Università degli Studi di Milano Bicocca
Scuola di Scienze
Dipartimento di Informatica, Sistemistica e Comunicazione
Corso di laurea in Informatica

RTEMS su Raspberry Pi per applicazioni real-time

Relatore: Prof. Domenico Giorgio Sorrenti
Co-relatore: Ing. Fabrizio Bernardini
Tutor aziendale: Prof. Pietro Braione

Relazione della prova finale di:
Clark Ezpeleta
Matricola 832972

Anno Accademico 2019-2020

Indice

Introduzione

Il lavoro che ho svolto ha due obiettivi principali: il 'porting' di RTEMS su Raspberry Pi e la creazione di applicativi RTEMS per validare il corretto funzionamento delle interfacce GPIO, UART, I2C, SPI e l'utilizzo degli interrupts tramite le API di RTEMS.
Per poter svolgere il 'porting' ho fatto riferimento al RTEMS User Manual[1], per comprendere meglio i concetti di RSB(RTEMS Source Builder) e BSP (board support packages), ed ho utilizzato la guida fornita da ing. Basile [2], di BIS-Italia, che raccoglie tutti i passaggi esposti sul blog di Alan Tech per il 'porting' della versione 4.11.
Purtroppo i passaggi illustrati sul blog non sono totalmente corretti, poiché sono per il 'porting' della versione di RTEMS precedente a quella che ho utilizzato, cioè la 5.1 che è la più recente e stabile.
Il 'porting' può essere definito corretto se alla fine della procedura si riesce a caricare uno degli applicativi di test forniti da RTEMS, su Raspberry Pi senza errori.
Dopo aver effettuato correttamente il 'porting', ho creato una guida in italiano che raggruppa tutti i passaggi effettuati integrando le correzioni necessarie, ed è stata corretta anche la guida di ing. Basile [3].
Dopo aver impostato l'ambiente di lavoro ho iniziato a creare gli applicativi RTEMS da eseguire sulla Raspberry Pi.
Per poter creare gli applicativi RTEMS ho dovuto familiarizzare con il linguaggio C, leggere l'RTEMS Classic API Guide [4] ed analizzare i codici sorgente di esempio trovati nel git repository di asuol[5] per poter comprendere l'utilizzo delle API.
Tutto il lavoro è stato eseguito in modalità "smart-working", per questo motivo mi è stata messa a disposizione da BIS-Italia la scheda Raspberry Pi 3B+ per poter effettuare l'attività. Inoltre Microchip Technologies ha gentilmente offerto dei componenti aggiuntivi utili per gli applicativi RTEMS di test dell'interfaccia I2C e SPI.
Oltre all'attività software ho dovuto eseguire una piccola attività hardware, cioè creare dei circuiti saldando i vari componenti e utilizzando la breadboard in modo da poterli collegare alla Raspberry Pi, per fare ciò ho seguito gli schemi elettrici che mi ha inviato ing. Bernardini e ho letto i datasheet dei componenti dell'interfaccia I2C [6] [7] e SPI [8] [9] in modo da comprendere il loro funzionamento e poter creare i driver.

Premesso tutto ciò ho suddiviso la mia relazione nei seguenti capitoli:

- **Capitolo 1** - in questo capitolo vengono descritte le principali tecnologie utilizzate durante il mio lavoro. Innanzitutto viene descritto RTEMS che è il sistema operativo su cui si basa tutta l'attività, dopodiché viene descritta la Raspberry Pi su cui verranno eseguiti gli applicativi RTEMS, ed infine viene descritto Eclipse che è l'IDE utilizzato per creare gli applicativi.
- **Capitolo 2** - in questo capitolo viene descritta tutta l'attività di 'porting' e la definizione della toolchain per poter realizzare applicativi RTEMS. Vengono esposte tutte le problematiche rilevate e le loro soluzioni.
- **Capitolo 3** - in questo capitolo vengono descritti le API che RTEMS ha a disposizione, la loro struttura e in che modo li ho testati e per quali motivi si è scelto di testarli.
- **Capitolo 4** - in questo capitolo viene descritta l'attività software che si basa sulla creazione degli applicativi RTEMS per testare il corretto funzionamento delle API di RTEMS. Viene esposto anche una descrizione dei componenti aggiuntivi offerti da Microchip di cui ho creato i driver per poterli utilizzare con RTEMS.
- **Capitolo 5** - in questo capitolo viene descritto ciò che si è raggiunto e la possibile estensione del mio lavoro.

1 Tecnologie

1.1 RTEMS

RTEMS sta per **Real-Time Executive MultiProcessor System** ed è un sistema operativo real-time (RTOS) general purpose open source (licenza GPL 2.0 modificata) progettato e gestito da OAR Corporation. Il suo sviluppo iniziò nella fine degli anni 80 utilizzando i linguaggi Ada e C, e venne usato inizialmente per scopi militari, invece le prime versioni utilizzabili sono state rese open-souce su server ftp nel 1994.
Attualmente viene utilizzato in molti settori tra cui quello aerospaziale, infatti è stato utilizzato in alcune missioni spaziali sia a livello di on-board computer che come computer embedded in altre attività di volo.
RTEMS è stato **validato dall'ESA**, European Space Agency, ciò vuol dire che sono stati scritti dei programmi che riproducono gli scenari critici (ad esempio la gestione di molti dispositivi oppure la gestione e l'esecuzione concorrente dei task), e sono stati eseguiti sul sistema operativo per poter verificare il suo corretto funzionamento in quei casi. La validazione viene tutt'ora aggiornata poiché è un sistema operativo che è in continua evoluzione e avrà più funzionalità con il passare del tempo.
RTEMS non è un sistema operativo a sé stante, usato per caricare altri programmi, ma **è un executive** che viene compilato con l'applicazione in un unico codice monolitico da eseguire.

RTEMS può anche essere visto come un insieme di direttive raggruppate in una serie di **manager**, che si occupano di varie funzionalità tra cui il controllo e la sincronizzazione dei task e processori, la gestione della memoria e la mutua esclusione. Invece la gestione dello scheduling, dispatching e object management sono forniti dal executive core.

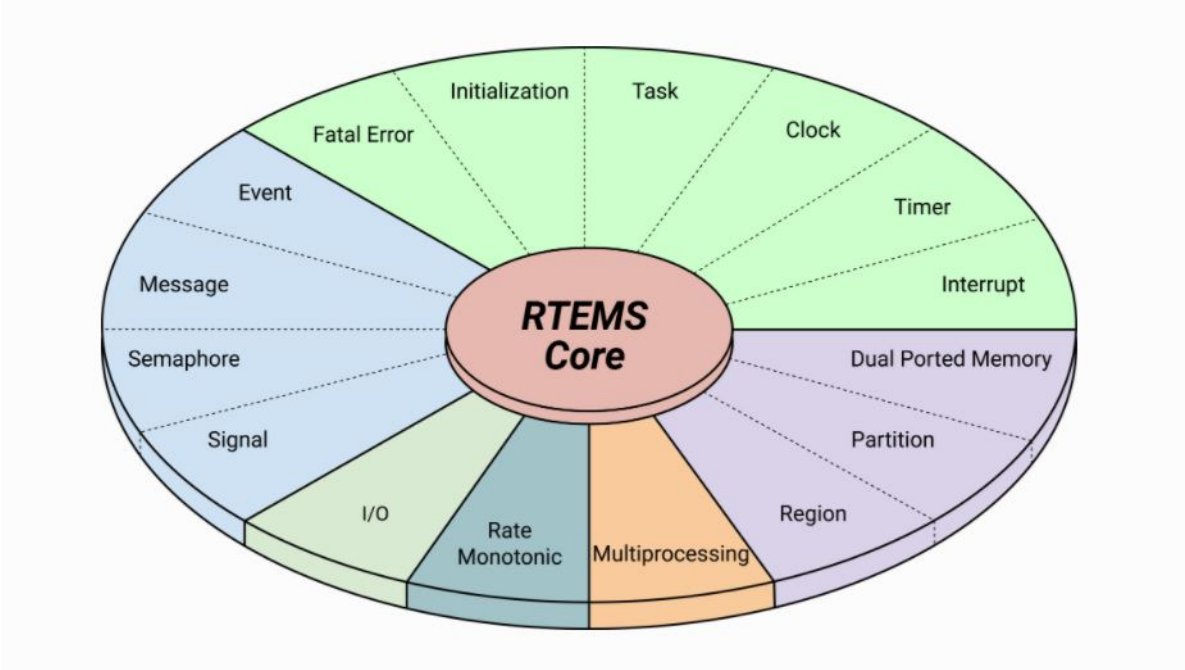


Figura 1.1: Architettura RTEMS

Utilizzando i managers di RTEMS lo sviluppatore può concentrarsi al solo sviluppo del applicativo e ciò riduce notevolmente il tempo di sviluppo.
La figura successiva mostra la logica di utilizzo di RTEMS.

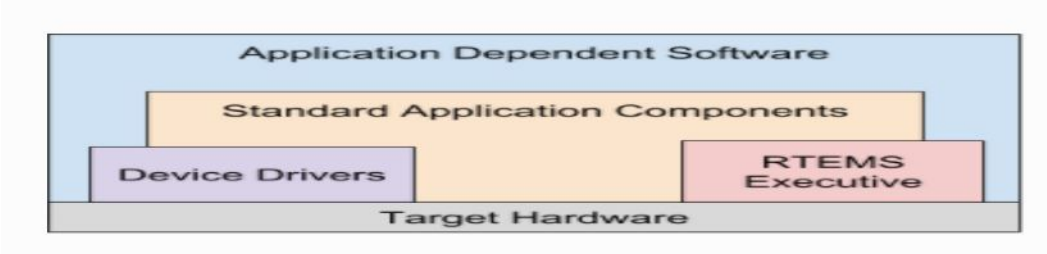
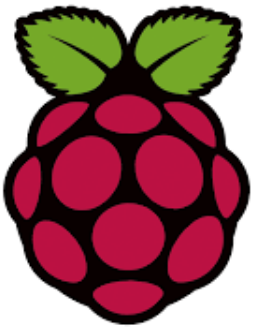


Figura 1.2: Struttura applicativo RTEMS

Come si può notare RTEMS Executive è un intermediario tra il codice dell'applicativo e il target hardware, invece le dipendenze hardware con altri device sono localizzati nel livello 'device drivers'. Il RTEMS I/O manager incorpora queste dipendenze hardware nel sistema mentre allo stesso momento fornisce all'application code l'accesso ad esse. Queste dipendenze hardware

sono isolate in specifiche **BSP**, Board Support Packages, per questo motivo il 'porting' di un applicativo RTMES su altri processori è semplice poiché basterebbe selezionare la BSP del microprocessore su cui si vuol eseguire l'applicativo e compilare con le sue librerie.
In questo modo durante lo sviluppo di un applicativo real-time si ha la totale indipendenza dall'architettura dei microprocessori.
E' disponibile il 'porting' di RTEMS su molte architetture CPU tra cui **ARM**, MIPS, LEON,ERC32 e i PowerPC; durante il mio lavoro ho trattato il 'porting' su architettura ARM utilizzando una Raspberry Pi 3B+.

1.2 Raspberry Pi



Raspberry Pi è una serie di computer a scheda singola sviluppata da Raspberry Pi Foundation in collaborazione con la Broadcom, in Inghilterra.
Originariamente è stato usato per insegnare le basi dell'informatica nelle scuole e nei paesi in via di sviluppo, ma attualmente grazie al suo basso prezzo viene usato anche nel settore della robotica, domotica e in molti altri. Al momento sono state rilasciate quattro generazioni di Raspberry Pi, e tutti i modelli hanno un Broadcom SoC (System on Chip) con processore ARM integrato e on-chip GPU (Graphics Processing Unit).
Durante il mio lavoro viene usata la **Raspberry Pi 3B+** che utilizza il Broadcom BCM2837B0 SoC [10] con processore Cortex-A53 (ARMv8) 64-bit 1.4Ghz.

Figura 1.3: Scheda Raspberry Pi 3B+

Tra le specifiche tecniche quelle che ci interessano maggiormente sono:

- SDRAM LPDDR2 da 1GB.
- supporto per la micro SD.
- accesso a 40 GPIO.

La scheda micro SD deve essere configurata nel seguente modo:

1. copiare il firmware di Raspberry Pi compatibile con RTEMS nella memoria SD.
2. cancellare i file 'kernel*.img'.
3. compilare i sorgenti di un applicativo RTEMS in modo da ottenere il file con estensione'.img'
4. copiare l'eseguibile RTEMS nella memoria SD.
5. impostare l'eseguibile RTEMS come kernel della scheda Raspberry Pi, modificando il file 'config.txt' aggiungendo il comando "kernel = nome_eseguibile.img"

Questi passaggi sono necessari perché il 'bootloader' di Raspberry Pi 3B+ è preimpostato in modo che, quando viene accesa la scheda, cerca un file denominato "kernel7.img" per caricare il sistema operativo. Nel mio caso non viene caricato un sistema operativo ma un eseguibile RTEMS, che quando avviato prende il controllo della scheda.

La figura successiva mostra tutti i pin GPIO che Raspberry Pi 3B+ ha a disposizione:

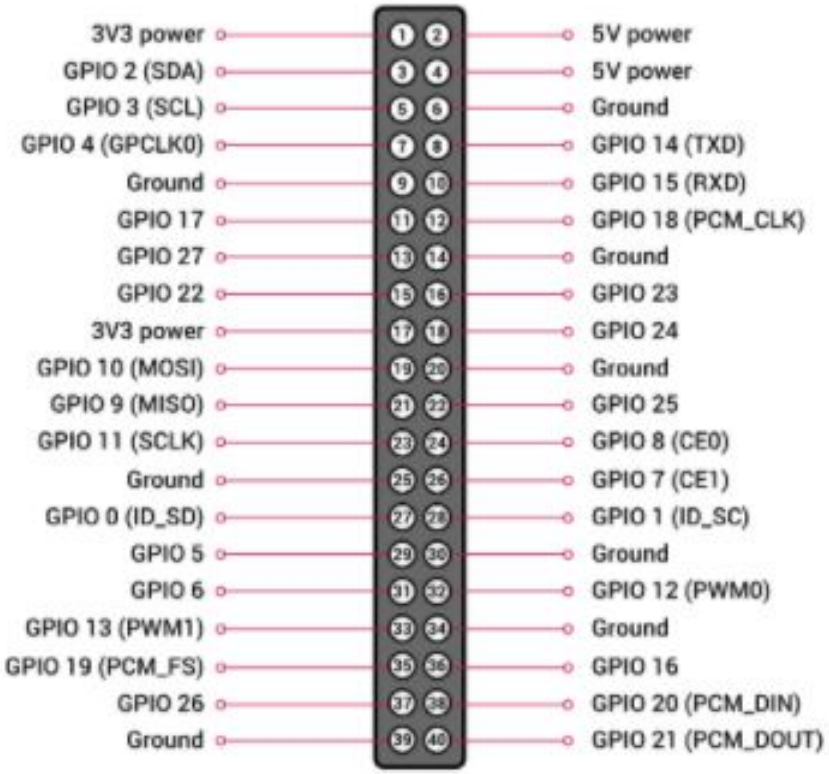


Figura 1.4: Raspberry Pi 3B+ GPIO

Nella scheda i livelli logici sono così definiti : alto se la tensione è pari a 3V3 e ciò corrisponde al valore binario 1, basso se la tensione è pari a 0V e ciò corrisponde al valore binario 0.
I pin configurabili come input sono 3V3 'tolerant', quindi ricevono in input una tensione che va da 0v a 3V3, ed ogni pin ha una resistenza di pull-down o pull-up che è possibile impostare da software. I pin configurabili come output possono dare una tensione da 0V a 3V3.
Infine alcuni pin possono essere utilizzati anche per gestire altre interfacce come l'I2C, UART, SPI.

1.3 Eclipse

Eclipse è un IDE open source rilasciato con licenza EPL (Eclipse Public License) creato da IBM usato principalmente per la programmazione in Java ma grazie a vari plugin può essere usato per altri linguaggi di programmazione come C, C++, COBOL, Python e molti altri.
L'ambiente di sviluppo Eclipse include l'Eclipse Java development tools per Java, e l'Eclipse CDT per C/C++.
Per utilizzare RTEMS gcc cross compiler su Eclipse C, bisogna installare il plugin di RTEMS e settare le variabili di ambiente.

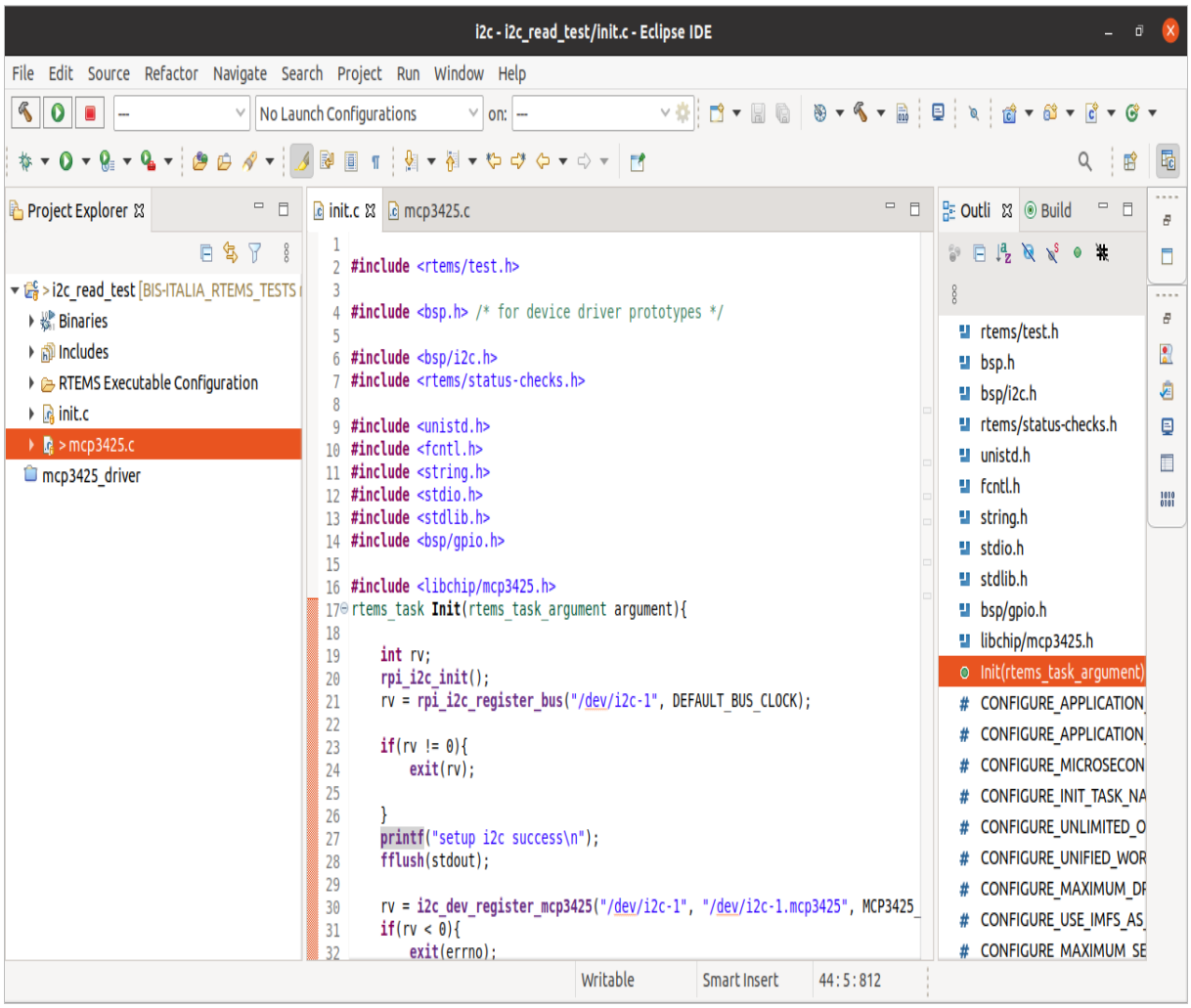


Figura 1.5: Schermata Eclipse del programma I2C

2 Porting di RTEMS su Raspberry Pi

L'attività di 'porting' è la prima fase del lavoro e consiste in:

1. Installazione dalla tool-suite sul computer host, dove vengono creati gli applicativi RTEMS.
2. Verifica del corretto funzionamento della tool-suite, eseguendo i programmi di test su Raspberry Pi
3. Configurazione del IDE Eclipse C, per la creazione di eseguibili RTEMS
4. Provare a compilare e eseguire i programmi sia da terminale, e sia da Eclipse C

Per svolgere tutta la procedura mi sono munito di un computer con sistema operativo Ubuntu, una scheda Raspberry Pi 3B+, una micro SD, e convertitore TTL USB.
RTEMS in sé è complesso per questo il team di RTEMS ci fornisce "l'ecosistema di RTEMS" che è una collezione di strumenti, packages, codici sorgente e documentazione, utile per definire come sviluppare, mantenere e usare RTEMS.
Durante il 'porting' ho utilizzato due importanti strumenti che fanno parte dell'ecosistema RTEMS, e sono :

- **RTEMS RSB** : l'RTEMS Source Builder è tool molto utile per compilare e fare la 'build' dei moduli di RTEMS e delle BSP
- **BSP raspberrypi** : la Build Support Package è il codice di supporto, che contiene le librerie di RTEMS per una specifica scheda (ad esempio la BSP raspberrypi viene usata per la Raspberry Pi 3B+)

L'installazione della tool-suite e BSP viene fatta tutta tramite comandi su terminale, non esiste un eseguibile .exe, ma viene usato il RTEMS RSB.
Ci sono due requisiti necessari da rispettare e sono, avere installato Git per scaricare il codice sorgente di RTEMS, ed essere in grado di compilare i programmi C/C++ e Python poiché molti moduli della tool-suite sono scritti con questi linguaggi.
La versione di cui bisogna effettuare il porting è la **5.1** che è la più recente e stabile, anche se ciò ha dato problemi durante l'installazione perché la guida a cui facevo riferimento era per la versione precedente 4.1 rilasciata nel 2015, e

veniva clonato il ramo main dove attualmente è presente una versione ancora in sviluppo.
Grazie all'aiuto dell'ing. Basile, siamo riusciti a capire che il problema stesse nel comando git che clonava la versione errata.
La BSP raspberrypi da a disposizione dei programmi .exe da compilare e eseguire sulla scheda, in modo da verificare che l'installazione sia stata effettuata con successo.
Anche in questo caso ho avuto un problema riguardo la versione del firmware di Raspberry Pi copiato sulla scheda SD che non era compatibile con RTMES.
Per creare i file sorgenti per un applicativo RTEMS si può usare un semplice editor di testo per poi compilarli con il 'gcc cross compiler' di RTEMS tramite terminale, oppure utilizzare l'IDE Eclipse C/C++ per semplificare il processo.
Nel IDE ho dovuto installare un plugin per supportare RTEMS e impostare le variabili d'ambiente, in modo da 'puntare' al gcc cross compiler di RTEMS e la BSP corretta.
Come primo programma ho voluto creare un 'hello world', che manda tramite UART il messaggio su terminale. Durante la creazione ho avuto molti problemi dato che è stata la prima volta che creavo un applicativo RTEMS, infatti l'applicativo non mandava il log come ci si aspettava. Il problema risiedeva nella configurazione del sistema che inizializzava il driver del clock che per il nostro caso era inutile, e il driver della console errato.
Sono riuscito a compilare correttamente i file sorgenti sia da terminale che da Eclipse, e ciò è verificato poiché il programma eseguito sulla Raspberry Pi non da errori.
A questo punto del lavoro ho creato una guida che espone tutti i passaggi corretti, cosicché qualunque utente voglia approcciarsi ad RTEMS su Raspberry Pi per la prima volta riesca a fare il porting e senza dover cercare tra le tante fonti sparse su internet.
La guida è disponibile in appendice alla relazione

3 Integrazione hardware e software

Effettuato il 'porting' sono passato all'attività software, cioè la creazione di RTEMS executive per testare il funzionamento delle RTEMS Classic API che sono le API per l'utilizzo delle interfacce.

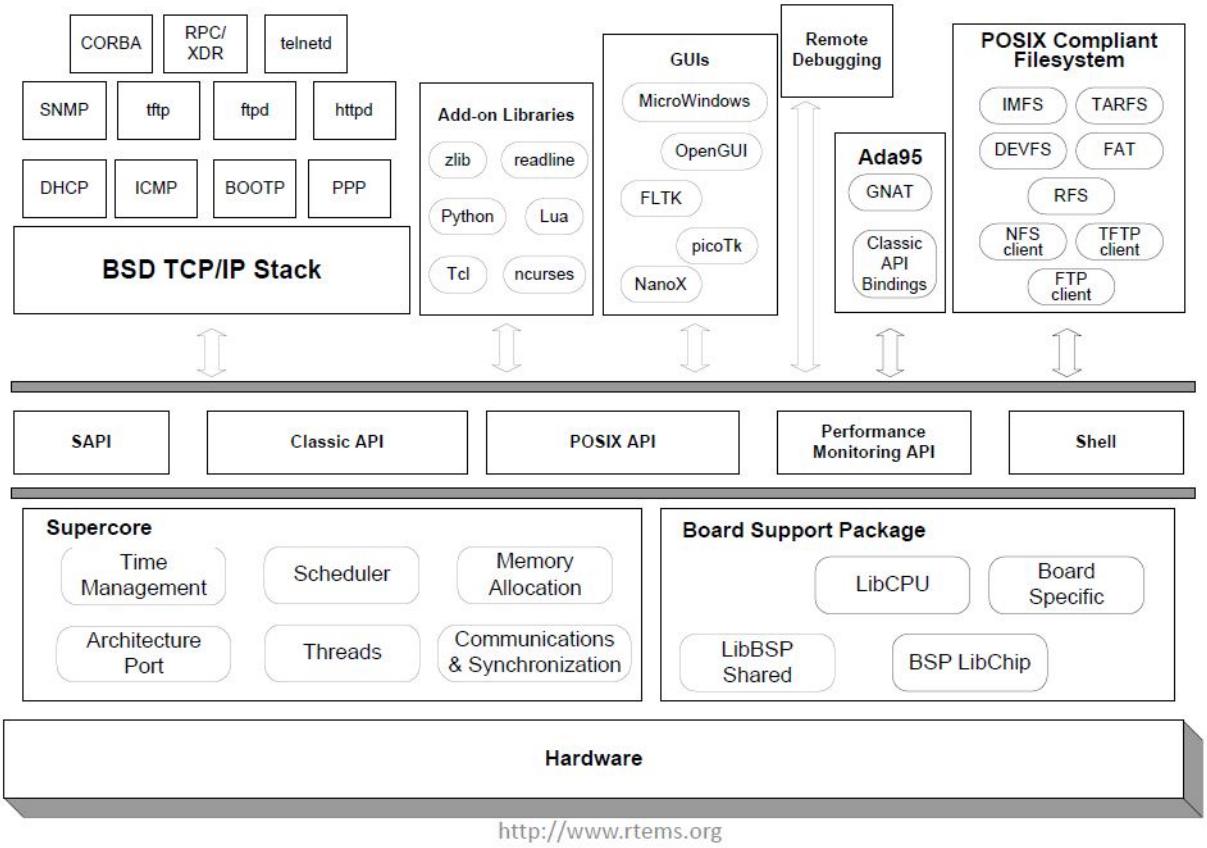


Figura 3.1: Architettura dettagliata sistema RTEMS

Le interfacce che ci interessano validare sono:UART, GPIO, I2C, SPI.
La BSP dà a disposizione API e drivers per interfacciarci alle interfacce che ci interessano.

La figura successiva mostra la struttura ed alcuni driver, file di configurazione e librerie presenti nella BSP utili per l'attività sperimentale:

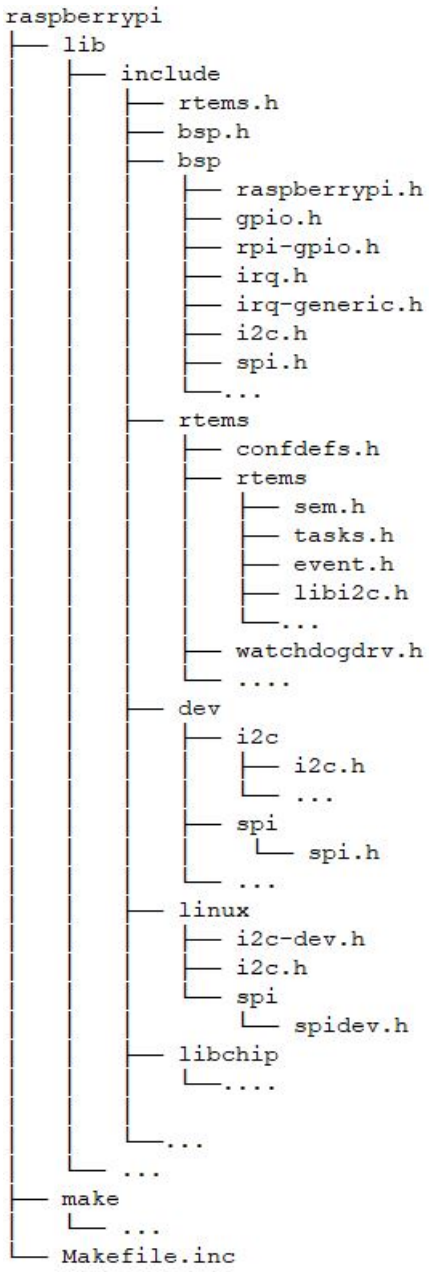


Figura 3.2: Struttura file BSP

- rtems.h : serve ad interfacciarsi ed includere tutte le RTEMS Classic API
- bsp.h : definisce le macro per la Rasperry Pi
- bsp/raspberrypi.h : definisce i registri della Raspberry Pi
- bsp/gpio.h : definisce le API per la gestione dei GPIO
- bsp/rpi-gpio.h : definisce le API per i GPIO specifiche per la Raspberry Pi
- bsp/irq.h : definisce le macro per utilizzare gli interrupt
- bsp/irq_generic.h : definisce le API per la gestione degli interrupts
- bsp/i2c.h : definisce le API per I2C specifiche per la Raspberry Pi
- bsp/spi.h : definisce le API per SPI specifiche per la Raspberry Pi
- rtems/confdefs.h : serve a configurare il sistema su cui gira l'applicativo
- rtems/rtems/sem.h : definisce le API per gestire i semafori
- rtems/rtems/tasks.h : definisce le API per la gestione dei task
- rtems/rtems/event.h : definisce le API per la gestione degli eventi
- dev/i2c/i2c.h : definisce l'I2C driver framework ed è compatibile con la Linux I2C user-space API
- dev/spi/spi.h : definisce il SPI driver framework ed è compatibile con la Linux SPI user-space API

L'elenco sopra rappresenta è solo una **piccola estrazione di cio che la BSP contiene**

4 Attività sperimentale

4.1 Obiettivi

L'attività sperimentale consiste nella creazione di applicativi RTEMS per validare le API di RTEMS. Un API può essere considerata valida se l'applicativo, eseguito sulla scheda, svolge correttamente ciò che si è programmato e nell'arco temporale definito.

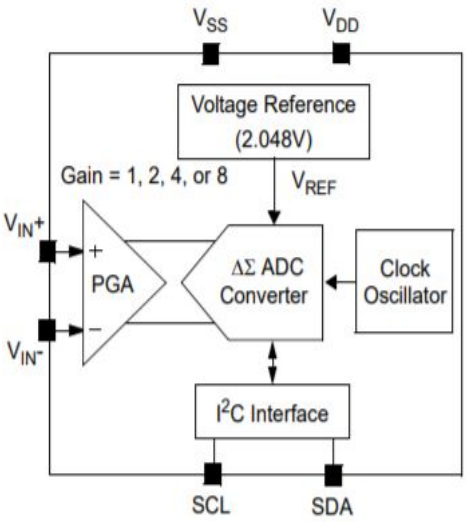
4.2 Test set-up

Ogni programma di test creato implica un circuito elettrico differente costruito con jumper, breadboard, resistenze, condensatori, pulsanti, LED e schede aggiuntive fornite dalla Microchip su cui ho dovuto saldare cavetti e componenti per poterli utilizzare e collegare alla Raspberry Pi.
I componenti aggiuntivi sono :

- **MCP3425 SOT23-6 Evaluation Board** [6]: un evaluation board, con un ADC (Analog Digital Converter) MCP3425, controllato tramite protocollo I2C
- **MSOP-10 and MSOP-8 Evaluation Board** [9]: un evaluation board generica che serve per il collegamento del componente MCP4822 alla Raspberry Pi
- **MCP4822** [8]: un DAC (Digital Analog Converter) controllato tramite protocollo SPI, montato sulla evaluation board sopracitata.

Il componente **MCP3425** è un ADC, 'Analog Digital Converter', da 16 bit a canale singolo e vengono passati come input due tensioni ai pin V_{IN}^{+} e V_{IN}^{-} e viene restituito come output una tensione pari a $V_{IN}^{+} - V_{IN}^{-} \times PGA$ ('Programmable Gain Amplifier') sul pin SDA (linea dati).

(a) MCP3425



(b) schema a blocchi di MCP3425

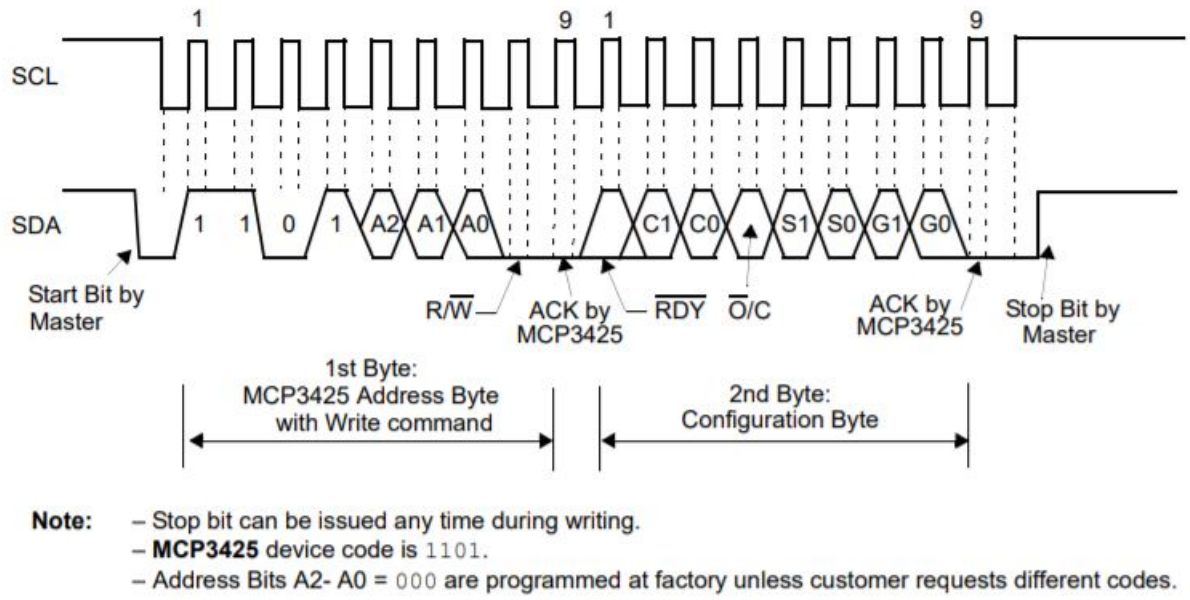
Il componente ha la seguente configurazione predefinita:

- Programmable Gain Amplifier(PGA) = 1
- Continuos Conversion
- Programmable Data Rate = 12 bit

Durante il lavoro viene utilizzata questa configurazione, ma nel caso si volesse cambiare la configurazione del componente, si può eseguire una scrittura di due byte dove il primo byte rappresenta l'indirizzo di periferica e il bit R/W (nel nostro caso è 11010000, dove R/W = 0 significa che stiamo eseguendo una scrittura) invece il secondo byte contiene i bit di configurazione

R/W-1	R/W-0	R/W-0	R/W-1	R/W-0	R/W-0	R/W-0	R/W-0
$\overline{RDY}$	C1	C0	$\overline{O}$/C	S1	S0	G1	G0
1 *	0 *	0 *	1 *	0 *	0 *	0 *	0 *
bit 7							bit 0

Figura 4.2: Registro di configurazione MCP3425 con la configurazione preimpostata



Note: – Stop bit can be issued any time during writing.
– **MCP3425** device code is 1101.
– Address Bits A2- A0 = 000 are programmed at factory unless customer requests different codes.

Figura 4.3: Diagramma temporale scrittura su MCP3425

Su questo componente ci interessa effettuare la lettura della tensione in uscita che è rappresentata da due byte. Per eseguire la lettura bisogna seguire questi passaggi :

1. mandare sul bus I2C l'indirizzo di periferica con R/W pari a 1
2. lettura dei due byte
3. conversione dei byte per avere il valore effettivo della tensione

Dato che utilizziamo la configurazione preimpostata, abbiamo una conversione di 12 bit del dato, ciò comporta che nei due byte che riceviamo il dodicesimo bit rappresenta il segno e viene ripetuto fino al sedicesimo bit, quindi non dovremo considerare gli ultimi quattro bit. Discorso analogo per la conversione in 14 bit e 16 bit.

Conversion Option	Digital Output Codes
16-bits	D15 ~ D8 (1st data byte) - D7 ~ D0 (2nd data byte) - Configuration byte. **(Note 1)**
14-bits	MMD13 ~ D8 (1st data byte) - D7 ~ D0 (2nd data byte) - Configuration byte. **(Note 2)**
12-bits	MMMMD11 ~ D8 (1st data byte) - D7 ~ D0 (2nd data byte) - Configuration byte. **(Note 3)**

Figura 4.4: Output MCP3425

Il componente **MCP4822** è un DAC, 'Digital Analog Converter', da 12 bit a due canali e come input viene passato da programma 2 byte in cui viene rappresentato il dato da convertire e come output una tensione pari a $V_{OUT} = \frac{V_{ref} \times D_n}{2^n} \times G$ dove:

- V_{ref} = 2.048V
- D_n = sono i bit che passo in input
- n = è il numero dei bit, cioè 12
- G = è il gain

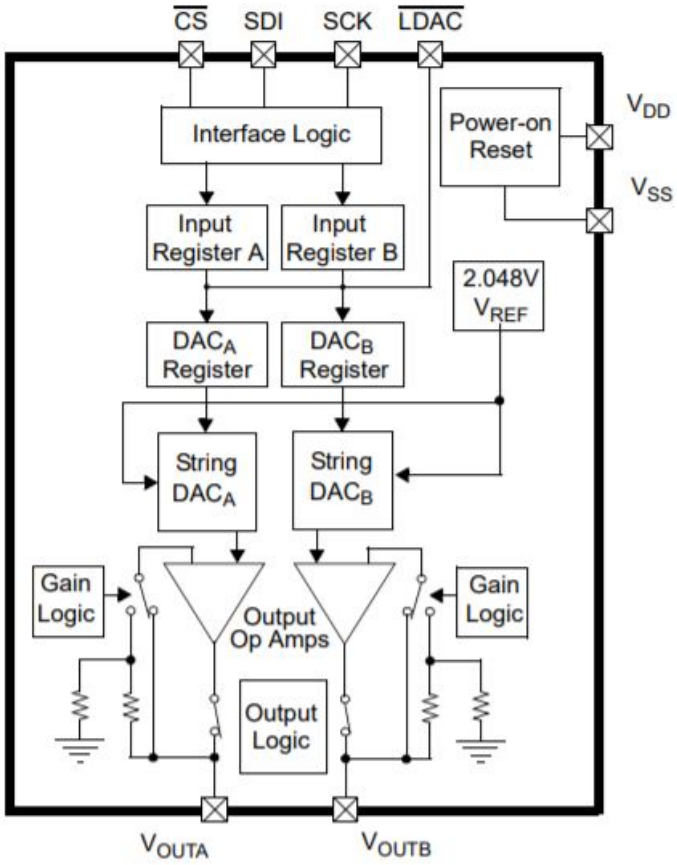


Figura 4.5: Schema a blocchi di MCP4822

Il componente ha due canali e la comunicazione è unidirezionale, cioè non puo' essere eseguita una lettura ma solo la scrittura

La scrittura viene effettuata mandando due byte di cui il primo è formato dai 4 bit di configurazione e gli ultimi 4 data bit, e il secondo è formato dai restanti data bit.

W-x	W-x	W-x	W-0	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x
$\overline{A/B}$	—	$\overline{GA}$	$\overline{SHDN}$	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
bit 15															bit 0

Figura 4.6: Registro di configurazione del MCP4822

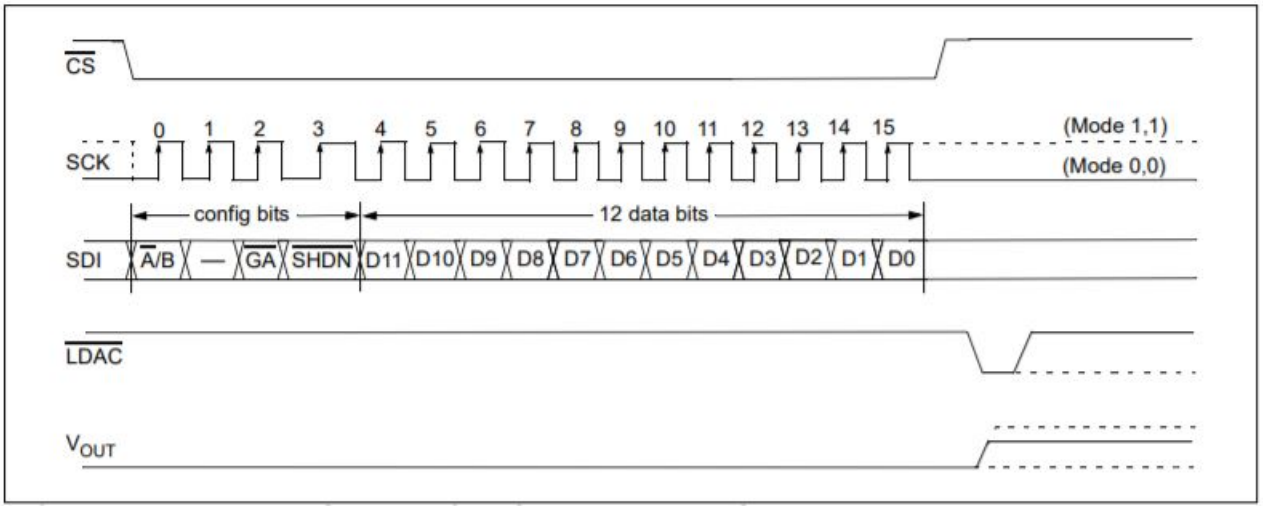


Figura 4.7: Diagramma temporale scrittura su MCP4822

Effettuata la scrittura sul canale selezionato, il componente procede alla conversione del dato e restituisce la tensione.

4.3 Test application software

La struttura principale che tutti i programmi seguono è la seguente:

- **init.c**: è il file sorgente contenente la funzione Init che è il punto di partenza dell'eseguibile. E' come se fosse il metodo main in Java.
- **init.h**: è un file header che contiene tutte le direttive RTEMS per configurare il sistema
- **task_helper.c**: è il file sorgente contenente la definizione dei task, le funzioni per la manipolazione delle variabili globali, le funzioni aggiuntive
- **task_helper.h**: è un file header che contiene tutte le dichiarazioni delle funzioni definite in task_helper.c che si vogliono usare in init.c, ad esempio la definizione dei task.

La configurazione del sistema viene definita tramite le direttive RTEMS, si può definire una configurazione di base :

```
#define CONFIGURE_APPLICATION_DOES_NOT_NEED_CLOCK_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_SIMPLE_CONSOLE_DRIVER
#define CONFIGURE_UNLIMITED_OBJECTS
#define CONFIGURE_UNIFIED_WORK_AREAS
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_INIT

#include <rtems/confdefs.h>
```

- CONFIGURE_APPLICATION_DOES_NOT_NEED_CLOCK_DRIVER: è necessario definirlo quando non si vuole utilizzare il driver del clock e del timer. Nel caso si voglia inizializzare il driver del clock bisogna utilizzare la direttiva CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER, e impostare i microsecondi per tick con la direttiva CONFIGURE_MICROSECONDS_PER_TICK.
- CONFIGURE_APPLICATION_NEEDS_SIMPLE_CONSOLE_DRIVER: imposta il Simple Console Driver, in questo modo possiamo utilizzare la funzione "printf()" per la visualizzazione dei log.
- CONFIGURE_UNLIMITED_OBJECTS: con questa direttiva non si ha limiti di oggetti rtems, e non si ha il bisogno di gestire la RAM. Ovviamene questa direttiva non deve essere usata per applicativi da mandare in produzione, ma può essere usata durante la fase di test

- CONFIGURE_UNIFIED_WORK_AREAS: da utilizzare quando la direttiva ”CONFIGURE_UNLIMITED_OBJECTS” è definita poiché in questo modo viene utilizzata tutta la memoria disponibile, cioè il RTEMS Workspace e il C programm heap saranno in un unico pool di memoria.
- CONFIGURE_RTEMS_INIT_TASKS_TABLE: questa direttiva è necessaria per indicare che l'applicativo parte eseguendo il task Init che ha la priorità massima (1) e non è preemptive.
- CONFIGURE_INIT: direttiva necessaria per far si che la libreria **rtems/confdefs.h** (che deve essere inclusa nel file header) istanzi la configurazione del sistema

Il primo programma di test che ho creato è quella per l'**interfaccia UART** e si tratta di un ”hello world!”. È stato il più semplice da creare, poiché durante il 'porting' è stato usato un programma di esempio simile per visualizzare i log di sistema. Dal codice sorgente dei programmi di esempio forniti da RTEMS, come quelli per il ticker.img, si può ipotizzare correttamente che per far stampare un messaggio sulla console è sufficiente definire la direttiva di sistema

```
#define CONFIGURE_APPLICATION_NEEDS_SIMPLE_CONSOLE_DRIVER
```

e usare le funzioni della libreria standard stdio.h

```
printf("log");    fflush(stdout);
```

ed ho come risultato:

```
RTEMS RPi 3B+ 1.3 (1GB) [00a020d3]


*** HELLO WORLD TEST ***
Hello World
*** END OF HELLO WORLD TEST ***

*** FATAL ***
fatal source: 5 (RTEMS_FATAL_SOURCE_EXIT)
fatal code: 0 (0x00000000)
RTEMS version: 5.0.0.61ccb9c05dcd695114541960aa6bfc1315f30514
RTEMS tools: 7.5.0 20191114 (RTEMS 5, RSB 5 (46e9d4911f09 modified), Newlib 7947581)
executing thread ID: 0x08a010001
executing thread nam
```

Figura 4.8: Hello world log

Il messaggio di errore su terminale è causato dalla funzione **'exit(n)'**, presente nel task Init per **terminare l'applicativo** con 'fatal code' uguale al parametro 'n'. La funzione 'exit(n)' causa un messaggio di errore perché si prevede che un applicativo RTEMS sia sempre in esecuzione, e quindi il termine di esso per RTEMS è considerato un comportamento imprevisto. Se non si vuole avere questo messaggio di errore bisogna inserire nel codice un ciclo infinito, in modo che il programma non termini.

L'**interfaccia GPIO** è stata validata con tre programmi per coprire le seguenti situazioni :

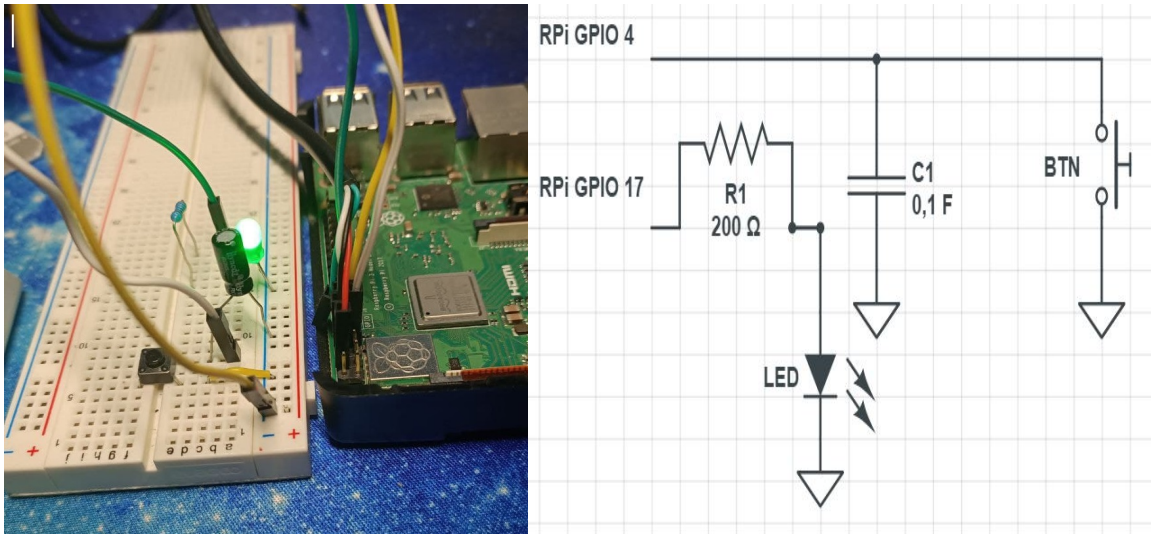
1. accensione e spegnimento di un led ad intervalli di 1 secondo.
2. alla pressione di un bottone si ha l'accensione o spegnimento di un led.
3. alla pressione di un bottone, viene avviato un task.

Il **primo programma** sui GPIO, il file init.c ha solo il task Init, dove è presente un ciclo infinito (while(1)) per il cambio di stato del LED e le API di RTEMS per la gestione del GPIO :

- rtems_gpio_initialize() : inizializza le API dei GPIO
- rtems_status_code rtems_gpio_bsp_select_input(uint32_t bank,uint32_t pin, void *bsp_specific) : imposta il pin come input, discorso analogo l'output.
- uint32_t rtems_gpio_bsp_get_value(uint32_t bank, uint32_t pin) : recupera il valore del pin
- rtems_status_code rtems_gpio_bsp_clear(uint32_t bank, uint32_t pin) : imposta il valore del pin a 0
- rtems_status_code rtems_gpio_bsp_set(uint32_t bank, uint32_t pin) : imposta il valore del pin a 1

```
...
rtems_interval  seconds = 1 * rtems_clock_get_ticks_per_second
  ();
rtems_gpio_initialize();
rtems_gpio_bsp_select_input(0, pin, &bsp_spec);
rtems_gpio_bsp_select_output(0, pin, &bsp_spec);
while(1){
  if(rtems_gpio_bsp_get_value(0, pin)){
    rtems_gpio_bsp_clear(0,pin);
  }else{
    rtems_gpio_bsp_set(0,pin);
  }
  rtems_task_wake_after(seconds);
}
...
```

Il **secondo programma** sui GPIO ha come scopo validare anche la gestione degli interrupts. Ho dovuto creare un circuito elettrico per generare un interrupt tramite pulsante e vedere la gestione dallo stato di un LED.



(a) Circuito con Raspberry Pi (b) Schema elettrico del circuito

In questo programma si hanno due task, Init e ld_switch. Il task Init si occupa della configurazione dei GPIO e interrupts, creazione del task ld_switch, creazione del semaforo e avvio del ld_switch. Il task ld_switch è quello che esegue il cambio di stato del LED che dipende dallo stato del pulsante gestito da interrupt. Per la configurazione degli interrupt ho utilizzato le API di RTEMS generiche :

- rtems_status_code rtems_gpio_resistor_mode(uint32_t pin_number, rtems_gpio_pull_mode mode) : imposta in pull-up o pull-down la resistenza del pin
- rtems_status_code rtems_gpio_enable_interrupt(uint32_t pin_number, rtems_gpio_interrupt interrupt, rtems_gpio_handler_flag flag, bool threaded_handling, rtems_gpio_irq_state (*handler) (void *arg), void *arg) : abilita l'interrupt sul pin, e assegna l'handler già definito.
- rtems_status_code rtems_gpio_debounce_switch(uint32_t pin_number, int ticks) : imposta il tempo del debounce del pin.

Tra l'handler dell'interrupt e il task ld_switch si verifica il problema della gestione della **sezione critica** sulla variabile dello stato del pulsante, ciò è stata risolta utilizzando gli **eventi** e un **semaforo binario**.
Per utilizzare i semafori è necessario aggiungere una direttiva in più "CONFIGURE_MAXIMUM_SEMAPHORES " impostando il numero massimo di semafori consentiti, nel nostro caso 1.

Di seguito il codice in cui viene gestita la sezione critica

```
...
rtems_gpio_irq_state btn_handler(void *arg) {
  sem_status = rtems_semaphore_obtain(sem_id,RTEMS_WAIT,
   RTEMS_NO_TIMEOUT);
  if(sem_status != RTEMS_SUCCESSFUL){
    print_to_console_error("rtems_sem_obt failed ", sem_status);
    exit(sem_status);
  }
  //send event to ld_switch task
  rtems_event_send(tid_1 , RTEMS_EVENT_0);
  if(get_ld_status()){
    set_ld_status(false);
  }else{
    set_ld_status(true);
  }
  rtems_semaphore_release(sem_id);
  return IRQ_HANDLED;
}
rtems_task ld_switch(rtems_task_argument sec){
  print_to_console("initialized btn_polling_task \n");
  rtems_event_set event_out;
  while(1){
    //wait until btn_interrupt send event
    rtems_event_receive(RTEMS_EVENT_0, RTEMS_WAIT,
   RTEMS_NO_TIMEOUT, &event_out);
    //wait until btn_interrupt release semamphore
    sem_status = rtems_semaphore_obtain(sem_id, RTEMS_WAIT,
   RTEMS_NO_TIMEOUT);
    if(sem_status != RTEMS_SUCCESSFUL){
      print_to_console_error("rtems_sem_obt failed ", sem_status
   );
      exit(sem_status);
    }
    if(get_ld_status()){
      rtems_gpio_bsp_set(0, ld_pin);
      print_to_console("1");
    }else{
      rtems_gpio_bsp_clear(0, ld_pin);
      print_to_console("0");
    }
    rtems_semaphore_release(sem_id);
  }
}
```

Il **terzo programma** è composto da due task oltre il task Init:

- **btn_polling** : controlla la variabile di stato del pulsante, se risulta true (quindi è stato premuto), allora viene avviato il task led_blink
- **led_blink** : accende o spegne 3 volte il LED con un intervallo di tempo di 1 secondo.

Il circuito elettrico è lo stesso del programma precedente. Anche in questo programma è presente la sezione critica per la variabile di stato del pulsante, e viene gestita nello stesso modo. In questo caso bisogna modificare lo stato dei task in modo da non farli interferire tra loro.

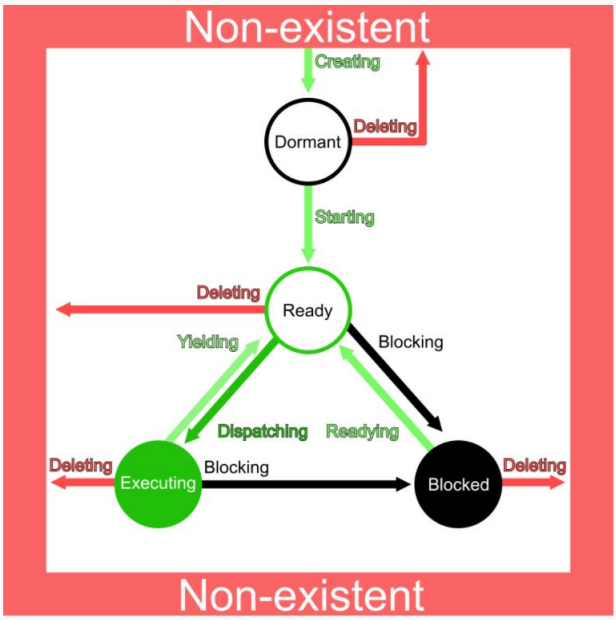


Figura 4.10: Stato dei task

Il task Init dopo aver configurato l'applicativo e creato i task, avvia il task btn_polling che passa dallo stato 'Ready' allo stato 'Executing'. Il task btn_polling quando riceve riceve l'evento dall'handler interrupt avvia il task ld_blink cambiandogli stato da 'Ready' a 'Executing' con l' API rtems_task_restart(..) oppure rtems_task_start(..), ciò dipende se il task era in stato 'Blocked'. Inoltre viene cambiato lo stato del task btn_polling in 'Blocked' con l'API rtems_task_suspend(..).
In questo modo rimane in esecuzione solo il task ld_blink, che come ultime operazioni ha il ripristino dell'esecuzione del task btn_polling con l'API rtems_task_resume e quindi viene cambiato lo stato in 'Executing' e viene sospeso il task ld_blink.
Seguendo questi passaggi rimane in esecuzione solo un task alla volta.

```
...
rtems_task btn_task(rtems_task_argument sec){
  print_to_console("initialized btn_polling_task \n");
  rtems_event_set event_out;
  while(1){
    rtems_event_receive(RTEMS_EVENT_0, RTEMS_WAIT,
   RTEMS_NO_TIMEOUT, &event_out);
    sem_status = rtems_semaphore_obtain(sem_id, RTEMS_WAIT,
   RTEMS_NO_TIMEOUT);
      if(btn_status == true){
        status_1 = rtems_task_restart( tid_2, 0);
        if ( status_1 == RTEMS_INCORRECT_STATE ) {
          rtems_task_start(tid_2,ld_blink,0);
         }else if (status_1 != RTEMS_SUCCESSFUL){
           print_to_console_error("ld_blink_task start failed ",
    status_1);
         }
        rtems_task_suspend(tid_1);
        btn_status = false;
      }
    rtems_semaphore_release(sem_id);
  }
}
rtems_task ld_blink(rtems_task_argument null){
  rtems_interval two_sec = 2 * rtems_clock_get_ticks_per_second
   ();
  uint32_t count = 0;
  print_to_console("initialized ld_blink_task \n");
  while(count<4){
    if(rtems_gpio_bsp_get_value(0, ld_pin)){
      rtems_gpio_bsp_clear(0, ld_pin);
    }else{
      rtems_gpio_bsp_set(0, ld_pin);
      count++;
    }
    rtems_task_wake_after(two_sec);
  }
  btn_status=false;
  rtems_task_resume(tid_1);
  rtems_task_suspend(tid_2);
}
```

Per l'interfaccia **I2C** è stato creato un programma che legge l'output del componente ADC che ,manipolandolo come indicato dal datasheet, da come risultato la tensione in entrata al ADC. Poiché bisogna utilizzare un componente aggiuntivo, ho dovuto creare il driver associato.
Il driver è composto da due file sorgenti:

- **mcp3425.h**: in questo file viene dichiarata la funzione **int i2c_dev_register_mcp3425(const char *bus_path, const char *dev_path, uint16_t address);** che serve per configurare il bus I2C ed associare il file in dev_path all'indirizzo di periferica address. Questo file header deve essere inserito nella cartella libchip, che contiene tutti i file header dei driver .

- **mcp3425.c**: in questo file viene definita la funzione dichiarata in "mcp3425.h", e i comandi ioctl che verranno utilizzato nel programma.

La funzione **i2c_dev_register_mcp3425** è così definita:

```
int i2c_dev_register_mcp3425(const char *bus_path, const char *
    dev_path, uint16_t address) {
  i2c_dev *dev;
  dev = i2c_dev_alloc_and_init(sizeof(*dev), bus_path, address);
  if (dev == NULL) {
    printf("error code %u from i2c_dev_alloc_and_init(..) \n",
    errno);
    fflush(stdout);
    return -1;
  }
  dev->ioctl = i2c_mcp3425_linux_ioctl;
  return i2c_dev_register(dev, dev_path);

}
```

L' API di RTEMS **i2c_dev_alloc_and_init** serve ad inizializzare e associare il 'device control' al indirizzo di periferica passato come parametro in input. Successivamente viene assegnato al campo **ioctl** la funzione **i2c_mcp3425_linux_ioctl** che contiene i comandi per la gestione del componente.

I comandi ioctl vengono così definiti:

```
static int i2c_mcp3425_linux_ioctl(i2c_dev *dev, ioctl_command_t
    command, void *arg) {
  uint8_t data [2] = {0};
  int rv = 0;
  switch (command) {
    case MCP3425_CONFIG:
      rv = i2c_mcp3425_write(dev, MCP3425_REG_CONF, 0x10);
      break;
    case MCP3425_READ:
      rv = i2c_mcp3425_read(dev, MCP3425_REG_IODIR, data);
      if(rv < 0){
        printf("read failed %d \n", rv);
        fflush(stdout);
        return rv;
      }
      uint16_t raw_adc = ((data[0] & 0x0F) * 256 + data[1]);
          if(raw_adc > 2047)
          {
            raw_adc -= 4095;
          }
      printf("value = %d\n", raw_adc);
      return rv;
      break;
    default:
      rv = -1;
  }
  return rv;
}
```

La funzione **i2c_mcp3425_linux_ioctl**gestisce due comandi che possono essere passati come parametro in input e sono "**MCP3425_CONFIG**", per cambiare la configurazione, e "**MCP3425_READ**" per eseguire una lettura.

Il comando **"MCP3425_READ"** utilizza la funzione **i2c_mcp3425_read** che valorizza l'array data[2] con i due byte ricevuti eseguiti in lettura ed è così definita:

```
static int i2c_mcp3425_read(i2c_dev *dev, uint8_t reg, uint8_t
    * reg_content) {
  i2c_msg msg [2] = {
    {
    .addr = dev->address,
    .flags = 0,
    .len = 1,
    .buf = &reg
    },
    {
    .addr = dev->address,
    .flags = I2C_M_RD,
    .len = 2,
    .buf = reg_content
    }
  };
  return i2c_bus_transfer(dev->bus, msg, 2);
}
```

In questa funzione creiamo due **i2c_msg**, una struttura che definisce i messaggi i2c da trasmettere, che verrano passati come parametro al'API **i2c_bus_transfer(...)**.
Al termine della lettura dei due byte, vengono eseguite delle operazioni sui bit dell'array data[2] in modo da escludere i 4 most significant bit ed avere il risultato in 16 bit.
Il comando ioctl "MCP3425_CONFIG" non viene utilizzato, ma viene inserito solo per rendere completo il driver, e serve a cambiare la configurazione del componente.

Per eseguire le funzioni di lettura/scritture devono essere invocate dal task Init utilizzando la system call **ioctl**(int fd, long request, ...), e serve a fare operazioni su file che rappresentano i punti di accesso alle periferiche, nel mio caso il file è ”/dev/i2c-1.mcp3425”.
La funzione open(...) recupera il file descriptor che identifica il punto di accesso, ed è il parametro in input (”fd”) nella funzione ioctl(int fd, long request), dove la request è un enum che rappresenta il comando ioctl del driver .

```
1    ...
2  int rv = i2c_dev_register_mcp3425("/dev/i2c-1", "/dev/i2c-1.
     mcp3425", MCP3425_ADDR);
3  if(rv < 0){
4    exit(errno);
5  }
6  printf("mcp34225 registration success\n");
7  fflush(stdout);
8  int fd = open("/dev/i2c-1.mcp3425", O_RDWR);
9  if(fd < 0){
10   printf("error open mcp3425 %u \n", errno);
11   fflush(0);
12   exit(errno);
13 }
14 printf("i2c.mcp3425 open success\n");
15 fflush(stdout);
16 ...
17 int count = 0;
18 while(1 && count < 3){
19   ioctl(fd, MCP3425_READ,NULL);
20   rtems_task_wake_after(1*rtems_clock_get_ticks_per_second());
21   count++;
22 }
23 rv = close(fd);
```

Per l'interfaccia **SPI** è stato creato un programma in cui si effettua una scrittura di un valore, da 0 a 4095, sul componente DAC che a sua volta restituisce come output la tensione risultante dalla conversione. Anche in questo caso si è dovuto creare il driver associato
Il driver è composto da due file sorgenti:

- **mcp4822.h**: in questo file header viene dichiarata la funzione **int spi_libi2c_register_mcp4822(unsigned spi_bus)**; che serve per configurare il bus SPI, inizializzare il driver ed associarlo al pin di selezione CE0. Questo file header deve essere inserito nella cartella libchip.
- **mcp4822.c**: in questo file viene definita la funzione dichiarata sul file header mcp4822.h, ed anche la funzione di scrittura sul componente che verrà utilizzata.

La funzione spi_libi2c_register_mcp4822 viene così definita:

```
static rtems_driver_address_table spi_mcp4822_rw_ops = {
  .read_entry = spi_mcp4822_read,
  .write_entry = spi_mcp4822_write
};

static rtems_libi2c_drv_t spi_mcp4822_rw_drv_t = {
  .ops = &spi_mcp4822_rw_ops,
  .size = sizeof(spi_mcp4822_rw_drv_t)
};

int spi_libi2c_register_mcp4822(unsigned spi_bus)
{
  return rtems_libi2c_register_drv(
            "mcp4822",
            &spi_mcp4822_rw_drv_t,
            spi_bus,
            SPI_MCP4822_ADDR
          );
}
```

L'API **rtems_libi2c_register_drv** assegna al file "mcp4822" la driver table "spi_mcp4822_rw_drv_t", il bus spi già inizializzato e passato come parametro in input, e il pin di selezione "SPI_MCP_4822_ADDR". Se il valore della macro SPI_MCP_4822_ADDR è pari a 0 allora il driver viene associato al pin CE0, analogamente se il valore della macro è pari a 1. Nella driver table vengono assegnate le operazioni tramite la 'rtems_driver_address_table' **spi_mcp4822_rw_ops** che definisce le entry point per la scrittura e lettura.
Il componente ha una comunicazione unidirezionale, per questo motivo nella funzione "spi_mcp4822_read" non viene fatta nessuna operazione.

Invece all'entry point della scrittura vien associata la funzione "**spi_mcp4822_write**" ed è così definita:

```
...
static rtems_status_code spi_mcp4822_write(
    rtems_device_major_number major, rtems_device_minor_number
    minor, void *arg) {
  rtems_status_code sc = RTEMS_SUCCESSFUL;
  rtems_libio_rw_args_t *rwargs = arg;
  int cnt = rwargs -> count;
  unsigned char *buf = (unsigned char *) rwargs -> buffer;
  /*send start*/
  sc = rtems_libi2c_send_start(minor);
  if(sc<0){
    printf("send start error \n");
    fflush(stdout);
    return sc;
  }else{
    printf("start sended \n");
    fflush(stdout);
  }
  /*set ioctl transfer mode*/
  sc = rtems_libi2c_ioctl(minor, RTEMS_LIBI2C_IOCTL_SET_TFRMODE,
    &tfr_mode);
   if ( sc != RTEMS_SUCCESSFUL ) {
    printf("ioctl error \n");
    fflush(stdout);
      return sc;
   }else{
    printf("ioctl setted \n");
    fflush(stdout);
   }
  /*send addr*/
  sc = rtems_libi2c_send_addr(minor, TRUE);
    if ( sc != RTEMS_SUCCESSFUL ) {
      printf("send address error \n");
    fflush(stdout);
      return sc;
    }else{
    printf("address sended \n");
    fflush(stdout);
    }
  /*write bytes*/
  int data_sent_cnt = rtems_libi2c_write_bytes(minor, buf, cnt);
    if (data_sent_cnt < 0) {
    printf("write bytes error \n");
    fflush(stdout);
      return RTEMS_IO_ERROR;
```

```
43      }else{
44        printf("wrote %d bytes \n", data_sent_cnt);
45        fflush(stdout);
46      }
47    /*send stop*/
48    sc = rtems_libi2c_send_stop(minor);
49      if ( sc != RTEMS_SUCCESSFUL ) {
50      printf("send stop error \n");
51      fflush(stdout);
52        return sc;
53      }else{
54      printf("stop sended \n");
55      fflush(stdout);
56      }
57    return RTEMS_SUCCESSFUL;
58  }
59  ...
```

In questa funzione viene utilizzata la libreria libi2c per la gestione del bus SPI. E' stato rilasciato anche un framework per SPI, ma dato che è stato implementato utilizzando la libreria libi2c si è pensato di usare quest'ultima
Per un corretto utilizzo delle API libi2c è necessario seguire un ordine preciso con cui vengono usate:

1. rtems_libi2c_send_start: blocca l'accesso al SPI bus alle altre periferiche, viene creata una mutua esclusione. Se si vuole utilizzare il bus SPI con altre periferiche bisogna utilizzare l'api "rtems_libi2c_send_stop".

2. rtems_libi2c_ioctl: serve ad effettuare operazioni ioctl sul componente. In questo caso viene usato per impostare le specifiche di trasferimento dei dati, ad esempio il baudrate.

3. rtems_libi2c_send_addr: attiva il pin di selezione associato al componente, in questo caso il pin CE0.

4. rtems_libi2c_write_bytes: effettua la scrittura dei byte sul componente

5. rtems_libi2c_send_stop: disattiva il pin di selezione e sblocca il bus SPI, in modo da poter essere utilizzato con altre periferiche.

La funzione rtems_libi2c_ioctl ha come parametro in input il ”&tfr_mode”, ed è un struttura che definisce la modalità di trasferimento dei dati:

```
static rtems_libi2c_tfr_mode_t tfr_mode = {
  .baudrate = 20000000,
  .bits_per_char = 8,
  .lsb_first = FALSE,
  .clock_inv = TRUE,
  .clock_phs = FALSE
};
```

Questa struttura viene definita facendo riferimento alle specifiche del componente, che si possono trovare nel datasheet [8]

4.4 Risultati finali

I programmi sono stati creati uno alla volta, e prima di passare alla creazione del successivo, sono state fatte delle prove per verificare effettivamente se il risultato è quello atteso. L'ordine seguito nel creare i programmi è stato per grado di difficoltà e riutilizzo delle API.
L'approccio iniziale non è stato semplice, poiché non era totalmente chiaro la struttura dei file presenti nella BSP, e quali API utilizzlare.
Per questo motivo è stato fatto un incontro con l'ing. Lastri, che ha già lavorato su RTEMS, per spiegare in linea generale il funzionamento di RTEMS con piccoli esempi pratici, e per rispondere ai dubbi rilevati durante la creazione dei primi programmi di validazione.
Il programma di validazione più complessi sono stati quelli nella creazione del driver per il componente I2C e SPI.
Per la creazione dei driver ho preso come riferimento i programmi di test dati a disposizione dall'ing. Marques, nel suo repository [5].
Durante la creazione del programma per l'I2C, il primo problema è stato l'utilizzo di un bus path errato, ciò causava un errore in run-time durante l'inizializzazione del bus. Dopo aver individuato ed impostato il bus path corretto, l'applicativo non dava più errori in run-time ma rimaneva bloccato sull'operazione di lettura.
A questo punto è stata aperta una 'issue' sul git repository dell'ing. Marques, per poter chiedere più informazioni riguardo le API di I2C. L'ing. Marques ha fatto notare che i cavetti che collegano la Raspberry Pi al componente mcp3425 non devono esssere più lunghi di 20cm, per evitare che i dati vengano persi durante la comunicazione per via di un alta impedenza. Questo causerebbe un attesa infinita della Raspberry Pi per la ricezione dei byte. Infatti sostituendo i cavetti si è riusciti a effettuare le letture desiderate sul componente.
Il programma di validazione del SPI sembra funzionare, perchè non vengono rilevati errori sul terminale e l'esecuzione non viene mai bloccata, ma il componente mcp4822 non da il risultato atteso, come se la scrittura non venisse eseguita.
E' stato contattato l'ing. Kubler per chiarire le specifiche del componente, e i requisiti che il software deve rispettare per la scrittura del dato. Infatti un requisito non soddisfatto è la gestione del livello del 'chip select', cioè il pin di selezione, che non viene impostato ad alto alla fine della scrittura. Quindi si ipotizza che ci sia un errore nell'API di RTEMS per la gestione del bus SPI, per questo motivo è stata aperta una 'issue' nel repository del'ing. Marques per avere più chiarimenti lato software.

5 Conclusioni

Le attività del mio tirocinio si basano principalmente su RTEMS ed il suo utilizzo.
Tutta la documentazione utilizzata è in lingua inglese, anche per questo all'inizio ho avuto un po' di difficoltà a capire come funzionasse RTEMS.
Infatti è stato speso molto tempo nel porting, dato che non è immediato e non c'è una guida aggiornata italiana. Per questo motivo ho prodotto la guida in italiano per chi vuole approcciarsi per la prima volta ad RTEMS ed utilizzarlo su Raspberry Pi.
In questo modo un utente medio può creare programmi real-time multiprocessore non banali, come quelli presenti nell'ambito aerospaziale.
Il lavoro svolto fa parte dei progetti di BIS-Italia, sezione italiana della British Interplanetary Society, società storica britannica di cui sono membro, che mi ha seguito durante lo stage. BIS-Italia prevede di utilizzare RTEMS su Raspberry Pi per il progetto di una replica in scala 1:3 di ExoMars Rover che verrà utilizzato per divulgazione.
Tutto il lavoro è stato svolto con l'aiuto dei membri di BIS-Italia e la collaborazione di Microchip.

Ringraziamenti

Bibliografia

[1] RTEMS Team. *RTEMS User Manual*. URL: https://docs.rtems.org/branches/master/user/index.html.

[2] Giorgio Basile. *RTEMS v4.11 on RasperryPi*. URL: https://gist.github.com/giorgiobasile/461d8b8c15d59c6c4445066af6a3124b.

[3] Giorgio Basile. *RTEMS v5 on RasperryPi*. URL: https://gist.github.com/giorgiobasile/1c1930a8a3ff8e36061cd7f4ef83da95.

[4] RTEMS Team. *RTEMS Classic API Guide*. URL: https://docs.rtems.org/branches/master/c-user/index.html.

[5] asuol. *RTEMS rpi testing*. URL: https://github.com/asuol/RTEMS_rpi_testing.

[6] Microchip. *MCP3425 SOT23-6 Evaluation Board User's Guide*. URL: https://ww1.microchip.com/downloads/en/DeviceDoc/51787a.pdf.

[7] Microchip. *16-Bit Analog-to-Digital Converter with I2C Interface and On-Board Reference*. URL: https://ww1.microchip.com/downloads/en/DeviceDoc/22072b.pdf.

[8] Microchip. *MCP4802/4812/4822 Data Sheet*. URL: https://ww1.microchip.com/downloads/en/DeviceDoc/20002249B.pdf.

[9] Microchip. *10-Pin MSOP and 8-Pin MSOP Evaluation Board User's Guide*. URL: https://ww1.microchip.com/downloads/en/DeviceDoc/50002569A.pdf.

[10] Broadcom. *BCM2835 ARM Peripherals*. URL: https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/BCM2835-ARM-Peripherals.pdf.

Guida porting RTEMS su Raspberry Pi

Clark Ezpeleta

1 Introduzione

Guida per l'installazione di RTEMS RSB e della tool-suite per l' utilizzo del kernel di RTEMS su architetture ARM, con target i BSP di Raspberry Pi 1 e Raspberry Pi 2, quest'ultimo è compatibile con la Raspberry pi 3.

L' RSB (RTEMS Source Builder) è un tool utile per buildare i moduli di RTEMS e delle BSP da utilizzare.

La BSP (Board Support Package) è il codice di supporto per una specifica scheda, esso contiene librerie di RTEMS utili per l'utilizzo della scheda.

RTEMS è un progetto open source in perenne sviluppo, al momento stanno sviluppando la v6 ma non è ancora stabile, e al momento della redazione di questa guida si installerà la v5.1 che è release stabile più recente.

In questa guida viene usato il sistema operativo Ubuntu LTS 20.04.

2 Configurazione iniziale del sistema

Prima di iniziare l'installazione di RTEMS RSB e della tool-suite bisogna configurare l'ambiente di sviluppo.

Procediamo con l'installazione dei seguenti packages:

- build-essential :

```
$ sudo apt-get install build-essential
```

- git:

```
$ sudo apt-get install git
```

- python-dev:

```
$ sudo apt-get install python-dev

```

A questo punto definiamo la struttura delle cartelle in cui verranno installati i componenti di RTEMS.

- $HOME/rtems-dev : base directory
- $HOME/rtems-dev/src : base directory dove vengono clonati da git il codice sorgente di RTEMS e la RSB di RTEMS
- $HOME/rtems-dev/src/rsb : RTEMS source builder
- $HOME/rtems-dev/src/rtems : RTEMS tool-suite
- $HOME/rtems-dev/rtems/ : dove verrà installata la tool-suite di RTEMS
- $HOME/rtems-dev/build : dove verranno installate i BSP di Rpi1 e Rpi2

3 Installazione RTEMS

Dopo aver preparato l'ambiente di sviluppo, possiamo procedere con l'installazione di RTEMS:

- Cloniamo i repository del RTEMS RSB e del codice sorgente di RTEMS :

```
$ mkdir -p $HOME/rtems-dev/src
$ cd $HOME/rtems-dev/src
$ git clone -b 5.1 git://git.rtems.org/rtems-source-builder.git rsb
$ git clone -b 5.1 git://git.rtems.org/rtems.git
```

- Eseguiamo la build e l'installazione della tool suite utilizzando la RSB :

```
$ cd $HOME/rtems-dev/src/rsb/rtems
$ ../source-builder/sb-set-builder --prefix=$HOME/rtems-dev/rtems/5 \
  5/rtems-arm
```

- Dopo aver completato l'installazione possiamo controllare che il C cross compiler di RTEMS sia stato installato correttamente utilizzando il seguente comando:

```
$ $HOME/rtems-dev/rtems/5/bin/arm-rtems5-gcc --version
```

  e si ha come risultato:

```
clark@clark-ezpe-pc:~/Scrivania/RTEMS-Rpi3B$ $HOME/rtems-dev/rtems/5/bin/arm-rtems5-gcc --version
arm-rtems5-gcc (GCC) 7.5.0 20191114 (RTEMS 5, RSB 5 (46e9d4911f09 modified), Newlib 7947581)
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

- Inseriamo nelle variabili di ambiente i comandi della toolchain e procediamo con il bootstrap :

```
$ export PATH=$HOME/rtems-dev/rtems/5/bin:"$PATH" (3.1)
$ cd $HOME/rtems-dev/src/rtems
$ ./rtems-bootstrap
```

- Adesso possiamo configurare ed installare le BSP che ci servono:

```
$ mkdir -p $HOME/rtems-dev/build
$ cd $HOME/rtems-dev/build
$ $HOME/rtems-dev/src/rtems/configure \
        --prefix=$HOME/rtems-dev/rtems/5 \
  --target=arm-rtems5 \
  --enable-rtemsbsp="raspberrypi raspberrypi2"\
  --enable-tests=samples --enable-networking --enable-posix
$ make
$ make install
```

Svolti tutti i passaggi precedenti abbiamo come risultato RTEMS installato sul computer host.

4 Prova sample test RTEMS

Installando le BSP, RTEMS ci fornisce dei sample test .exe da cui possiamo generare i file .img e utilizzarli per testare il funzionamento della raspberrypi

I sample test sono in :

- rpi 1:

```
$ $HOME/rtems-dev/build/arm-rtems5/c/raspberrypi1/testsuites/samples
```

- rpi 2 e 3:

```
$ $HOME/rtems-dev/build/arm-rtems5/c/raspberrypi2/testsuites/samples
```

In questa guida utilizziamo il sample 'ticker.exe', ma i passaggi che verranno illustrati valgono anche per gli altri sample presenti in cartella. Per poter utilizzare il sample test dobbiamo fare 2 passaggi:

- Creazione kernel file .img:

  assicurarsi di avere come variabile di ambiente i comandi della tool-suite:

```
$ echo $PATH
```

  se non è presente '$HOME/rtems-dev/rtems/5/bin' allora bisogna inserirla (vedi comando 3.1).

  posizionarsi nella cartella dove vogliamo che venga creato il file .img:

```
$ cd $HOME/rtems-dev/rtems
```

  generare il file .img:

  Per Rpi1:

```
$ arm-rtems5-objcopy -Obinary $HOME/rtems-dev/build/arm-rtems5/c
  /raspberrypi1/testsuites/samples/ticker.exe ticker.img
```

  Per Rpi2 e Rpi3:

```
$ arm-rtems5-objcopy -Obinary $HOME/rtems-dev/build/arm-rtems5/c
  /raspberrypi2/testsuites/samples/ticker.exe ticker.img
```

- Configurare la SD card: copiamo nella scheda sd il firmware di Rpi (v4.19.11.3) compatibile con RTEMS. Il firmware puo' essere scaricato da questo link: https://github.com/raspberrypi/firmware/tree/5574077183389cd4c65077ba18b59144ed6ccd6d/boot Eliminiamo tutti i file kernel*.img, questi verranno sostituiti dal file kernel che abbiamo generato precedentemente. Copiamo nella sd il file ticker.img creato precendetemente. Creiamo nella sd il file config.txt che contiene il seguente testo:

```
    enable_uart=1
    kernel_address=0x200000
    kernel=ticker.img
```

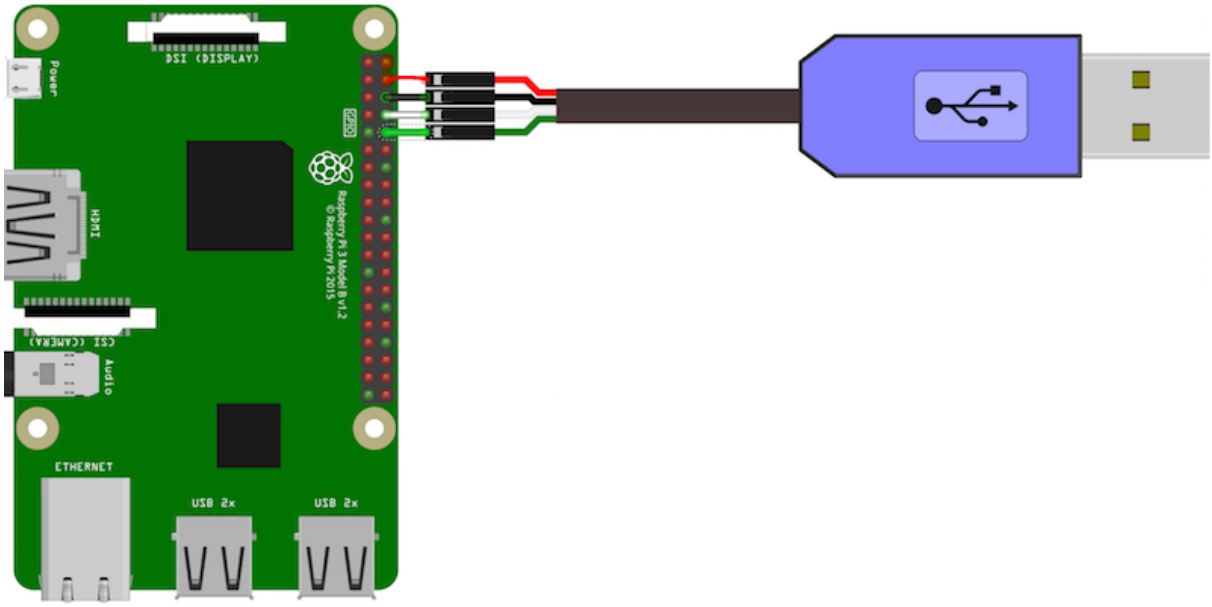
  Nel campo kernel mettiamo il nome del kernel file che vogliamo che rpi esegua.

A questo punto siamo pronti con il test.

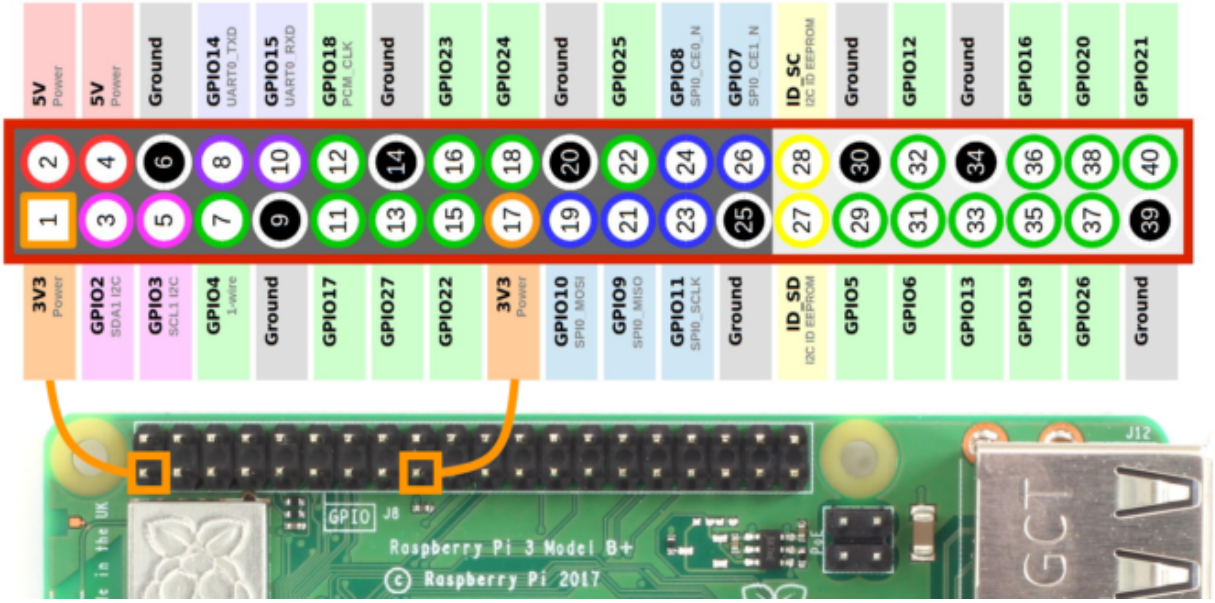
Per poter vedere la log della UART dobbiamo installare minicom:

```
$ sudo apt-get install minicom
```

Colleghiamo il convertitore UART alla Rpi :



dove i GPIO pin sono :



Colleghiamo la chiaveta USB al computer ed eseguiamo il comando per vedere la log della UART :

```
$ sudo minicom -b 115200 -D /dev/serial/
  by-id/<indirizzo periferica utilizzata per UART di solito un USB TTL>
```

il risultato dovrebbe essere simile a questo:

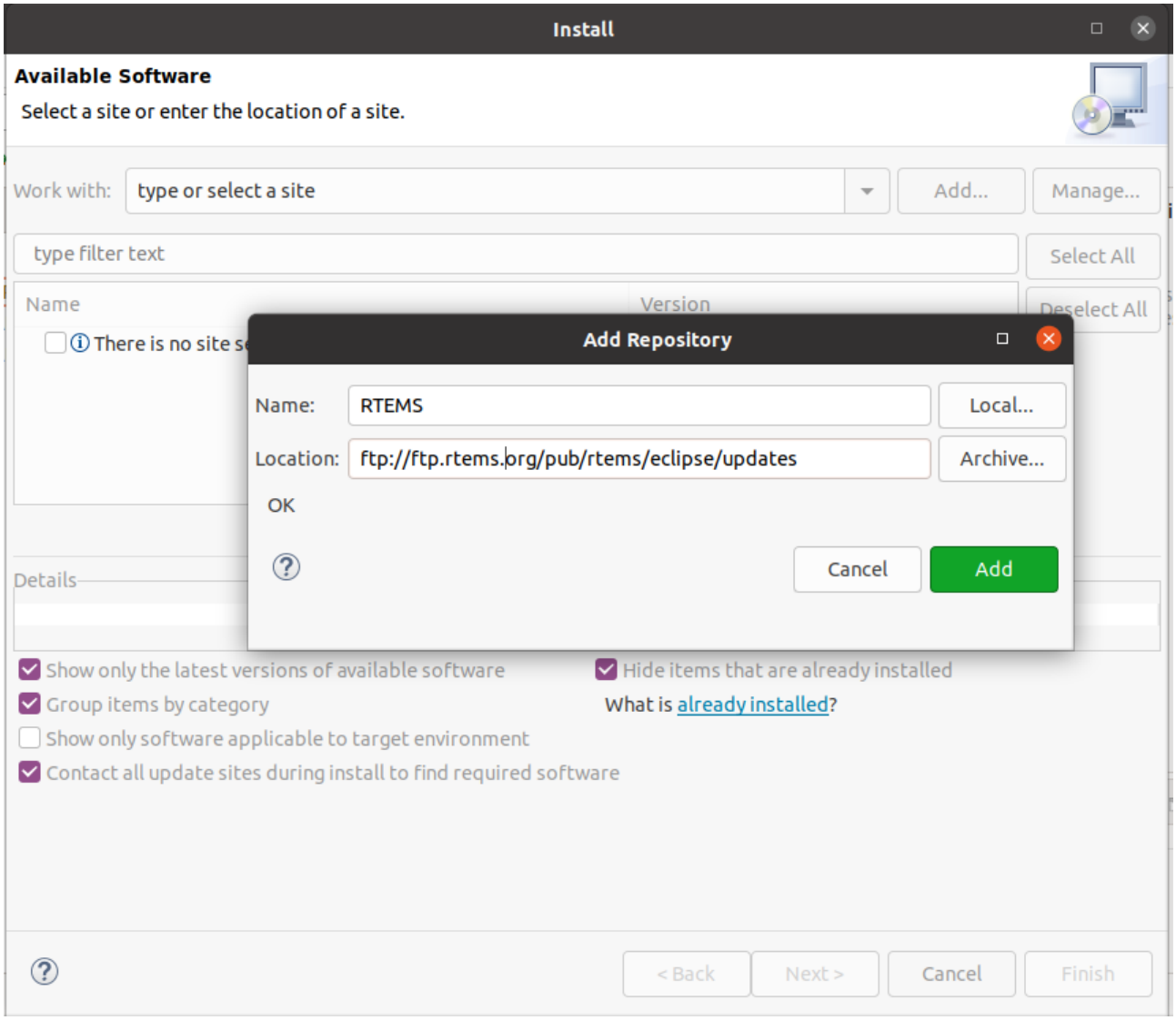
```
RTEMS RPi 3B+ 1.3 (1GB) [00a020d3]


*** BEGIN OF TEST CLOCK TICK ***
*** TEST VERSION: 5.0.0.61ccb9c05dcd695114541960aa6bfc1315f30514
*** TEST STATE: EXPECTED_PASS
*** TEST BUILD: RTEMS_NETWORKING RTEMS_POSIX_API
*** TEST TOOLS: 7.5.0 20191114 (RTEMS 5, RSB 5 (46e9d4911f09 modified), Newlib 7947581)
TA1  - rtems_clock_get_tod - 09:00:00   12/31/1988
TA2  - rtems_clock_get_tod - 09:00:00   12/31/1988
TA3  - rtems_clock_get_tod - 09:00:00   12/31/1988
TA1  - rtems_clock_get_tod - 09:00:04   12/31/1988
TA1  - rtems_clock_get_tod - 09:00:09   12/31/1988
TA2  - rtems_clock_get_tod - 09:00:10   12/31/1988
TA1  - rtems_clock_get_tod - 09:00:14   12/31/1988
TA3  - rtems_clock_get_tod - 09:00:15   12/31/1988
TA1  - rtems_clock_get_tod - 09:00:19   12/31/1988
TA2  - rtems_clock_get_tod - 09:00:20   12/31/1988
TA1  - rtems_clock_get_tod - 09:00:24   12/31/1988
TA1  - rtems_clock_get_tod - 09:00:29   12/31/1988
TA3  - rtems_clock_get_tod - 09:00:30   12/31/1988
TA2  - rtems_clock_get_tod - 09:00:30   12/31/1988
TA1  - rtems_clock_get_tod - 09:00:34   12/31/1988

*** END OF TEST CLOCK TICK ***


*** FATAL ***
fatal source: 5 (RTEMS_FATAL_SOURCE_EXIT)
fatal code: 0 (0x00000000)
RTEMS version: 5.0.0.61ccb9c05dcd695114541960aa6bfc1315f30514
RTEMS tools: 7.5.0 20191114 (RTEMS 5, RSB 5 (46e9d4911f09 modified), Newlib 7947581)
executing thread ID: 0x08a010002
executing thread nam
RTEMS RPi 3B+ 1.3 (1GB) [00a020d3]
```

5 Configurazione Eclipse C/C++

Per creare i file sorgenti per programmi di RTEMS possiamo utilizzare l'IDE Eclipse C/C++ scaricabile da questo link : https://www.eclipse.org/downloads/packages/

Una volta installato Eclipse C/C++ bisogna installare il plugin di RTEMS:

- Andiamo su Help¿ Install New Software
- Aggiungiamo come Software site quello di RTEMS, quindi clicchiamo add e inseriamo il seguente url ftp://ftp.rtems.org/pub/rtems/eclipse/updates

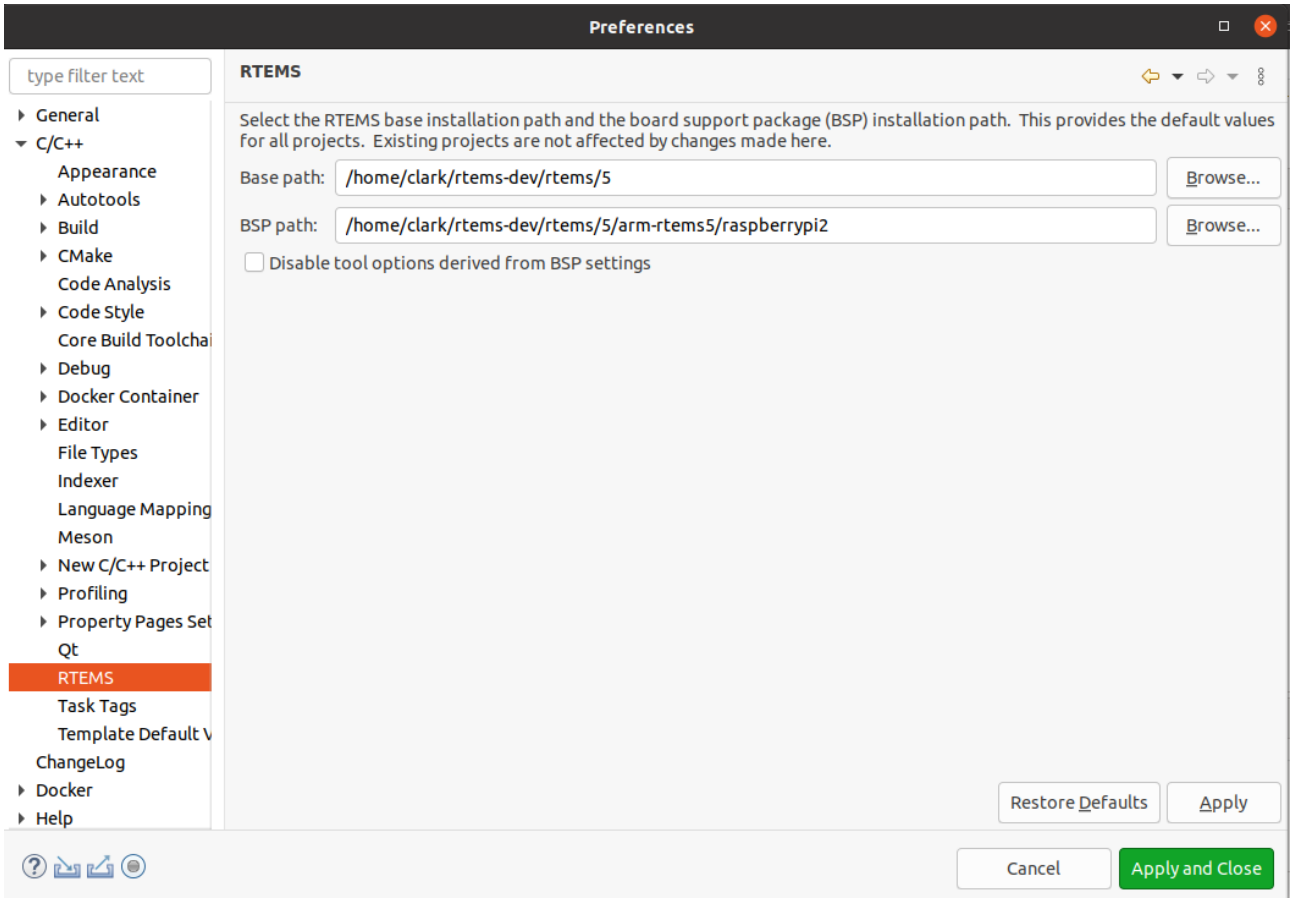


- Selezioniamo il plugin RTEMS CDT Support e installiamo

A questo punto abbiamo installato il plugin di RTEMS ed è pronto ad essere utilizzato.

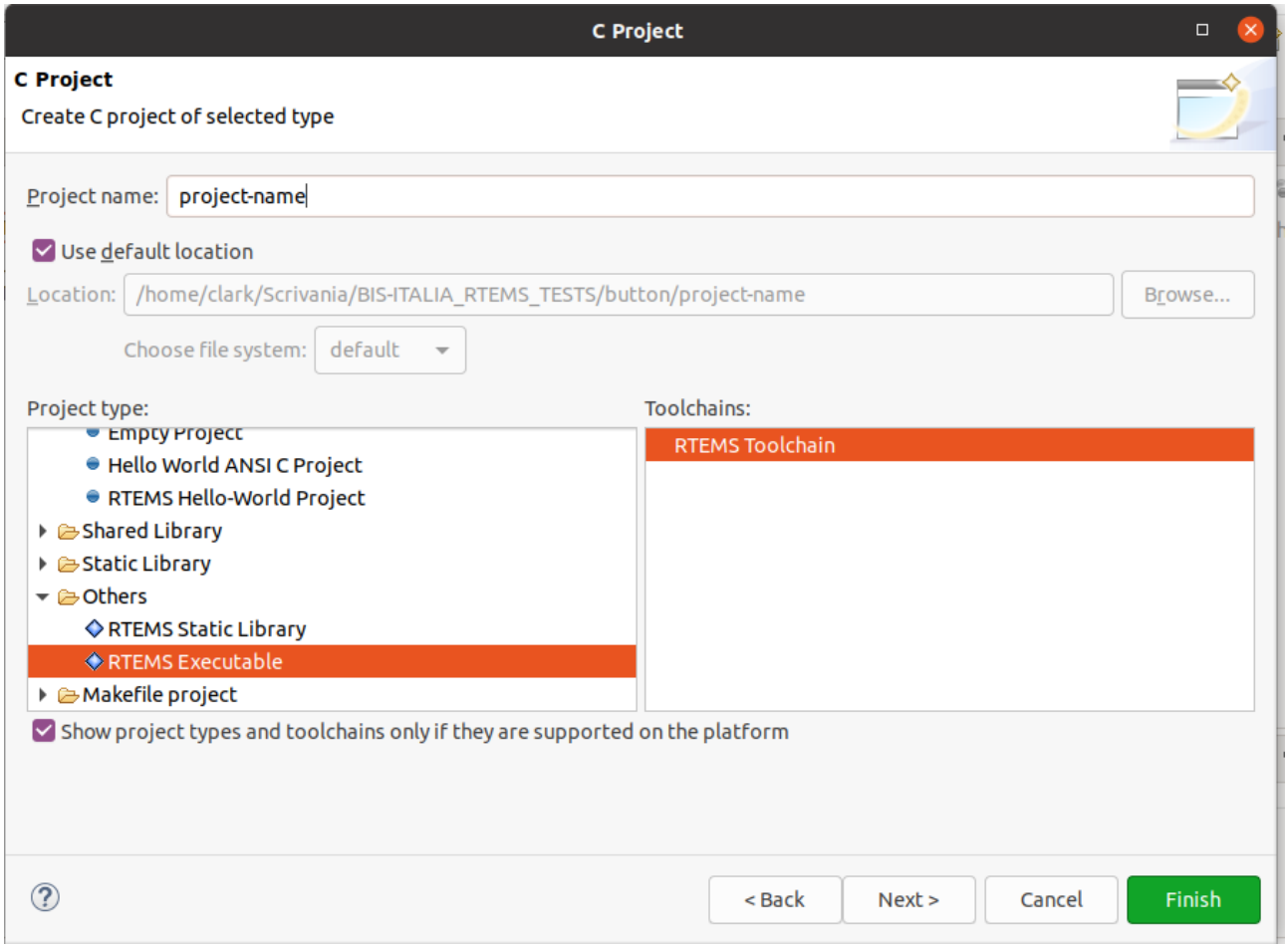
Quando creiamo un progetto RTEMS bisogna configurare la base e il BSP path di RTEMS in modo che si riesca ad utilizzare le librerie che ci servono:

- Window → Preferences → C/C++ → RTEMS
- Base path : base path dell'installazione , nel nostro caso home/¡username¿/rtems-dev/rtems/5
- BSP path : path dell'installazione della BSP, nel nostro caso home/¡username¿/rtems-dev/rtems/5/arm-rtems5/raspberrypi2 (o 1 se si utilizza Rpi1)



Abbiamo completato la configurazione del plugin di RTEMS su Eclipse C, adesso possiamo procedere alla creazione di un progetto RTEMS:

- New project→ C Project
- Selezionare il project type corretto: Others→ RTEMS Executable
- Selezionare la Toochain corretta : RTEMS Toolchain



Possiamo controllare che il progetto è stato creato correttamente controllando le sue properties → C/C++ Build → RTEMS i path siano quelli corretti.

6 Prova con Eclipse C/C++

Possiamo adesso fare un semplice programma di prova ”hello world”, creiamo 2 file sorgenti: - init.c : conterrà il task principale (Init) che è il task di ”ingresso”. - system.c : conterrà il codice di configurazione di RTEMS.

in init.c inseriamo il seguente codice:

```
/* Hello world example */
#include <rtems.h>
#include <stdlib.h>
#include <stdio.h>
#include <bsp/mmu.h>

rtems_task Init(rtems_task_argument ignored) {
  printf("\n\n*** HELLO WORLD TEST ***\n");
  printf("Hello World\n");
  printf("*** END OF HELLO WORLD TEST ***\n");
  exit(0);
}
```

in system.c inseriamo il seguente codice :

```

/* Simple RTEMS configuration */

#define CONFIGURE_APPLICATION_DOES_NOT_NEED_CLOCK_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_SIMPLE_CONSOLE_DRIVER

#define CONFIGURE_UNLIMITED_OBJECTS
#define CONFIGURE_UNIFIED_WORK_AREAS

#define CONFIGURE_RTEMS_INIT_TASKS_TABLE

#define CONFIGURE_INIT

#include <rtems/confdefs.h>
```

per una migliore comprensione del codice si rimanda ai seguenti link:

- per le api:
  https://devel.rtems.org/browser/rtems/bsps/arm/raspberrypi
- per la configurazione di rtems su codice (system.c):
  https://ftp.rtems.org/pub/rtems/people/joel/docs-eng/c-user/configuring_a_system.html

A questo punto possiamo buildare il programma, e nella cartella del progetto troveremo la cartella ’RTEMS Executable Configuration’ che contiene il file .exe che utilizzeremo per creare il file .img da mettere nella scheda SD. I passaggi per la creazione del file .img sono sopracitati.

7 Creazione file kernel senza IDE

Se si vuole creare il file .img partendo dai file sorgente allora dobbiamo usare i seguenti comandi nel terminale:

- ci posizioniamo nella cartella dove ci sono i file sorgenti.
- eseguiamo i comandi del cross compiler di RTEMS su tutti i file .c:

```
$ $HOME/rtems-dev/rtems/5/bin/arm-rtems5-gcc -B \
  $HOME/rtems-dev/rtems/5/arm-rtems5/raspberrypi2/lib/ \
  -specs bsp_specs -qrtems -ffunction-sections -fdata-sections \
  -march=armv7-a -mthumb -mfpu=neon -mfloat-abi=hard \
  -mtune=cortex-a7 -Os -g -Wall -c -fmessage-length=0 -pipe -MMD \
  -MP -MF"init.d" -MT"init.o" -o "init.o" "../init.c"
```

- utilizziamo il linker per la creazione del file .exe :

```
$ $HOME/rtems-dev/rtems/5/bin/arm-rtems5-gcc -B \
  $HOME/rtems-dev/rtems/5/arm-rtems5/raspberrypi2/lib/ \
  -specs bsp_specs -qrtems -ffunction-sections -fdata-sections \
  -march=armv7-a -mthumb -mfpu=neon -mfloat-abi=hard \
  -mtune=cortex-a7 -Wl,--gc-sections \
  -o "hello-rpi.exe"  ./init.o ./system.o
```

a questo punto possiamo creare il file .img e copiarlo sulla scheda sd e fare i passaggi già esposti precedentemente.

8 Sitografia

- Documentazione RTMES :
  https://docs.rtems.org/branches/master/user/index.html
- Guida RTEMS di Giorgio :
  https://gist.github.com/giorgiobasile/1c1930a8a3ff8e36061cd7f4ef83da95
- Sito spiegazione porting RTEMS (Alan Tech):
  http://alanstechnotes.blogspot.com/2013/03/rtems-on-raspberry-pi.html
- Documentazione API RTEMS:
  https://docs.rtems.org/branches/master/c-user/index.html