statica. Per questo motivo abbiamo utilizzato una funzione di affinità. Inoltre, notiamo che avendo la priorità più alta, deve fare preemption su tutti gli altri task e pertanto, è presente anche un meccanismo di `boost`; questo deve avvenire prima di assegnarlo alla CPU corretta, altrimenti non farà preemption sugli altri task già in esecuzione.

```
void spawnSatelliteTask(){
    taskPrioritySet(taskIdSelf(), 0); // boost
    cpuset_t affinity;
    CPUSET_ZERO(affinity);
    CPUSET_SET(affinity, 1);
    taskCpuAffinitySet(taskIdSelf(), affinity);
    taskSatellite = taskSpawn("Satellite", P1, jobSatellite);
}
```

  Notiamo che per brevità la funzione `taskSpawn()` non presenta tutti gli argomenti e neanche i vari cast.

- Come già detto in fase di progettazione, il task `Altitude` è *event driven* e si attiva alla ricezione del messaggio da parte del task `AtmosphericPressure`. Per implementare questo meccanismo, abbiamo inizialmente scelto di utilizzare le mailbox dello standard POSIX per utilizzare la funzione `notify()`, che fa lo spwan di un `pthread`. Ma risultava poi di difficile gestione l'assegnazione a questo di una CPU statica e la priorità voluta. Quindi abbiamo usato una message queue di VxWorks, gestendo la ricezione nel generatore.

taskset.h

Il file `taskset.h` definisce i nove job rilasciati dai generatori descritti prima. In queste funzioni vengono gestite le interazioni con i semafori, le mailbox e stampate le transizione nel file di log, oltre a chiamare le entry-point dei vari chunk.

```
int jobSatellite();
int jobDirection();
int jobMovement();
int jobPhotograph();
int jobGeologicalSample();
int jobAtmosphericPressure();
int jobAltitudeRecord(long prs);
int jobTemperatureRecord();
int jobSandStormDetection();
```

A titolo di esempio, mostriamo qualche job:

- Il `jobMovement()`, effettua la ricezione del messaggio della mailbox di VxWorks, inviato da parte del `jobDirection()`, e avvia la missione con `move()` in sezione critica.

```
int jobDirection(){
    loadPositions(); // C21
    computeDest(); // C22
    msgQSend(prs);
    taskDelete(taskIdSelf());
    return 0;
}
```

```
int jobMovement(){
    msgQReceive();
    receiveCmd(); // C31
    semTake(mov);
    pinMode(MUTEX, OUT);
    gpioWrite(MUTEX, HIGH);
    move(); // C32
    semGive(mov);
    gpioWrite(MUTEX, LOW);
    taskDelete(taskIdSelf());
    return 0;
}
```

- Il `jobGeologicalSample()` condivide il semaforo con `jobMovement()`, in quanto, abbiamo scelto di seguire la semplice politica di progettazione che consiste nel fatto che se Resilience si muove, non può trivellare o raccogliere campioni e viceversa. Notiamo che avendo implementato il *priority ceiling protocol*, si effettua un *boost* all'inizio del job.

```
int jobGeologicalSample(){
    taskPrioritySet(taskIdSelf(), mov); // boost
    semTake(mov);
    pinMode(MUTEX, OUT);
    gpioWrite(MUTEX, HIGH);
    drill();  // C51
    collect();  // C52
    semGive(mov);
```

```
    gpioWrite(MUTEX, LOW);
    taskPrioritySet(taskIdSelf(), P5); // deboost
    taskDelete(taskIdSelf());
    return 0;
}
```

entrypoints.h

Il file `entrypoints.h` definisce tutte e dodici le funzioni dei chunk, mostrate nella timeline in figura 4.

```
int transmitData();
int loadPositions();
int computeDest();
int receiveCmd();
int move();
int takePhoto();
int drill();
int collect();
int recordPress();
int recordAltitude();
int recordTemp();
int recordStorm();
```

Mostriamo nel dettaglio qualche funzione:

- La funzione `move()` è il chunk che si occupa di gestire la missione con la libreria `l298n` definita in precedenza.

```
int move(){
    // Turn ON Sensor
    pinMode(MOTORS, OUT);
    gpioWrite(MOTORS, HIGH);

    // Computation Mission
    if (!EOF(mission.txt)){
        switch (mission){
            case FORWARD:
                forward_vehicle();
            case BACKWARD:
                backward_vehicle();
            case LEFT_ROTATE:
                left_rotate_vehicle();
            case RIGHT_ROTATE:
                right_rotate_vehicle();
            case STOP:
                stop_vehicle();
        }
    }
```

```
    // Turn OFF Sensor
    stop_vehicle();
    gpioWrite(MOTORS, LOW);
    return 0;
}
```

Dove, per motivi di spazio, sono state omesse tutte le `busySleep()` e i vari `break`.

- Le altre funzioni, non avendo hardware dedicato, si comportano tutte allo stesso modo.

```
int entrypoint(){
    pinMode(PIN_LED, OUT);
    gpioWrite(PIN_LED, HIGH);
    busySleep(executionTime(BCET, WCET));
    gpioWrite(PIN_LED, LOW);
    return 0;
}
```

Validazione del taskset

Test preliminari

Prima di procedere alla validazione formale del taskset, sono stati condotti dei test su alcuni dettagli implementativi, come la funzione `BusySleep()` e le istruzioni aggiuntive per l'hardware.

Funzione BusySleep

Come descritto nel precedente capitolo, la computazione è stata simulata attraverso la funzione `BusySleep()` della libreria *BusySleep* sviluppata da [9]. Questa funzione attraverso un ciclo con istruzioni del tutto sterili simula una computazione per un certo periodo di tempo.

La corrispondenza tra il numero di cicli da effettuare e la durata desiderata viene trovata in base a dati sperimentali: assumendo che la relazione tra le due grandezze sia di tipo lineare, vengono effettuati un certo numero test con un numero di cicli crescente; i dati ottenuti sono interpolati linearmente in modo da minimizzare l'errore quadratico medio. Questo calcolo viene effettuato dalla funzione `getBusySleepParams()`, che noi abbiamo eseguito partendo da 300.000 cicli, per 30 step; arrivando a coprire fino a 1800 ms, ben oltre i 1100 ms di durata massima di cui abbiamo bisogno da `BusySleep()`.
Una volta ottenuti i parametri della relazione lineare tra numero di cicli e durata, è sufficiente metterli nella funzione `BusySleep()`.

Nella libreria è presente una funzione `testBusySleep()` che va ad eseguire la funzione `BusySleep()` più volte, misurando il tempo impiegato realmente e calcolandone l'errore medio. A differenza di quanto fatto dagli autori, che hanno simulato l'ambiente, noi abbiamo eseguito il codice sul dispositivo Raspberry. Questo si è rivelato estremamente più preciso del simulatore. Infatti mentre gli autori hanno ottenuto errori nell'ordine dei ms, noi abbiamo dovuto modificare la funzione di test, per rilevare errori nell'ordine dei μs. In particolare in media si ottengono poche centinaia di μs di errore, sempre positivo in segno ($errore = misurazione\ reale - durata\ desiderata$, quindi ci mette più del voluto), e con poca variabilità, ovvero per ogni valore della durata della BusySleep l'errore si concentra su pochissimi valori: si va dai $+280\mu s$ per una BusySleep di 1 s ai $+580\mu s$ per una BusySleep di 50 ms.

Hardware

Come visto, alcuni chunk prima e dopo la BusySleep si occupano di accendere e spegnere uno o più LED o i motori. Chiaramente le istruzioni di codice sono poche e molto semplici, tuttavia abbiamo condotto dei test anche su questi aspetti, per mettere in conto dell'overhead dovuto all'hardware.

LED

Nel file `LED_time_test.c` si trova una funzione che consiste in un ciclo che accende e spegne un LED per un certo numero di volte, misurando la durata totale dell'operazione. L'esecuzione del test, con 1.000.000 di ripetizioni, ha impiegato un totale di circa 500 ms, evidenziando così che l'accensione e spegnimento di un LED occupa un tempo del tutto trascurabili (nell'ordine dei decimi di μs).

Motori

Già il risultato sui LED fornisce delle indicazioni anche sui motori, trattandosi pur sempre di sole attivazioni/disattivazioni dei PIN del bus GPIO. Richiedendo però l'attivazione del motore l'attivazione di più PIN e soprattutto uno stack di chiamate a funzione più profondo, abbiamo condotto un test simile anche sui motori (il file di riferimento è `MOTORS_time_test.c`).

Effettivamente le previsioni erano giuste, perché a parità di numero di ripetizioni, il tempo impiegato è stato doppio (circa 1000 ms), ma l'ordine di grandezza per una singola operazione rimane quello dei μs e dunque trascurabile.

Profilazione dei tempi di esecuzione

L'ultimo passo da compiere è rappresentato dal testing, inteso come validazione del taskset. In altre parole si va a verificare che l'implementazione sia effettivamente conforme al modello teorico. Questo passo è supportato dalla teoria delle PTPN tramite la verifica che gli eventi rappresentati dalle transizioni siano avvenuti in istanti temporali conformi con i *firing times* corrispondenti.

Log

L'operazione di validazione può essere condotta dal plugin PetriSimulator di Oris, a patto di fornire in input un file di log con i tempi di avvenimento degli eventi di interesse.

Il file di log può essere ottenuto instrumentando il codice con degli opportuni meccanismi di log. Questi chiaramente introducono dell'overhead, quindi potrebbe essere necessario in fase di simulazione rilassare leggermente i vincoli temporali, intanto però possiamo adottare delle soluzioni che minimizzino questo overhead. Come suggerito in [4], e realizzato da [9], i log vengono messi in una coda FIFO, che noi abbiamo realizzato con una *message Queue* di VxWorks, e solamente al termine salvati su file, essendo questa un'operazione onerosa.

Nella funzione `start()` si inizializza la coda; ad ogni evento di interesse (scadere dell'offset, rilascio di un job, boosting, acquisizione del mutex, ricezione di un messaggio e completamento dell'esecuzione) viene inserita nella coda una variabile `event` che consiste nel nome della transizione corrispondente e dell'istante temporale; infine nella funzione `stop()` viene analizzata la coda, rendendo i tempi relativi all'evento precedente, e salvato il file di log[4].

[4]Proprio per completare quest'ultima operazione è stato necessario modificare la connessione FTP aperta tra host e target per abilitarla alla scrittura (da target a host): è stato sufficiente aggiungere il parametro `-w`

Simulazione

La validazione vera e propria del taskset avviene solitamente tramite la simulazione su Oris della PTPN sulla base della profilazione dei tempi di esecuzione contenuti nel file di log. Purtroppo in questo caso non abbiamo potuto procedere così a causa di un malfunzionamento di Oris. Tuttavia abbiamo provveduto a fare una verifica manuale del log. A causa della modalità manuale l'esecuzione profilata ha avuto una durata relativamente breve, di 45 secondi.

Il codice instrumentato presenta un ulteriore overhead, oltre a quelli già studiati e discussi, dovuto al log (ovvero l'invio dei messaggi alla message queue). Per far fronte alla somma dei vari overhead i firing times sono stati rilassati di 10 ms.

La validazione ha prodotto un risultato positivo, ovvero l'esecuzione del codice ha rispettato il modello PTPN. Questo vuol dire che il modello PTPN usato è congruo.

La verifica manuale del file di log ha permesso anche di identificare con facilità tutte le occorrenze in cui c'è stata una prelazione tra i task[5]. In 45 secondi ci sono state 6 occasioni di prelazioni[6]: in 4 di queste il task interrotto è stato quello di rilevazione delle tempeste di sabbia, nelle altre 2 quello per la misurazione della pressione atmosferica (entrambi a priorità 4); i task che sono sopraggiunti, prendendo la cpu, sono stati quelli relativi al movimento (a priorità 2).

[5]Questa operazione può essere fatta anche quando la validazione avviene tramite Oris, tuttavia solitamente con Oris vengono validati dei log relativi a esecuzioni molto più lunghe

[6]Le prelazioni sono avvenute a: $12,5$ s, $22,5$ s, 25 s, $32,5$ s, $42,5$ s e 45 s

Conclusioni

Questo progetto è stato condotto seguendo il modello a V del ciclo di vita del software. La semplicità e l'efficacia con cui è stato portato a termine mette in luce e amplifica tutti i vantaggi dell'utilizzo del formalismo delle PTPN nella realizzazione di un sistema Real-Time, grazie alla possibilità di essere integrato nel modello a V.

Una volta raccolti i requisiti di sistema e chiarita l'architettura del sistema, la definizione del taskset deriva in maniera molto semplice da essi, costituendo un software design preliminare.

Il design più dettagliato può avvenire tramite la stesura della timeline, che oltre ad essere particolarmente intuitiva, è già una pratica industriale consolidata. A questa deve però necessariamente seguire lo studio del taskset circa la sua schedulabilità. Sicuramente i metodi che esplorano lo spazio degli stati offrono maggiori vantaggi in termini di espressività rispetto ai metodi analitici; e tra questi quello delle PTPN non solo è molto espressivo (nel senso che può modellare taskset molto complessi, con offset, task periodici, sporadici o con jitter, accesso esclusivo alle risorse, vincoli di precedenza e tempi di computazione non deterministici), ma è anche vantaggioso in termini di praticità, sebbene non sia adottato in ambito industriale per la scarsa intuitività.

Il modello PTPN che modella il taskset infatti può essere ottenuto algoritmicamente dal modello in forma di timeline, permettendo così di trarre vantaggio da entrambe le forme. Infatti la PTPN può essere analizzata tramite Oris, sia per enumerare lo spazio degli stati, sia per estrarre le tracce di esecuzione di ogni task in modo da verificare se sia possibile schedularlo correttamente, senza che sfori la deadline.

Inoltre anche il codice Real-Time può essere ottenuto algoritmicamente partendo dalla PTPN, risultando quindi un codice ben strutturato e maggiormente leggibile, lasciando allo sviluppatore la sole responsabilità delle funzioni entry-point. Questo è un aspetto importante affinché un approccio del genere possa essere accettato nell'industria, dal momento che aiuta la comprensione del modello PTPN.

Dopo aver supportato le fasi di progettazione e implementazione, l'uso del modello PTPN supporta anche la fase di testing, inteso come validazione del taskset, grazie alla simulazione della PTPN sulla profilazione dei tempi di esecuzione.

In conclusione la realizzazione in questo progetto di un taskset implementato in un "vero" sistema (inteso come un sistema "fisico", non simulato, dotato di hardware), dotato di un RTOS industriale come VxWorks, dimostra che l'uso dei modelli PTPN si può applicare con successo anche in ambiti industriali, pratici, e non solo a livello formale come oggetto di ricerca.

Bibliografia

Riferimenti

[1] ORIS tool (v. 1.0). `https://stlab.dinfo.unifi.it/oris1.0/`.

[2] G. Bucci, A. Fedeli, L. Sassoli, and E. Vicario. Modeling Flexible Real Time System with Preemptive Time Petri Nets. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS03)*, Luglio 2003.

[3] G. Bucci, A. Fedeli, and E. Vicario. Timed State Space Analysis of Real-Time Preemptive System. *IEEE Transaction on Software Engineering*, 30(2):97–111, Febbraio 2004.

[4] L. Carnevali, L. Ridi, and E. Vicario. Putting Preemptive Time Petri Nets to Work in a V-Model SW Life Cycle. *IEEE Transaction on Software Engineering*, 37(6):826–844, 2011.

[5] Laura Carnevali. *Formal methods in the development life cycle of real-time software*. PhD thesis, Università degli Studi di Firenze, 2010.

[6] Angelo D'Amante and Kevin Maggi. raspberry-vxworks. `https://github.com/AngeloDamante/vxWorks-rb3plus-blink-test`, 2021.

[7] Angelo D'Amante and Kevin Maggi. rover-raspberry-vxworks. `https://github.com/AngeloDamante/rover-Raspberry-VxWorks`, 2021.

[8] Kevin Maggi. Progettazione e sviluppo di un componente software per la derivazione di modelli di Petri Preemptive da specifiche timeline, 2020.

[9] Andrea Puccia, Lorenzo Nuti, and Lorenzo Mandelli. Esame di software engineering for embedded system, 2021.

[10] Linux Wiki. Rpi gpio code samples, 2020.

[11] Wikipedia. Vxworks — wikipedia, l'enciclopedia libera, 2021. [Online; in data 15-luglio-2021].

Documentazione VxWorks

[12] WindRiver. API Reference: Application Core OS Test.

[13] WindRiver. Application Programmer's Guide.

[14] WindRiver. VxWorks 7 BSP for Raspberry Pi 3B/3B+. `https://github.com/Wind-River/vxw7-bsp-raspberry-pi`.

[15] WindRiver. VxWorks 7 SDK for Raspberry Pi 3B. https://labs.windriver.com/downloads/wrsdk-vxworks7-docs/README-raspberrypi3b.html.

[16] WindRiver. VxWorks Drivers API Reference.