Alma Mater Studiorum
Università degli Studi di Bologna

SCUOLA DI INGEGNERIA E ARCHITETTURA

dipartimento di ingegneria dell'energia elettrica e dell'informazione "Guglielmo Marconi" - DEI

corso di laurea in Ingegneria Elettronica e Telecomunicazioni

tesi di laurea in Laboratorio di Architetture e Programmazione dei Sistemi Elettronici Industriali

SVILUPPO E TEST DI UNA RETE DI SENSORI INDOSSABILI PER IL MONITORAGGIO E LA RIABILITAZIONE DI PAZIENTI AFFETTI DA MORBO DI PARKINSON

Candidato:
Leonardo Cecconi

Relatore:
Chiar.mo Prof. Luca Benini

Correlatori:
Dott. Filippo Casamassima

Anno Accademico 2014/2015 - Sessione II

A mia nonna
che sarebbe felice di vedermi qui oggi

Introduzione

Il recente sviluppo commerciale di smartphone, tablet e simili dispositivi, ha portato alla ricerca di soluzioni hardware e software dotate di un alto livello di integrazione, in grado di supportare una potenza di calcolo e una versatilità di utilizzo sempre più crescenti, pur mantenendo bassi i consumi e le dimensioni dei dispositivi. Questo sviluppo ha consentito parallelamente a simili tecnologie di trovare applicazione in tanti altri settori, tra i quali quello biomedicale. In particolare l'emergere di sensoristica sempre più compatta ed efficiente, affiancata da unità a microprocessore e moduli di comunicazione wireless a basso consumo, ha portato allo sviluppo di applicazioni che si inseriscono nella categoria delle Wireless Sensor Network (WSN) o, più genericamente, delle Wireless Body Area Network (WBAN). Questo tipo di architettura consiste in genere di diversi nodi hardware indossabili da un soggetto biologico (Wearable Devices), in comunicazione wireless tra loro, con l'obbiettivo di acquisire dati dall'ambiente circostante tramite appositi sensori, elaborarli e inviarli o restituire feedback di diverso tipo a seconda dell'applicazione. E' facile intuire come le WBAN abbiano presto interessato il settore biomedicale con lo scopo: da un lato di fornire nuovi strumenti di diagnostica al personale sanitario e dall'altro di consentire terapie riabilitative "smart" che consentano al paziente una riabilitazione più personalizzata e all'operatore sanitario un monitoraggio da remoto più efficiente e costante. Questo tipo di applicazioni inoltre sono più vantaggiose dal punto di vista economico rispetto alle opzioni convenzionali e risultano meno invasive per il paziente stesso durante le sue attività quotidiane.

Obbiettivi

Il lavoro esposto in questa tesi si inserisce nel contesto appena descritto e, in particolare, consiste nello sviluppo di un sistema WBAN ideato per garantire maggiore flessibilità, controllo e personalizzazione nella terapia riabilitativa dei pazienti affetti da Morbo di Parkinson (MP), una malattia neurodegenerativa che coinvolge determinate aree del cervello che sono connesse al controllo dei movimenti. In questo campo è stata dimostrata l'efficacia, in termini di miglioramento delle condizioni

di vita dell'individuo, dell'esercizio fisico e in particolare di una serie di fisioterapie riabilitative specifiche.
Tuttavia manca ancora uno strumento in grado di garantire più indipendenza, continuità e controllo,per le persone affette da MP, durante l'esecuzione di questi esercizi; senza che sia strettamente necessario l'intervento di personale specializzato per ogni seduta fisioterapeutica. Inoltre manca un sistema che possa essere comodamente trasportato dal paziente nelle attività di tutti i giorni e che consenta di registrare e trasmettere eventi particolari legati alla patologia, come blocchi motori e cadute accidentali.
L'obbiettivo di questo progetto è quindi quello di fornire tale strumento, in modo da poter porre le basi per una terapia riabilitativa mirata che si possa svolgere per il paziente in un contesto casalingo e che garantisca al contempo anche un sistema di monitoraggio sia per il personale medico che segue il paziente, sia per il paziente stesso. Questo per rendere parzialmente indipendete il paziente nello svolgimento dell'attività riabilitativa e nel controllo degli episodi critici della patologia, al fine ultimo di migliorarne le condizioni di vita.

Progetto

Il presente lavoro di tesi, in particolare, tratta della realizzazione di un Firmware per la gestione di un Nodo Centrale che funge da master in una rete WBAN a tre nodi. L'obbiettivo per il seguente testo è quello di integrare in tale firmware le funzioni di acquisizione dati dai sensori on-board, comunicazione tra i nodi della rete e gestione delle periferiche hardware secondarie; il tutto utilizzando per lo sviluppo un Sistema Operativo Real-Time (RTOS), al fine di valutarne i vantaggi per questo tipo di applicazione. Nello specifico tutti i nodi, compreso il Nodo Centrale, sono costituiti da schede hardware "ad hoc" già dotate di tutti i sistemi di comunicazione, acquisizione e feedback necessari per le finalità del progetto.

Sintesi dei contenuti

Essenzialmente il progetto software è costituito da due parti interconnesse: la raccolta ed elaborazione dati e la comunicazione nella WBAN tra due nodi sensori e il Nodo Centrale ricevente che funge da master. Per quanto riguarda la raccolta dati dai sensori, dopo l'inizializzazione delle periferiche e l'avvio del sistema operativo, vengono chiamati diversi task, con struttura simile, a intervalli temporali prefissati a seconda dalla frequenza di campionamento voluta. Il compito di questi task è la lettura, via bus I2C, dei parametri fisici campionati dai sensori. Ogni task di lettura si conclude con il passaggio dei dati ad un task ricevente, che li filtra e li salva

in memoria. La comunicazione nella WBAN invece può essere divisa in due parti: l'inizializzazione della connessione tra i nodi e la gestione della linea, comprensiva dell'invio e ricezione dei pacchetti dati. La prima parte consiste in una procedura di inizializzazione ad alta priorità che viene avviata prima di tutte le altre e che, a partire dai Device Address, stabilisce dal Nodo Centrale (master della WBAN) una connessione Multipoint bidirezionale con i due nodi sensori. E' stata inoltre scritta una seconda procedura di inizializzazione per stabilire connessioni singole punto-punto per facilitare le operazioni di debug e manutenzione del sistema.
La seconda parte di gestione e comunicazione è costituita da un insieme più ampio di task che vengono inseriti nell'esecuzione ciclica dello scheduler di RTOS e mantenuti in attesa fino al sopraggiungere di eventi particolari. Nello specifico: un task è stato dedicato alla sola trasmissione Bluetooth, ed è "triggerato" dall'arrivo di stringhe da inviare da altri task di controllo. Ad un secondo task è invece affidato la sola ricezione dai nodi sensori. Questo è attivato da interrupt di ricezione sulla interfaccia USART (collegata al modulo) e collabora con una interrupt service routine per gestire i pacchetti in arrivo. Il task si conclude inviando i dati ricevuti al task per il salvataggio e l'elaborazione finale. Infine un terzo e quarto task sono invece triggerabili da utente e interagiscono con il task di trasmissione, inviando a questo i comandi da trasmettere ai nodi sensori per modificarne il comportamento ed iniziare o terminare le attività di sensoristica. L'insieme di tutti questi processi, come già detto, è regolato dallo scheduler del RTOS, seguendo particolari temporizzazioni e livelli di priorità. Per una corretta gestione dei processi tramite RTOS sono stati anche implementate le misure necessarie per il controllo di esecuzione dei vari task sviluppati. In particolare si è fatto ricorso agli strumenti software forniti dal RTOS come i sistemi Mutex, Mailbox, Event e Virtual Timer, che verranno approfonditi nel testo.

Sintesi dei risultati

In conclusione dall'esito del lavoro svolto si è potuto osservare che:
L'utilizzo di un RTOS è risultato vantaggioso sotto molti aspetti in applicazioni complesse, come è questo il caso, e fornisce artifici software immediati e di grande utilità. L'esecuzione parallela dei task richiede tuttavia grande attenzione verso problemi non presenti in uno stile di programmazione sequenziale "Super-Loop"; in particolare verso la gestione delle risorse hardware, l'interrompibilità delle routine software e l'interferenza tra processi paralleli.
Per quanto riguarda la connessione Bluetooth, la modalità Multipoint è affidabile e la potenza di trasmissione è più che sufficiente per una WBAN, tuttavia velocità di trasmissione troppo elevate (benchè spesso non necessarie) vanno a destabilizzare

in parte la sincronizzazione dello scheduler del RTOS. Si sono resi necessari quindi controlli più complessi sulla gestione dei task nell'ambito della prioritizzazione e della gestione degli interrupt; in particolare è stato necessario ridurre al minimo il carico computazionale sugli handler degli interrupt.
Nel complesso si è giunti alla realizzazione di un firmware funzionante rispetto agli obbiettivi prefissati. In particolare l'utilizzo di un RTOS non ha avuto impatti apprezzabili in termini di perdita di prestazioni del sistema e si è dimostrato una valida scelta per il caso di studio. Nel complesso il carico computazionale sviluppato dai processi nel firmware è ridotto e lascia ampio spazio alle implementazioni di eventuali moduli software successivi. In particolare è stato calcolato che il software sviluppato richiede un'occupazione di memoria RAM stimata attorno ai 17 KByte e uno spazio in memoria (flash) di circa 64 KByte. Questi valori sono supportati con largo margine dalla MCU scelta la quale offre fino a 192 KByte di RAM e fino ad 1 MByte di memoria flash.
Sono stati anche analizzati alcuni aspetti dei consumi, stimati attorno a 300 mW con picchi di assorbimento in corrente fino a 73 mA. Tuttavia, non essendo obbiettivo di questo testo, non sono state implementate misure per l'attuazione di modalità low power (per l'unità microcontrollore o il modulo Bluetooth) che restano parte degli sviluppi futuri del progetto.
Quello che si è riusciti ad ottenere è quindi la dimostrazione della funzionalità dell'utilizzo di un sistema operativo real-time per questa applicazione e la prova della potenzialità del sistema hardware sviluppato, e dell'utilizzo di una connessione Bluetooth Multipoint per lo scambio dati nella WBAN.

Viene ora esposto il testo dettagliato del lavoro. In particolare in una prima parte sono elencati gli obbiettivi tecnici ed è illustrata una panoramica sul supporto hardware e software a disposizione, in termini di componenti usati e loro caratteristiche significative; successivamente vengono analizzate in ordine, secondo indice, le strutture dei vari task e la gestione di questi nell'ottica di un RTOS.
Infine vengono esposte alcune considerazioni sul lavoro svolto, sui risultati ottenuti e sulle possibilità di ampliamento del progetto.

Indice

Elenco delle figure

Elenco dei Listati

0.1 Progetto Cupid

Il presente lavoro è basato sul progetto Europeo denominato CuPiD.
CuPiD è un progetto Europeo guidato dall'Università di Bologna, della durata di tre anni, il cui obbiettivo è quello di rendere disponibili strumenti tecnologici in grado di guidare attraverso esercizi interattivi personalizzati pazienti affetti da morbo di Parkinson.
In particolare questo progetto si occupa di fornire un sistema di riabilitazione - home-based costituito da una rete di sensori indossabili e un'unità di elaborazione centrale. Tale sistema ha come proposito quello di supportare il paziente fornendo feedback audio durante l'esecuzione degli esercizi riabilitativi e permette la supervisione delle sue attività da remoto. I risultati sperimentali e i dati acquisiti saranno utilizzati ai fini di ricerca scientifica su questa patologia.

Dal progetto CuPiD ci si aspettano i seguenti principali risultati:

- Dimostrare e quantificare l'influenza di un appropriato allenamento sul morbo di Parkinson
- Fornire uno strumento Smart in grado di rendere possibile lo svolgersi della terapia in un ambiente domestico
- Fornire per l'ambito di ricerca uno strumento più accurato, in grado di correlare terapie ad effettiva efficacia per mezzo dell'analisi dei punteggi motori nei pazienti
- Sviluppo di nuovi algoritmi di Bio-FeedBack
- Fornire un Personal Health System innovativo, in grado di elaborare localmente i dati per fornire terapie personalizzate più velocemente e più accuratamente
- Svincolare il paziente da un supporto medico costante durante l'esercizio
- Contribuire all'utilizzo di standard nel campo della Health Information Technology

0.2 Dispositivi

Nel complesso il sistema è costituito da tre parti: Un Nodo Centrale (CN) e due nodi sensori remoti (RSN). Il dispositivo oggetto del seguente testo è il CN. Questo consiste di un circuito stampato di dimensioni ridotte, indossabile, il cui obbiettivo è sia la raccolta dati tramite appositi sensori On-Board, sia di fungere da ricevitore per la raccolta (senza fili) di ulteriori acquisizioni inviate dai nodi sensori. (generalmente collocati alle caviglie del paziente).
Il CN di CuPiD ha l'obbiettivo di fornire tre servizi essenziali:

- Feedback visivo, tattile e uditivo durante l'esercizio del paziente
- Stimolazione audio esterna per avvertire il paziente della condizione sintomatica di Freezing of Gait (FOG) ed allenare la sua prevenzione
- Allenare le funzioni moto-cognitive tramite Audio-BioFeedback

Al fine di fornire questi servizi il CN si avvale dei seguenti **supporti Hardware** principali:

- Micro Controller Unit, STM32F4
- Barometric Pressure and Temperature Sensor, MS5611
- Inertial Measurement Unit, MPU-9150A
- Bluetooth Module, SPBT2632C1A
- Piccolo Motore per la Vibrazione
- Buzzer e connessione auricolari

Di questi solamente i primi quattro saranno interesse di questo testo. Oltre a quelli elencati sono presenti altri componenti, come regolatori di tensione, batteria, memoria flash, etc., che tuttavia non verranno descritti nel dettaglio.

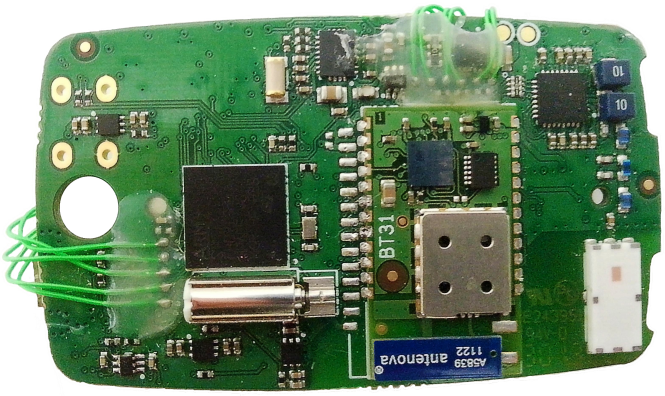


Figura 1: Nodo Centrale, Top

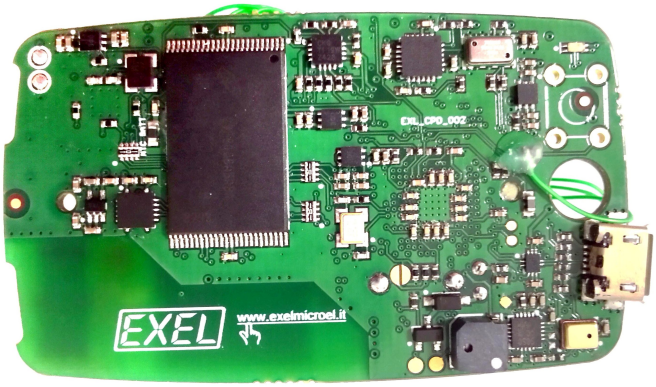


Figura 2: Nodo Centrale, Bottom

1

Strumenti ed Obbiettivi

Passiamo ora all'analisi degli strumenti utilizzati per la realizzazione del progetto. In particolare l'ultima sezione di questo capitolo sarà dedicata alla descrizione più approfondita degli obbiettivi prefissati per il lavoro.

1.1 Hardware del Nodo Centrale

1.1.1 MCU STM32F4

Il Nodo Centrale della rete utilizza, per la gestione delle varie periferiche, l'unità microcontrollore (MCU) **STM32F407IGH6** basata sul processore STM32F407IGH6. Questa MCU consente al contempo una buona gestione dei consumi ed un'elevata potenza di calcolo (fino a 210 Dhrystone MIPS (Milioni di istruzioni al secondo in virgola fissa)) necessaria per un'elaborazione On-Board dei dati raccolti. In particolare può operare ad una frequenza massima di 168 MHz, include una unità Floating Point e rende dispondibili 192(+4) KByte di RAM e 1 MByte di memoria flash. Oltre a queste caratteristiche include varie periferiche On-Chip, necessarie per la gestione degli altri dispositivi del nodo. Di queste, saranno di particolare interesse in questo testo le seguenti:

- General Purpose I/O
- Interfaccia I2C
- Interfaccia USART
- Core Clock

1.1.2 Modulo Bluetooth

Il Nodo Centrale comunica *wireless* tramite uno di due moduli RF possibili. Uno di questi è una radio a 868 MHz, l'altro un modulo Bluetooth (a 2.4 GHz). In questo testo si porrà attenzione solamente sull'utilizzo del modulo Bluetooth.

Il modulo Bluetooth installato sul CN è **SPBT2632C1A** prodotto da ST Microelectronics©. Questo modulo supporta lo standard Bluetooth v3.0 e la comunicazione con esso avviene tramite interfaccia UART. I comandi per la configurazione del modulo e della connessione rientrano nel set denominato AT2; comandi AT2 utilizzati nel testo sono riportati in appendice. La gestione del firmware del modulo è affidata alla MCU ST Micro Cortex-M3. Una delle caratteristiche più interessanti di questo modulo, per il lavoro svolto in questo testo, è la possibilità di stabilire *connessioni Multipoint* tra un Master e due Slave simultaneamente. Questo come si vedrà è il caso di utilizzo per questo lavoro. Altre caratteristiche per questo modulo sono la presenza di un'antenna integrata che quindi lo rende totalmente indipendente e di facile uso una volta alimentato e connesso ad una interfaccia UART.

1.1.3 IMU e Sensore di Pressione

La parte sensoristica del CN di CuPiD è costituita da due circuti integrati. Il primo è una Inertial Measurement Unit (IMU) e il secondo un sensore di pressione barometrica e temperatura (PTS).
La IMU è la **MPU-9150** prodotta da *InvenSens*©. Questo circuito integrato racchiude un sistema di 3 MEMS a 3 assi ciascuno: un Giroscopio, un Accelerometro e separatamente, ma sempre sullo stesso chip, un Magnetometro. In questo dispositivo è inoltre incluso un Digital Motion Processor che funge da acceleratore hardware che tuttavia non sarà utilizzato in quanto non necessario. La lettura di questi sensori avviene quindi On-Chip, così come la conversione digitale tramite ADC dedicati per ognuno dei tre MEMS. Questa unità supporta anche una modalità Low Power e comunica verso l'esterno tramite bus I^2C i campioni in uscita a ciascun ADC. Nella seguente tabella sono riportati risoluzione e fondo scala per i tre sensori integrati:

Sensore	Risoluzione	Range
Giroscopio	$(8,15,30,61)\cdot 10^{-3}$ °/s	±250, ±500, ±1000, ±2000 °/s
Accelerometro	$(6, 12, 24, 48) \cdot 10^{-2} g$	±2g, ±4g, ±8g and ±16g
Magnetometro	0.3 μT	±1200 μT

Il Sensore di pressione barometrica e temperatura è il **MS5611-01BA03** della *Measurement Specialties*©. La principale caratteristica di questo sensore è l'alta precisione equivalente, di circa 0.1 metri, necessaria per il riconoscimento di condizioni patologiche, quali ad esempio cadute, nel paziente. Questo integrato include un sensore di pressione e temperatura ad alta linearità e un ADC $\Delta\Sigma$ ultra low power a 24 bit. La comunicazione dei campioni in uscita dall ADC avviene tramite

interfaccia SPI o I2C. Nella seguente tabella sono riportate risoluzione e range delle misure di questo sensore:

Sensore	Risoluzione	Range
Pressione	$(6.5,4.2,2.7,1.8,1.2)\cdot 10^{-2}$ mbar	$10 \div 1200$ mbar
Temperatura	$1 \cdot 10^{-2}$ C	$-40 \div 85$ C

Da specificare, il fatto che le letture in formato digitale in uscita sul bus I^2C non sono soggette ad una conversione banale per risalire al valore fisico effettivo. La procedura che a partire dalla lettura[1] delle conversioni e di appositi offset di calibrazione restituisce i valori fisici è parte del firmware sviluppato ed è qui riportata:

```
void PT_Convert(U32 D1, U32 D2, const U16* C, int* TEMP, int* PRESS){
  long int off,sens = 0;
  int dT = 0;

  dT = D2-(C[5]<<8);

  *TEMP = (2000+((dT*C[6])>>23));

  off = (C[2]<<16)+((C[4]*dT)>>7);
  sens = (C[1]<<15)+((C[3]*dT)>>8);

  *PRESS = ((((D1*sens)>>21)- off)>>15);
}
```

Listato 1.1: PT_Convert: conversione temperatura e pressioni reali

Questa procedura rispecchia le indicazioni di conversione espresse sul datasheet del sensore e scrive, nelle variabili passate per riferimento TEMP e PRESS, i valori convertiti di temperatura e pressione rispettivamente.

1.2 RL-ARM Real Time Operating System

Analizziamo ora, in questa sezione, le componenti fondamentali e la struttura di un Sistema Operativo Real-Time. Questa parte non verrà sviluppata nel dettaglio in quanto non obbiettivo di questo testo. Saranno analizzati i concetti base delle strutture software maggiormente utilizzate per realizzare il progetto. Per informazioni più approfondite rimandiamo al testo riportato in bibliografia.

[1]Nella procedura di conversione: D1 e D2 sono le misure successive lette dal sensore, mentre il vettore C contiene i parametri di calibrazione letti durante l'inizializzazione del sensore

1.2.1 Utilizzo di un RTOS e vantaggi

La realizzazione del firmware di questo progetto è basata sull'uso di un Sistema Operativo Real-Time, sviluppato da ARM© e costituito essenzialmente da un Kernel denominato (**RTX**) e da librerie accessorie per la gestione semplificata di memoria Flash, protocollo TCP/IP, interfaccia CAN e interfaccia USB. In particolare, in questo progetto, verrà fatto largo uso delle funzioni rese disponibili dal Kernel e non delle librerie in quanto riferite a periferiche non necessarie per il progetto. Il Kernel RTX consiste essenzialmente di uno **scheduler** e definizioni e strutture di costrutti fondamentali, quali **task** e altri componenti software, che vengono gestiti dal Kernel stesso.
L'approccio di programmazione classico per il firmware di un microcontrollore prevede una fase di inizializzazione seguita da una parte nella quale si ripetono ciclicamente tutte le istruzioni in quello che viene definito "Super-Loop". Gli svantaggi di questo paradigma di programmazione sono essenzialmente:

- Dover ricorrere spesso a variabili globali per lo scambio di informazioni tra sezioni di codice
- Dover accordare l'esecuzione ciclica con la gestione degli interrupt nel caso di ISR con forte carico computazionale
- Staticità nel ciclo di esecuzione che si può ripetere solo uguale a se stesso
- Difficoltà nell'implementazione di progetti complessi dal punto di vista del numero di mansioni e della loro sincronizzazione
- Difficoltà e spreco di risorse nell'implementazione di operazioni cicliche con diversi tempi di esecuzione

A questi punti deboli, e ad altri, pone in buona parte rimedio l'approccio proposto dall'utilizzo di un RTOS. In particolare elenchiamo alcuni aspetti di interesse:

- Strumenti di comunicazione inter-processo per lo scambio di informazioni
- Interrupt handler brevi grazie alla delega della routine a processi accessori
- Esecuzione ciclica dinamica di processi a struttura modulare (*task*)
- Gestione più semplificata e ad alto livello di progetti complessi
- Gestione semplificata delle temporizzazioni con un minor impiego di risorse associato

Inoltre il grande vantaggio di questo nuovo paradigma di programmazione è dato dalla struttura modulare del software, più simile ad una gestione ad oggetti, che consente di avvicinarsi alla "illusione" di un'esecuzione parallela delle operazioni (*multitasking*) e ad una manutenzione più semplificata del codice, al costo di una più complessa inizializzazione e gestione delle dipendenze tra i moduli sviluppati. L'impatto sulle risorse hardware del sistema risulta ovviamente maggiore rispetto

ad un approccio software classico (dato che è necessario dedicare risorse al Kernel), tuttavia in un microcontrollore a 32 bit questo impatto è spesso trascurabile rispetto ai benefici; a maggior ragione quando il *core* è un Cortex-M4 come nel nostro caso. Ulteriori informazioni riguardo all'occupazione di memoria di RTX sono riportate in appendice.

1.2.2 Task

Struttura e comportamento

L'elemento software di base di un RTOS è il task. Il task si identifica come un'istanza indipendente che comunica tramite appositi strumenti software con altri processi e la cui esecuzione è gestita dallo scheduler del RTOS. I task vengono usati per svolgere le varie attività prese in carico dalla MCU, sostituendo il paradigma di programmazione sequenziale classico o "super-loop". Questi possono essere chiamati da altri task e hanno una certa durata di esecuzione, possono essere eliminati oppure messi in attesa, il tutto sulla base di un sistema di richieste e priorità affidato allo scheduler stesso. La politica di priorità adottata dallo scheduler è configurabile nel file *RTX_Config.c* assieme ad altri parametri necessari per l'inizializzazione e la gestione del sistema operativo. In particolare il livello di priorità del singolo processo può essere cambiato dinamicamente rendendo la gestione del sistema più dinamica e introducendo la possibilità di implementare modalità a basso consumo con maggiore facilità.
All'attivazione, ogni task esegue le istruzioni in uno spazio di memoria dedicato definito **task stack**. Fisicamente questa area è collocata sulla RAM della MCU per mantenere veloce l'esecuzione del processo stesso in operazioni di scrittura e lettura. Le seguenti linee di codice rappresentano il formato di un task in RL-ARM:

```
__task void taskname (void){
    //inizializzazione
  for(;;){
    //task routine
  }
}
```

Listato 1.2: Formato di un task

Come detto è lo scheduler a gestire l'esecuzione dei task e nel suo operare si appoggia ad una tabella di priorità dei processi configurabile da utente. Da sottolineare il fatto che RTX consente ad un solo processo di essere in fase RUNNING; gli altri o sono READY cioè pronti per il loro turno di esecuzione, oppure sono WAITING e attendono un particolare evento nel sistema. Gli eventi READY quindi passano

(secondo priorità) allo stato RUNNING uno alla volta, ciclicamente, ad intervalli temporali molto ristretti fornendo l'illusione di un sistema Multi-tasking. Nella seguente immagine è riportato uno schema della struttura descritta:

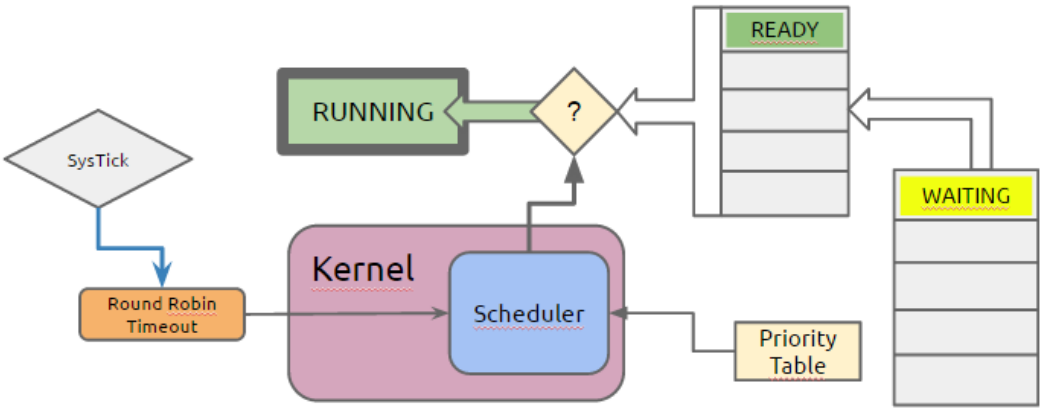


Figura 1.1: Scheduler: diagramma del funzionamento

Senza scendere in dettagli non propri di questo testo, di seguito vengono analizzati brevemente alcuni elementi del Kernel, legati ai cicli di esecuzione dei task, dei quali si è fatto uso nel progetto corrente.

Gestione ad Eventi

Al momento della creazione un task conta 16 **event flag**. Tramite le flag, è possibile sospendere l'esecuzione di un task fino a che una o un gruppo di esse non viene settato da un altro task o da un evento nel sistema. In questo modo un processo resta in uno stato di WAIT_EVNT (gestito dallo scheduler) fino allo scadere di un timeout, o fino a che una o più flag predefinite vengono attivate. L'uso delle flag è quindi un semplice ed efficiente metodo per il controllo delle azioni fra i vari task e per la gestione coordinata degli interrupt in un RTOS. In particolare questo strumento può essere utile per gestire più dinamicamente la sincronizzazione fra i vari task.

Mailbox System

Come appena visto, RTX fornisce strumenti per il "trigger" di un certo task da un secondo, grazie al modello delle flag e della gestione ad eventi. Oltre a questo tipo di interazione, RTX supporta un costrutto software che consente sia l'attivazione, che lo *scambio di dati tra task*: la **Mailbox**. Questo strumento è molto utile nella realizzazione di un programma multi-task come è questo il caso. Senza scendere troppo nei dettagli, la struttura essenziale di questo costrutto è composta da una serie di funzioni che servono per allocare ed inizializzare un buffer detto **Mailbox**, diviso in un dato numero di **Slot**. L'oggetto Mailbox inizializzato viene utilizzato

da altre funzioni che consentono di costruire e ricevere **Messaggi** tramite l'utilizzo di appositi puntatori. Il tipo di dati scambiati è configurabile dall'utente come una struttura dati qualsiasi, a seconda delle necessità. Questo meccanismo è utile soprattutto in situazioni in cui due o più task inviano, anche simultaneamente, ad un terzo task dei dati da elaborare. Nel caso di ricezioni multiple, infatti, i messaggi vengono salvati ordinatamente nel buffer della Mailbox gestito con politica FIFO. In questo modo si evita la perdita di informazioni mantenendo lineare l'esecuzione del software ed un'elaborazione ordinata in uscita. Nella seguente immagine è rappresentato in sintesi il funzionamento:

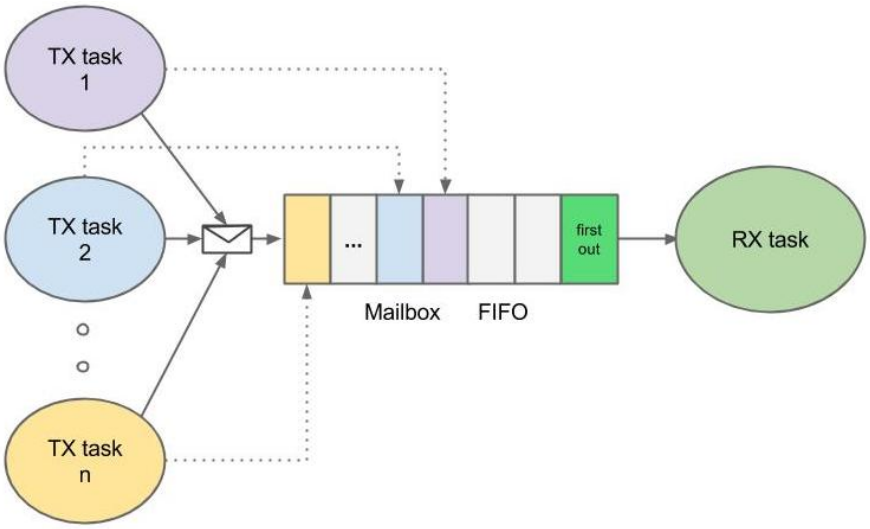


Figura 1.2: Mailbox System: diagramma del funzionamento

Mutex

Un ulteriore strumento di grande utilità fornito da RTX è il Mutex. Essenzialmente è un caso particolare del più noto meccanismo denominato "semaforo". In particolare viene dichiarata una struttura dati apposita contenente un **token** virtuale. In generale il token rappresenta la *possibilità di controllare* una periferica o un processo non utilizzabile da più client contemporaneamente. Nel caso di un RTOS il primo task che esegue la chiamata per il controllo acquisice il token e svolge la sua routine. Nel frattempo gli altri task che tentano di controllare la periferica o il processo oggetto del Mutex vengono messi in attesa. Quando la routine del primo task termina, il token è restituito e si ripete il ciclo. Il principio di funzionamento è illustrato nella seguente immagine.

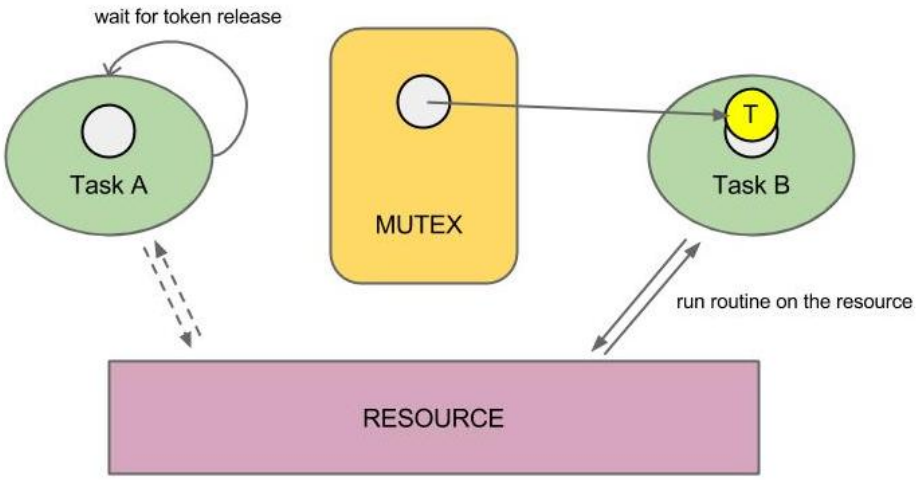


Figura 1.3: Mutex: diagramma del funzionamento

1.2.3 Gestione delle Interrupt Request (IRQ)

La gestione degli interrupt, utilizzando RTX, differisce leggermente rispetto al caso standard. Per non trattenere troppo a lungo l'esecuzione all'interno degli Handler delle IRQ, col rischio di scoordinare l'attività dello scheduler, si preferisce mantenere minimo il tempo speso in essi. Di conseguenza nella Interrupt Service Routine (ISR), funzione che risponde all'arrivo dell'interrupt, si tende a ridurre al minimo il carico computazionale impostando solamente una flag che porta all'attivazione di un task ad alta priorità, gestito dallo scheduler, che viene usato come Handler vero e proprio.
Nel task attivato dalla ISR vengono quindi svolte tutte le elaborazioni necessarie senza il rischio di intralciare l'attività dello scheduler. Il tutto viene quindi regolato secondo la politica standard dei task in RTX e l'Handler vero e proprio funge solamente da "trigger".

1.3 Comunicazione con i Sensori

La comunicazione con i sensori sul Nodo Centrale avviene, sia per l'unità adibita alle misure inerziali (IMU) che per il sensore di pressione e temperatura (PTS), tramite un'interfaccia seriale I^2C. Entrambi sono collegati al medesimo bus I^2C che li connette alla MCU la quale funge da Master nella comunicazione. La velocità massima per il Bus I^2C è 400 KHz ma in questo progetto viene utilizzata una velocità di 100 KHz, sufficiente per gli scopi. Come da standard per questo protocollo di comunicazione, le due linee SCL e SDA del bus I^2C sono tenute a livello logico alto da due resistori da 10 $K\Omega$, uno ciascuna, tra la tensione di alimentazione (Va) e le linee del bus.

1.3.1 Unità per Misure Inerziali (IMU)

Vediamo ora le caratteristiche della transizione I^2C specifica per la IMU. Come detto la comunicazione avviene tramite il bus I^2C, tuttavia le sequenze di scrittura e lettura differiscono leggermente tra i vari dispositivi. In particolare viene ora descritta la sola sequenza di lettura del Sensore; questo in quanto di maggior interesse per il progetto e in quanto la procedura di scrittura ricalca esattamente i primi passi di quella di lettura.
Considerando il caso di lettura di due Byte, l'ordine delle azioni che la MCU, Master della comunicazione, deve compiere è il seguente:

1. Invio della **Start Condition**, portando la linea dei dati a livello logico basso
2. Invio dell'indirizzo I^2C della IMU (7-bit), seguito dal **bit R/W posto a 0** (modalità scrittura)
3. Attesa del acknowledge (**ACK**) dalla IMU
4. Invio dell'indirizzo del primo dei due **Registri Interni** che si intende leggere (in questa modalità di lettura gli indirizzi interni sono incrementati automaticamente dopo ogni lettura)
5. Attesa del ACK dalla IMU
6. Invio della Start Condition
7. Invio dell'indirizzo I^2C della IMU, seguito dal bit **R/W posto a 1** (modalità lettura)
8. Attesa del ACK dalla IMU
9. Lettura del byte posto serialmente sul bus dalla IMU (Primo byte)
10. Invio di un ACK da parte della MCU (Master)
11. Lettura del byte messo serialmente sul bus dalla IMU (Secondo byte)
12. Invio di un negative acknowledge **NACK** da parte della MCU [la linea SDA viene lasciata alta]
13. Invio dello **Stop Bit**

Questi passaggi sono sinteticamente riportati nella seguente tabella:

MCU	S	AD+W		RA		S	AD+R			ACK		NACK	P
IMU			ACK		ACK			ACK	DATA1		DATA2		

.

In particolare con S e P vengono indicate rispettivamente la Start e Stop condition, con RA l'indirizzo del registro interno da leggere e con DATA1 e DATA2 i byte restituiti serialmente dal dispositivo letto, mentre le restanti sigle sono già descritte nell'elenco precedente.

Vediamo ora l'indirizzo della IMU e dei suoi registri interni di interesse per questo progetto. L'indirizzo I^2C della IMU è configurabile via hardware nel suo ultimo bit per consentire l'uso di due di questi componenti sullo stesso bus I^2C contemporaneamente. Dal momento che il pin di configurazione dell'indirizzo è, nel nostro caso, collegato a GND (massa), la IMU risulta avere indirizzo **0x68** o 1101000. Da questo deriva che l'indirizzo di accesso ad 8-bit in Write Mode è 0xD0 mentre quello in Read Mode è 0xD1.
In appendice sono riportati i registri interni di interesse per la MPU-9150 in oggetto. I registri notevoli che vengono utilizzati in questo lavoro rappresentano l'essenziale per un corretto funzionamento della lettura delle grandezze campionate dal sensore.
In particolare, va posta attenzione al Magnetometro interno alla MPU-9150 il quale, pur essendo sullo stesso chip, è esterno all'unità al contrario di Accelerometro e Giroscopio.
Tra i registri di interesse vi sono quindi quelli che consentono il collegamento del Magnetometro al bus I^2C principale per una corretta lettura.
I registi che invece modificano sampling rate, scala dei valori misurati, correzioni e calibrazioni non vengono trattati, in quanto la regolazione di questi parametri non è obbiettivo di questo testo.

1.3.2 Sensore di Pressione e Temperatura

Come per l'unità IMU, vediamo ora la comunicazione del sensore di Pressione e Temperatura. Come visto, anch'esso è connesso allo stesso bus I^2C della IMU. Il sensore dispone sia di interfaccia SPI che I^2C. Per selezionare la modalità I^2C viene collegato alla tensione di alimentazione il pin Protocol Select (PS) del sensore. In questa modalità il LSB dell'indirizzo I^2C della periferica è il complemento del pin di Chip Select (CSB); nel nostro caso questo pin è collegato a GND di conseguenza l'inditizzo termina con LSB = 1. L'indirizzo del sensore sul bus I^2C risulta essere **0x77** ovvero 1110111. Vediamo ora la lettura dei valori di pressione e temperatura per questo disposivo. La sequenza dei comandi da eseguire è riportata di seguito. In particolare viene qui presa ad esempio la lettura della pressione.

1. Invio della **Start Condition**
2. Invio dell'indirizzo I^2C del sensore (7-bit), seguito dal **bit R/W messo basso** (modalità scrittura)
3. Attesa del **ACK**nowledge dal sensore
4. Invio del comando di Conversione Pressione,(8-bit)
5. Attesa del ACK dal sensore

6. Invio della Stop Condition
7. Invio della Start Condition
8. Invio dell'indirizzo I^2C del PTS, seguito dal bit **R/W messo basso** (modalità scrittura)
9. Attesa del ACK dal sensore
10. Invio Comando lettura ADC
11. Invio della Stop Condition
12. Invio della Start Condition
13. Invio dell'indirizzo I^2C del PTS, seguito dal bit **R/W messo alto** (modalità lettura)
14. Attesa del ACK dal sensore
15. Lettura primo Byte dal bus I^2C da parte della MCU
16. Attesa del ACK dalla MCU
17. ⟨ *vengono ripetuti i passi 14 e 15 per i Byte successivi* ⟩
18. Lettura quarto Byte dal bus I^2C da parte della MCU
19. Invio di un **NACK** da parte della MCU (la linea SDA viene lasciata alta)
20. Invio dello **Stop Bit**

In particolare, il comando di conversione per la lettura della pressione risulta essere **0x48** mentre per la temperatura **0x58**. Per entrambi i casi il risultato della conversione viene letto tramite il comando **0x00** (ADC Read). Di seguito è riportata la tabella riassuntiva della transazione I²C descritta:

MCU	S	AD+W		CONV		P	S	AD+W		ADC	P	S	AD+R			ACK	..		NACK	P
PTS			ACK		ACK				ACK					ACK	DATA1		..	DATA4		

.

I caratteri nella tabella hanno lo stesso significato descritto precedentemente per la lettura della IMU; differiscono solamente CONV e ADC che sono rispettivamente i comandi di inizio conversione e lettura del risultato appena commentati.

1.4 Comunicazione Bluetooth

Analizziamo ora nel dettaglio la procedura per la creazione della connessione **Bluetooth Multipoint** tra un Master e due Slave. Facciamo riferimento al caso in cui il master conosce a priori il BD Address dei due nodi ai quali si deve collegare.

1.4.1 Inizializzazione moduli

Il Nodo Centrale funge da Master per la Comunicazione. All'accensione il modulo Bluetooth viene resettato tramite il suo pin di *RESET* mappato sul Pin7 della GPIOB della MCU. Dopo questa operazione, il modulo risponde sulla USART6 con la stringa *AT-AB -CommandMode-* seguita da una seconda *AT-AB BDAddress [BDAddr]*; dove BDAddr è il MAC Address del Modulo Bluetooth sottoposto a Reset.
Una volta resettato, occore configurare le varie opzioni di comunicazione, in particolare, in questo caso, la modalità Multipoint. Operativamente, tramite UART, viene passato il comando per settare la variabile 35 (MPMode) e quindi attivare il Multipoint Mode. Per quanto riguarda il **Master** la stringa di configurazione è la seguente:

AT+AB Config MPMode = 1

Per quanto riguarda i due **Slave**, allo stesso modo, devono essere entrambi inizializzati con la medesima stringa. Nel comando visto: 0 indica una normale connessione Point To Point, 1 rappresenta la connessione Multipoint, mentre 2 è utilizzato per una connessione di tipo Broadcast.

1.4.2 Connessione Multipoint

Una volta inizializzati il Master e i due slave, si può procedere all'invio, tramite UART, dei comandi per consentire il collegamento Bluetooth tra i tre dispositivi (creazione della WBAN). Supponendo che i nodi sensori accettino automaticamente i tentativi di connessione(opzione di default) e, come già detto, che il Master conosca il BD Address di questi, si procede come segue:

1. Il Master in *Command Mode* intenta una **connessione Seriale Point to Point** con il primo Slave

2. Una volta stabilita esce dal *Bypass Mode* tramite la **Escape Sequence**

3. Nuovamente in *Command Mode*, intenta una connessione Seriale Point to Point con il secondo Slave

- Una volta stabilità si trova nuovamente in Bypass Mode con due canali Bluetooth aperti verso i due Slave.

Di seguito vediamo le stringhe di **comandi AT** [2] inviate dal Master per eseguire queste operazioni:

1. *AT+AB SPPConnect [BDAddr Slave0]*

2. ^#^$^%

3. *AT+AB SPPConnect [BDAddr Slave1]*

Ogni comando è seguito da una risposta, di successo o fallimento, da parte del modulo stesso. Queste istruzioni devono essere inviate ad intervalli temporali più o meno lunghi (fino a 3-4 secondi per *SPPConnect*), per consentire al modulo Bluetooth di portare a termine l'istruzione corrente prima della successiva[3] e ai nodi sensori di rispondere al tentativo di connessione.
Infine va specificato che la comunicazione diretta tra Slave non è possibile in quanto questi vedono, come da configurazione, una linea Point To Point diretta con il solo Master. Infatti, il motivo per cui è necessaria una connessione Multipoint è essenzialmente che la comunicazione Point to Point tra tre nodi differenti dovrebbe prevedere la connessione tra due di questi, il distacco e la connessione tra gli altri due. Questa procedura inserisce delle forti latenze se paragonate ai tempi di trasmissione del singolo pacchetto; di conseguenza risulta inattuabile.

1.4.3 Trasmissione Ricezione e Pacchetto Dati

Una volta stabilito il collegamento Bluetooth tra il Nodo Centrale Cupid e i due nodi sensori, lo scambio dati può avvenire in maniera **Bidirezionale** e deve rispettare determinate caratteristiche.
Nella WBAN creata, basata su Multipoint, ogni Slave è identificato da un ID che lo distingue dagli altri connessi al medesimo Master. Questo è necessario per risalire alla provenienza dei pacchetti. Nel nostro caso i due nodi remoti connessi avranno **ID pari rispettivamente a 0 e 1**.
L'invio dei pacchetti, sia in direzione Master → Slave che Slave → Master, deve inoltre rispettare un formato prestabilito. Si possono inviare fino a **315 Byte per pacchetto** (o 315 caratteri). Qualsiasi sequenza di Byte inviata deve essere preceduta da un **header di 4 caratteri** dove:

- Il primo carattere è una cifra da 0 a 9 indicante l'**ID dello Slave** che manda o riceve il pacchetto

[2] **N.B.** Tutte le istruzioni AT vanno terminate con il doppio carattere di terminazione $\backslash r \backslash n$. Fa eccezione la Escape Sequence.

[3] Un'approccio più rigoroso vorrebbe un attesa delle stringhe di risposta del Modulo e un controllo di queste, anzichè un semplice delay e un controllo finale a connessione avvenuta

- I seguenti tre caratteri(ASCII) devono esprimere in cifre decimali il **numero di Byte che vengono inviati** (fino a 315)

Di conseguenza un possibile pacchetto[4] prende la forma riportata nella seguente figura:

ID	LENGHT	DATA
0	007	esempio

Da ribadire il fatto che non solo il Master deve apporre l'header ai pacchetti ma anche gli Slave devono identificarsi nell'invio dei dati.

1.5 Obbiettivi

Dopo aver presentato l'Hardware e il Software dei quali si farà uso nel processo ed il loro funzionamento, vediamo ora gli obbiettivi di questo progetto.
Quello che si vuole ottenere per il dispositivo è un firmware funzionante dal punto di vista della comunicazione e della lettura dei sensori e programmato su Sistema Operativo Real-Time. Vengono quindi prefissati i seguenti obbiettivi:

- Lettura delle grandezze fisiche da **IMU** e **PTS** tramite interfaccia I^2C
- Creazione di una **WBAN** con modalità Bluetooth Multipoint tra il Nodo Centrale e i nodi sensori remoti
- Invio di **comandi** ai nodi tramite la connessione stabilita
- **Ricezione** contemporanea di dati da entrambi i nodi remoti attraverso la WBAN creata
- Progetto dei punti precedenti come moduli utilizzabili su **Sistema Operativo Real-Time**
- Valutare qualitativamente consumi e affidabilità del sistema ottenuto

[4] **N.B.** nell'esempio fornito la parola "esempio" è di 7 lettere, tuttavia solitamente ogni stringa si conclude con i caratteri \r e \n dei quali occorre tenere conto nel campo lenght, che in questo caso sarebbe quindi 009

Per quanto riguarda l'ultimo punto, questo non rientra nel progetto del firmware vero e proprio e si intende un'analisi di primo livello per quanto riguarda le effettive prestazioni del sistema, in grado di fornire indicazioni sulla efficacia dell'hardware scelto.

2

Realizzazione Software

Viene ora analizzato il lavoro svolto per il raggiungimento degli obbiettivi appena esposti. Questa analisi sarà suddivisa come esposto nell'indice a inizio testo. In particolare verranno trattati in ordine: l'inizializzazione del Nodo Centrale e dei suoi dispositivi, la creazione della connessione con i nodi sensori ed infine i task in esecuzione nel RTOS, che costituiscono la parte centrale del firmware.

2.1 Inizializzazione

A seguire è illustrata la parte di codice dedicata all'inizializzazione della MCU e delle sue periferiche, del RTOS e dei Sensori sul bus I^2C. Inoltre, data la semplicità riportiamo subito il *main* del programma:

```
int main (void)
{
  RCC_SetHseClock (CRYSTAL_VALUE);
  HAL_System_Start (CRYSTAL_VALUE, TARGETSYSFREQUENCY, &systemFreq);
  os_sys_init_prio(Init,TSK_PRIO_IRQ);                    /*
   Initialize RTX and start init     */

  while (1); // never reaches this point\\
}
```

Listato 2.1: main function

Possiamo notare come il main sia utilizzato unicamente per impostare i parametri di clock per la MCU ed inizializzare il RTOS tramite il primo task: **Init**. La frequenza di funzionamento utilizzata, TARGETSYSFREQUENCY , è nel nostro caso **120 MHz** (dei 168MHz massimi configurabili).

2.1.1 I²C e Sensori

Al fine di leggere i due sensori presenti sul Nodo Centrale (CN), nella procedura di inizializzazione, vengono operati i passaggi necessari per confgiurare l'unico bus I²C sulla scheda. A tale bus sono connessi sia il Sensore Inerziale a tre assi (IMU), sia il Sensore di Pressione e Temperatura precedentemente descritti. L'interfaccia I²C della MCU è settata dalla procedura di inizializzazione **I2C_Sens_Setup**. Questa procedura setta la seguente configurazione:

```
I2C_InitStruct.I2C_ClockSpeed = 100000;       // 100kHz speed
I2C_InitStruct.I2C_Mode =I2C_Mode_I2C;        // I2C mode
I2C_InitStruct.I2C_DutyCycle = I2C_DutyCycle_2; // 50% duty cycle
I2C_InitStruct.I2C_OwnAddress1 = 0x00;        //Master Address
I2C_InitStruct.I2C_Ack = I2C_Ack_Enable;      // Ack when reading
I2C_InitStruct.I2C_AcknowledgedAddress =
  I2C_AcknowledgedAddress_7bit;               // 7-bit I2C address
```

In particolare, la velocità scelta per il bus, 100 KHz non è la massima ottenibile (400 KHz), ma risulta tuttavia più che sufficiente in quanto questa velocità di trasferimento dati è già molto maggiore rispetto alla frequenza di campionamento (lettura dei sensori) che si intende usare (tra i 50 e i 100 campioni al secondo). Dopo l'inizializzazione dell'interfaccia, secondo i restanti parametri riportati nel riquadro, vengono chiamate due funzioni per inizializzare i sensori. Entrambe operano il settaggio di alcuni registri interni per preparare i dispositivi alla lettura.
Queste due funzioni hanno struttura simile e prendono il nome di **I2C_ByteWrite** e **I2C_CmdSend**. La loro struttura è riportata in appendice ed essenzialmente eseguono i passaggi descritti precedentemente [§1.3] per la scrittura via I²C dei sensori. Entrambe restituiscono un esito della scrittura dei registri. Nello specifico ad ogni passo della procedura di scrittura, un contatore tiene traccia del tempo trascorso in attesa della risposta dello slave. Nel caso in cui il contatore raggiunga lo zero prima che sia arrivata la risposta, la funzione si interrompe tramite un *return* con codice di errore. Se anche un solo registro restituisce un errore durante la configurazione, il processo è completamente interrotto e l'inizializzazione viene fermata e l'errore segnalato.
Di seguito sono riportate le due procedure per IMU e PTS:

```
// IMU INIT
I2CStatus IMU_MPU9150_Init(void){
  I2CStatus ret = Error;

  ret = I2C_ByteWrite(MPU9150_MPU_I2C_ADDRESS, 0x80,
    MPU9150_PWR_MGMT_1, I2C1);
  ret |= I2C_ByteWrite(MPU9150_MPU_I2C_ADDRESS,0x07,
    MPU9150_SIGNAL_PATH_RESET, I2C1);
```

```
  ret |= I2C_ByteWrite(MPU9150_MPU_I2C_ADDRESS,0x09,
   MPU9150_SMPLRT_DIV,I2C1);
  ret |= I2C_ByteWrite(MPU9150_MPU_I2C_ADDRESS,0x02,MPU9150_CONFIG,
   I2C1);
  ret |= I2C_ByteWrite(MPU9150_MPU_I2C_ADDRESS,0x00,
   MPU9150_GYRO_CONFIG,I2C1);
  ret |= I2C_ByteWrite(MPU9150_MPU_I2C_ADDRESS,0x00,
   MPU9150_ACCEL_CONFIG,I2C1);
  ret |= I2C_ByteWrite(MPU9150_MPU_I2C_ADDRESS,0x00,MPU9150_FIFO_EN,
   I2C1);
  ret |= I2C_ByteWrite(MPU9150_MPU_I2C_ADDRESS,0x00,
   MPU9150_INT_ENABLE,I2C1);
  ret |= I2C_ByteWrite(MPU9150_MPU_I2C_ADDRESS,0x00,MPU9150_MOT_DUR,
   I2C1);

  // Bypass to Magnetometer
  ret |= I2C_ByteWrite(MPU9150_MPU_I2C_ADDRESS,0x02,
   MPU9150_INT_PIN_CFG,I2C1);
  ret |= I2C_ByteWrite(MPU9150_MPU_I2C_ADDRESS,0x20,MPU9150_USER_CTRL
   ,I2C1);

  // MPU-9150 wake up
  ret |= I2C_ByteWrite(MPU9150_MPU_I2C_ADDRESS,0x00,
   MPU9150_PWR_MGMT_1,I2C1);

  // Magnetometer Self Test
  ret |= I2C_ByteWrite(MPU9150_CMPS_I2C_ADDRESS7, 0x00,
   MPU9150_CMPS_CTRL,I2C1);
  ret |= I2C_ByteWrite(MPU9150_CMPS_I2C_ADDRESS7, 0x0F,
   MPU9150_CMPS_CTRL,I2C1);
  ret |= I2C_ByteWrite(MPU9150_CMPS_I2C_ADDRESS7, 0x00,
   MPU9150_CMPS_CTRL,I2C1);

  // Magnetometer Single Conversion Mode
  ret |= I2C_ByteWrite(MPU9150_CMPS_I2C_ADDRESS7, 0x01,
   MPU9150_CMPS_CTRL,I2C1);

  return ret;
}
```

Listato 2.2: IMU_MPU9150_Init: inizializzazione sensore inerziale

In particolare, nella *IMU_MPU9150_Init*, le prime scritture dei registri sono dedicate alla configurazione di Accelerometro e Giroscopio, mentre la configurazione dei registri MPU9150_INT_PIN_CFG e MPU9150_USER_CTRL serve a collegare il Magnetometro al bus I²C principale. Questo affinché sia possibile la sua lettura

come terzo sensore, indipendente (pur essendo, come visto, sullo stesso integrato di Accelerometro e Giroscopio).
Viene qui riportata la seconda funzione di inizializzazione per il Sensore di Pressione e Temperatura.

```
I2CStatus PTSens_MS5611_Init(void){
  I2CStatus ret = Error;
  U8 i = 0;

  U8 PROM[16]= {0};

  /* MS5611 Reset*/
    ret = I2C_CmdSend(MS5611_I2C_ADDRESS, MS5611_RESET,I2C1);
      OS_SLEEP_MS (10);

    if(ret == Error) return ret;

  /*MS5611 PROM coefficients read*/
    ret |= I2C_BufferRead(MS5611_I2C_ADDRESS,&PROM[0],
   MS5611_PROM0_READ,2,I2C1);

            /* repeat for PROM1 to PROM13 */

    ret |= I2C_BufferRead(MS5611_I2C_ADDRESS,&PROM[14],
   MS5611_PROM7_READ,2,I2C1);

    //change endianess & save in buffer
    for(i = 0 ; i < 8 ; i++){
      MS5611_COEFF[i] = PROM[2*i+1]|(PROM[2*i]<< 8);
    }

  return ret;
}
```

Listato 2.3: PTSens_MS5611_Init: inizializzazione sensore pressione e temperatura

Per quanto riguarda il Sensore di Pressione e Temperatura (PTS), è integrata una seconda parte di lettura, in aggiunta alla configurazione dei registri (simile a quella per la IMU). Questa, tramite la **I2C_BufferRead**, legge una PROM interna al sensore contenente i coefficienti di correzione e calibrazione necessari per ricavare la conversione delle letture ad unità fisiche effettive.

2.1.2 USART

La comunicazione tra MCU e modulo Bluetooth avviene tramite l'interfaccia USART6 utilizzata in modalità asincrona (come UART). UART è un protocollo di comu-

nicazione seriale a due fili e nel nostro caso la configurazione della linea è fornita dalla procedura **BT_Serial_Setup** e i parametri utilizzati sono i seguenti:

```
UART_InitStruct.USART_BaudRate = 115200;      //115200 baud
UART_InitStruct.USART_WordLength = USART_WordLength_8b;
UART_InitStruct.USART_StopBits = USART_StopBits_1;   //1 stop bit
UART_InitStruct.USART_Parity = USART_Parity_No;     // No Parity
UART_InitStruct.USART_HardwareFlowControl =
  USART_HardwareFlowControl_None;      // No Flow control
//config both TX & RX
UART_InitStruct.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
```

La velocità di 115200 baud è sufficiente per i nostri scopi. Inoltre essendo trasferito uno stream continuo di dati che saranno poi opportunamente filtrati, non è necessario porre il bit di parità nel pacchetto inviato. Nell'eventualità che un pacchetto contenga errori, verrà scartato in base al formato o filtrato se i valori contenuti non sono consistenti.

2.1.3 Interrupt Requests

Passiamo ora alla configurazione delle interrupt request per il progetto. L'impostazione, in questo caso, riguarda essenzialmente la sola ricezione dei dati via modulo Bluetooth; di conseguenza la configurazione interessa solamente gli interrupt legati all'interfaccia USART6 della MCU che comunica con tale modulo.
In particolare è stato configurato un solo interrupt sulla ricezione dei dati da tale interfaccia. Le istruzioni legate a tale configurazione sono presenti nell' *Init* task e nel task che gestisce la corretta ricezione dei pacchetti. L'impostazione è essenzialmente la seguente:

```
HAL_IRQ_Register(IRQ_UART6,0x0000,ISR_USART6_IRQHandler);
USART_ITConfig(USART6,USART_IT_RXNE,DISABLE);
```

La prima delle due istruzioni inizializza gli interrupt per l'interfaccia USART6 registrando il nome della callback function che verrà usata per servire tali interrupt: **ISR_USART6_IRQHandler**.
La seconda istruzione, invece, è utilizzata per abilitare l'interrupt specifico sullo stato del buffer di ricezione di tale interfaccia. L'utilizzo di questa seconda funzione sarà ripetuto nel programma al fine di abilitare e disabilitare gli interrupt secondo convenienza, per consentire una gestione completa delle richieste da parte del RTOS.
La callback function utilizzata per servire gli interrupt contiene un "filtro" sul tipo di interrupt che consente di considerare solo quelli di ricezione. Il contenuto di tale

funzione sarà riportato e commentato in seguito nella descrizione organica della gestione delle ricezioni dal modulo Bluetooth.

2.1.4 RTOS MailBox e Mutex

Passiamo ora a vedere la configurazione del RTOS vero e proprio e di alcune sue strutture. Le principali opzioni per il Kernel RTX sono configurabili tramite il file **RTX_CONFIG.c**. Di seguito è riportata una tabella riassuntiva delle principali scelte effettuate per il sistema.

Option	Value
Task Configuration	
Number of concurrent running tasks	24
Number of tasks with user-provided stack	20
Task stack size [bytes]	1024
Check for the stack overflow	☑
Run in privileged mode	☐
Tick Timer Configuration	
Hardware timer	Core SysTick
Timer clock value [Hz]	1000
Timer tick value [us]	1000
System Configuration	
Round-Robin Task switching	☑
Round-Robin Timeout [ticks]	5
Number of user timers	5
ISR FIFO Queue size	16 entries

Figura 2.1: RTOS Configuration

Come riportato nell'immagine, viene utilizzato come timer per lo scheduler il Core SysTick (opzione standard). Gli interrupt per questo timer (Timer Clock Value) sono settati a 1000 Hz ovvero ogni 1 ms. La frequenza dei **ticks** è configurata poi a 1000 μs ovvero ogni 1 ms. In particolare la frequenza dei tick determina la granularità delle temporizzazioni attuabili nel sistema. Come già detto, per il progetto è stata scelta, come gestione dello scheduler, la politica di Round Robin. Il tempo dedicato ad ogni task è settato a 5 ticks. In base alle impostazioni appena viste quindi, risulta che il time slice affidato ad ogni task in esecuzione ciclica è:

$$Timeout\_Ticks \times Timer\_Tick\_Value = 5 * 1\ ms = 5\ ms$$

. Oltre a queste impostazioni, nella sezione *Task Configuration* viene impostato a 24 il numero massimo di task in esecuzione parallela e viene fissata la dimensione di default del *task stack* a 1024 Byte. I risvolti di aver impostato questo valore saranno approfonditi in seguito.

Per quanto riguarda gli strumenti software forniti da RTX, si è fatto ricorso a

due di questi in particolare: **Mutex** e **Mailbox**.
I primi, come visto precedentemente [§1.2.2], si rendono utili nell'utilizzo di risorse condivise da parte di più task contemporaneamente. Nello specifico sono stati utilizzati nel progetto due Mutex: uno per il controllo dell'interfaccia USART e l'altro per il controllo del bus I2C1. Nel file *main.c* del progetto vengono dichiarate le strutture corrispondenti tramite le istruzioni *OS_MUT UART6_MUTX* e *OS_MUT I2C1_MUTX*, dove *OS_MUT* è una struttura dati specifica definita nelle librerie di RL-ARM. Una volta dichiarate, le strutture sono quindi inizializzate tramite le funzioni os_mut_*init(UART6_MUTX)* e *os_mut_init(I2C1_MUTX)*. La scelta di utilizzare dei Mutex è stata dettata dal fatto che nel progetto esistono più di un task, eventualmente concorrenti, che richiedono l'utilizzo di queste periferiche. Un esempio sono i processi indipendenti che leggono i vari sensori sul Nodo Centrale.

Il secondo strumento fornito da RTX, e utilizzato nella realizzazione, è il sistema Mailbox, già descritto nel primo capitolo [§1.2.2]. Per il progetto sono state create due Mailbox separate. Queste fungono da buffer rispettivamente per le Trasmissioni (TX) e ricezioni (RX) del modulo Bluetooth sul Nodo Centrale. Riportiamo il codice per la dichiarazione e l'inizializzazione delle due strutture:

```
os_mbx_declare(TXbox, 16);  // declare TX Mailbox
unsigned int TXpool[16*2*sizeof(TXmsg)/4 + 3]; // memory for 16 TXmsg

os_mbx_declare(RXbox, 16);  // declare RX Mailbox
unsigned int RXpool[16*2*sizeof(RXmsg)/4 + 3]; // memory for 16 RXmsg

 //initialize fixed size memory pool for TXbox and RXbox
_init_box (TXpool, sizeof(TXpool), sizeof(TXmsg));
_init_box (RXpool, sizeof(RXpool), sizeof(RXmsg));

 //initialize TXbox and RXbox
os_mbx_init(TXbox,sizeof(TXbox));
os_mbx_init(RXbox,sizeof(RXbox));
```

Come si può vedere dai commenti al codice stesso, vengono dichiarati in successione e inizializzati un **Pool** ed un **Box** per ognuna delle due mailbox di trasmissione e ricezione. In questo caso la dimensione della mailbox per entrambi è stata fissata a 16 **Slot**. Complessivamente, nel codice, le prime quattro istruzioni di dichiarazione sono contenute nel file *main.c* mentre le ultime quattro funzioni di inizializzazione dei *Pool* e dei *Box* sono chiamate nel task **Init** ed ovviamente vengono eseguite prima dell'avvio di un qualsiasi task che utilizzi il servizio Mailbox. In particolare si nota dal testo che vengono passati come argomento nelle varie funzioni descritte

gli elementi *TXmsg* ed *RXmsg*. Questi sono il tipo di variabile che costituisce i messaggi delle due rispettive mailbox e la loro struttura interna, definita da utente, verrà descritta in seguito.

2.2 Connessione Multipoint

Una volta inizializzata la linea seriale UART tra MCU e modulo Bluetooth, la creazione della WBAN tra Il Nodo Centrale e i nodi sensori si compone di due fasi: un reset e riconfigurazione del modulo e una procedura di connessione a partire dai Device Address dei due slave. In particolare vengono ripresi i passi descritti precedentemente [§1.4.2] per la creazione della connessione Multipoint.
Il reset del modulo è di tipo Hardware ed è effettuato tramite il *pin* di reset dello stesso. La procedura per fare questo prevede prima l'alimentazione del componente tramite semplici comandi GPIO nella procedura **BluetoothPowerOn** ed in seguito il reset sempre tramite istruzioni GPIO nella funzione **ResetBluetooth**. Entrambe queste istruzioni sono eseguite a partire dal task *Init* che sarà analizzato nel dettaglio più avanti. Per quanto riguarda la parte di configurazione e setup invece, questa è affidata alla procedura di inizializzazione **BT_Init_MTP_Connection**, anch'essa lanciata dal task *Init* e della quale è riportata nella pagina seguente la struttura[1]:

```
void BT_Init_MTP_Connection(const char* Slave0Addr, const char*
    Slave1Addr){

    char temp[RX_BUFF_SIZE];

    //format 1st conn string
    strcpy(temp,AT_SPP_CONN);
    strcat(temp,Slave0Addr);
    strcat(temp,CRLF);

    //Set Multipoint Mode
    USART_TX(USART6,AT_MTP_MODE);
      OS_SLEEP_MS(WAIT_SHORT);

    //1st slave connection
    USART_TX(USART6,temp);
    USART_TX(USART6,temp);
```

[1] Oltre a questa procedura, ne è stata realizzata una seconda denominata (BT_Init_PTP_Connection), dalla struttura simile, che consente la connessione Point to Point (PTP) con un singolo slave, pensata nelle fasi iniziali del progetto per il test della connessione

```
    USART_TX(USART6,temp);
      OS_SLEEP_MS(WAIT_LONG);

    //back in command Mode
    USART_TX(USART6,AT_ESC);
      OS_SLEEP_MS(WAIT_MEDIUM);

    //format 2nd conn string
    strcpy(temp,AT_SPP_CONN);
    strcat(temp,Slave1Addr);
    strcat(temp,CRLF);

    //2nd slave connection
    USART_TX(USART6,temp);
    USART_TX(USART6,temp);
    USART_TX(USART6,temp);
      OS_SLEEP_MS(WAIT_LONG);
}
```

Listato 2.4: BT_Init_MTP_Connection: setup collegamento Multipoint Bluetooth

Essenzialmente in questa procedura vengono passati tramite interfaccia UART i comandi AT al modulo Bluetooth, che alterna l'invio di questi ad attese più o meno lunghe, a seconda dei tempi di risposta degli slave, tramite la funzione **OS_SLEEP_MS(time)**[2].
Le stringhe inviate dalla MCU al modulo configurano in una prima parte i registri necessari e successivamente cercano di far connettere il dispositivo ai nodi sensori a partire dal loro BDAddress. In particolare le stringhe che iniziano con "*SPP_Connect*", usate per la connessione con gli slave, sono inviate in rapida successione, tre volte, per questioni di sicurezza sulla corretta ricezione da parte di questi ultimi.

2.3 Task

Passiamo ora all'analisi della struttura vera e propria del firmware per il Nodo Centrale (CN), ovvero il nucleo del lavoro descritto in questo testo. Nello specifico in questa sezione viene affrontata una descrizione dei task fondamentali del progetto e del loro ruolo nella lista di obbiettivi descritti alla fine del primo capitolo.

[2]La funzione di delay *OS_SLEEP_MS(time)* accetta come argomento un numero intero positivo che corrisponde al numero di Tick di sistema da attendere prima di proseguire con l'esecuzione del codice. La sua durata fisica quindi dipende dalle impostazioni scelte per le temporizzazioni di RTX

2.3.1 Init Task

Il compito di inizializzare e configurare l'hardware come visto nella sezione precedente, è affidato allo **Init Task**. Questo è in particolare il primo ad essere chiamato all'avvio del RTOS e oltre che dell'inizializzazione del sistema e delle sue periferiche, si occupa di creare e avviare i task che costituiranno l'esecuzione ciclica di funzionamento del dispositivo. L'invocazione dello Init avviene con alta priorità in modo che ci sia la certezza di una corretta inizializzazione e sincronizzazione tra i processi, prima che le mansioni successive siano avviate. Questa che segue è la sua struttura:

```
__task void Init(void){

  //init USART6 interrupt
  HAL_IRQ_Register(IRQ_UART6,0x0000,ISR_USART6_IRQHandler);

  InitGpio();
  I2C_Sens_Setup();
  AudioOff();

  MemsPowerOn();      //power to sensors
    OS_SLEEP_MS(10);
  if( IMU_MPU9150_Init() == Error)   //IMU inint -> I2C Write
    while(1);

  if( PTSens_MS5611_Init() == Error)   //PT sensor inint -> I2C Write
    while(1);

      Vibro_MS(20);    // config is OK feedback

  GPIO_SetBits(GPIOE,GPIO_Pin_6);
  BluetoothPowerOn();        //power to BT module
  ResetBluetooth();        //BT Module Reset

  BT_Serial_Setup();         //BT uart config

  // BT Multipoint Connection start
  BT_Init_MTP_Connection(Slave0_Addr,Slave1_Addr);

  Vibro_MS(20);    // BT connection OK feedback

  //MAILBOX init
   _init_box (TXpool, sizeof(TXpool), sizeof(TXmsg));
   _init_box (RXpool, sizeof(RXpool), sizeof(RXmsg));

   os_mbx_init(TXbox,sizeof(TXbox));
```

```
    // MUTEX init
    os_mut_init (UART6_MUTX);
    os_mut_init (I2C1_MUTX);

  //------------------------- TASK INIT-----------------------------------

     tsk3 = os_tsk_create(IMU_Read_tsk,TSK_PRIO_RR); //IMU read start
     tsk4 = os_tsk_create(LED_tsk,TSK_PRIO_RR);     // blink led
     tsk2 = os_tsk_create(BT_TX_tsk,TSK_PRIO_RR);   //BT TX task start

         //IRQ TASKS

     tsk5 = os_tsk_create(BT_RX_tsk,TSK_PRIO_RR); //BT RX task start

  // init is complete -> self delete Init task
  os_tsk_delete_self();
}
```

Listato 2.5: Init task

Come si può vedere dal codice riportato, dopo un setup di inizializzazione per una MCU (quale potrebbe essere per il caso di una programmazione sequenziale tipica), si passa: prima all'instaurare una connessione con i nodi sensori e successivamente all'inizializzazione delle strutture software necessarie all'interazione tra task nel RTOS, in questo caso servizio MailBox e Mutex. In ultimo luogo vengono creati i task di background per lo svolgimento delle funzioni del firmware, assegnando ad ognuno di essi un **task ID**[3] ed un certo livello di priorità, il quale peso determina l'ordine di esecuzione, che sarà analizzato nel dettaglio alla fine di questo capitolo.

2.3.2 Comandi ai nodi Bluetooth

I nodi sensori remoti, una volta stabilita la connessione, sono comandati tramite due appositi task, creati per lo scopo, quasi identici tra loro e differenti solamente per destinatario del comando (nodo 0 o nodo 1). Nello specifico, questi task sono normalmente inattivi e vengono creati da utente tramite l'apposita funzione di RTX *os_tsk_create(task,prio)* alla quale è passato come primo argomento il nome di uno tra i due processi, a seconda del destinatario. I comandi inviati sono tendenzialmente quelli di inizio e fine acquisizione per i nodi remoti. A seconda del nodo "target" i due task prendono il nome di:

- BT_Slave0Cmd_tsk

[3]Il task ID è necessario per il riconoscimento univoco di un task, soprattutto nel caso di chiamata parallela multipla dello stesso

- BT_Slave1Cmd_tsk

Chiamando in modo generico *BT_SlaveX_Cmd_tsk* i due processi, questi hanno la medesima seguente struttura:

```
__task void BT_Slave0Cmd_tsk(void *cmd){

  char temp[TX_BUFF_SIZE]={0};
  U8 dim=0;
  TXmsg *TXptr;

  for(;;){

  // string format
  FormatString(temp,cmd,0x00);

    TXptr = _alloc_box(TXpool);
    TXptr = temp;

    //send to BT_TX via Mailbox
    os_mbx_send(TXbox,TXptr,INDEF_TIME);

    os_tsk_delete_self(); //self delete task
  }
}
```

Listato 2.6: BT_SlaveX_Cmd_tsk: comandi ai nodi sensori

L'azione di questi task è quella di utilizzare il servizio MailBox, precedentemente descritto [§1.2.2], con l'obbiettivo di inviare una stringa al task di trasmissione *BT_TX_tsk*, che verrà analizzato in seguito. Il formato dei dati inviati consiste di una stringa in chiaro sull'azione da svolgere, alla quale è aggiunto un header secondo le regole[4] sui pacchetti della connessione Multipoint viste precedentemente [§1.4.3]. Nello specifico la funzione *FormatString* (riportata in appendice), a partire dalla dimensione della stringa da inviare, ricostruisce il numero, in cifre decimali, dello header tramite una successione di divisioni e resti.
Come ultima istruzione per questo processo troviamo *os_tsk_delete_self()* che fa si che una volta inviato il comando, il task si auto elimini dalla coda dello scheduler.

Passiamo ora ad analizzare la struttura del codice del task **BT_TX_tsk**, ovvero il processo di trasmissione vero e proprio che riceve stringhe dal task di comando appena descritto, tramite servizio MailBox, e utilizza l'interfaccia UART per co-

[4] Alla stringa devono essere aggiunti quattro caratteri in testa: il primo deve essere l'ID dello slave target, mentre i successivi tre sono un numero da 0 a 315 ed indicano il numero di Byte (caratteri) della stringa che sarà inviata

municare con il modulo Bluetooth del Nodo Centrale. Ne riportiamo il codice per poter commentare la sua struttura:

```
__task void BT_TX_tsk(void){

  TXmsg* TXptr;      //pointer to incoming message
  tsk5 = os_tsk_create(BT_RX_tsk,TSK_PRIO_RR);

  for(;;){
    //wait for new message
    os_mbx_wait(TXbox,(void**) &TXptr,INDEF_TIME);
    os_mut_wait(UART6_MUTX,INDEF_TIME);        //wait for idle USART

      USART_TX(USART6, TXptr);        // TX string message

    os_mut_release(UART6_MUTX);       // release USART6 token

    _free_box(TXpool,TXptr);        //clear message pointer
  }
}
```

Listato 2.7: BT_TX_tsk: trasmissione via Bluetooth

Questo task ad alta priorità (TASK_PRIO_HI) creato in *Init* resta in attesa[5] fino all'arrivo di un messaggio nella Mailbox osservata **TXbox**.
All'arrivo del messaggio passa quindi in stato di READY e, a discrezione dello scheduler, in RUNNING.
Il termine della sua attesa coincide con la lettura del contenuto del puntatore **TXptr**[6], ovvero il messaggio effettivamente ricevuto.
Una volta raccolti questi dati dalla *mailbox*, il task si occupa quindi del suo invio tramite interfaccia USART (attraverso la funzione *USART_TX*[7]) al modulo Bluetooth che si prende carico dell'invio a livello fisico.
Da notare, l'utilizzo di un MUTEX in testa al corpo centrale del task. Questa accortezza è stata utilizzata per consentire di prendere il controllo della USART6 fino al termine delle istruzioni necessarie e, al contempo, di aspettare nel caso la periferica sia in fase di utilizzo da parte di altri task in esecuzione nello stesso momento. Di seguito è riportato un diagramma logico dell'azione delle varie componenti appena descritte.

[5]Passando come parametro di Timeout ad una funzione di tipo *wait* il valore INDEF_TIME = 0xFFFF, l'attesa è prolungata indefinitamente fino al risolversi dell'evento che causa l'attesa
[6]TXptr è un puntatore di tipo TXmsg, dove TXmsg è definito come char
[7]La struttura di questa funzione è riportata in appendice

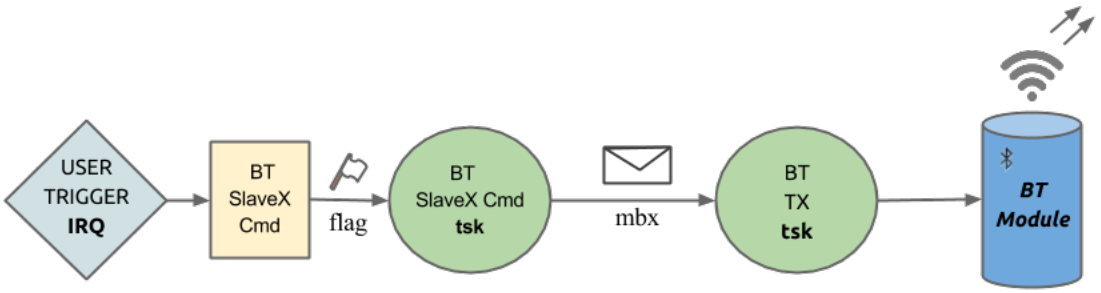


Figura 2.2: Invio comandi via Bluetooth

In particolare il primo blocco di forma romboidale nell'immagine non è implementato in questo firmware in quanto ancora non è nota l'organizzazione ad alto livello dell'operazione di invio dei comandi ai nodi; ovvero ad esempio non è ancora stata determinata quella che sarà la causa dell'invio dei comandi di inizio e fine campionamento ai nodi sensori.

2.3.3 Raccolta Dati

Vediamo ora i task e le strutture preposte alla raccolta dei dati, sia da parte dei nodi remoti che dai sensori presenti sul Nodo Centrale.
L'elaborazione ed il salvataggio dei dati non è oggetto di questo testo, tuttavia viene suggerita la seguente impostazione di base in vista di futuri sviluppi. In particolare questa parentesi si rende necessaria in quanto i task che verranno analizzati in seguito si appoggiano, per il salvataggio e l'invio dei dati, alla seguente struttura:

```
typedef struct RXmsg{
U8 ID;
U8 time;
  union{
    struct{
      IM acc;
      IM gyro;  // for IMU data
      IM cmps;
    };
    struct{
      int temp; // for PTS data
      int press;
    };
  };
}RXmsg;
```

Listato 2.8: RXmsg: struttura per la raccolta dati

Nella struttura proposta i primi due campi sono due interi rispettivamente a 8 e 16 bit. Per i campi struttura seguenti invece è stato scelto di implementare un costrutto *union*. In questo modo infatti è possibile l'utilizzo dello stesso tipo di struttura (*RXmsg*) sia per dati provenienti dalla IMU che per quelli derivati da letture del PTS. In particolare, essendo i dati dalla IMU costituiti da misure sulle tre dimensioni spaziali, per questi viene utilizzata la seguente sottostruttura denominata *IM*:

```
typedef struct IM{
  int16_t x;
  int16_t y;     // Inertial Measure new type
  int16_t z;
}IM;
```

I campi del nuovo tipo RXmsg sono pensati per mantenere allo stretto essenziale il carico informativo del singolo pacchetto. Questo al fine di poter guadagnare, in termini di letture al secondo (soprattutto dalle ricezioni dai nodi remoti) e di spazio di archiviazione dei dati raccolti.
Analizziamo ora in particolare i campi della struttura RXmsg. In primo luogo il campo **ID** viene utilizzato per identificare il mittente del pacchetto, tra Master, Slave0 e Slave1, ai quali corrispondono tre differenti byte[8]. I due bit di maggior peso del Byte ID vengono inoltre utilizzati per distinguere tra pacchetti da una IMU (quando sono zero) e pacchetti da un PTS (quando sono 1). Il secondo campo, **time**, rappresenta la necessità di tenere traccia della posizione temporale dei campioni. Questo accorgimento è preso nell'ottica di una sincronizzazione con le acquisizioni dei nodi sensori e di una registrazione del tempo vero in cui le misure sono state effettuate.
I campi struttura seguenti, come detto, variano a seconda del sensore letto. Nel primo caso questi sono:*acc*, *gyro* e *cmps*. Questi sono costituiti da tre interi a 16 bit ciascuno (uno per ogni direzione spaziale) e rispecchiano le grandezze di interesse lette sia dai sensori del Nodo Centrale, che provenienti dai nodi sensori. Nel secondo caso invece i campi struttura sono *temp* e *press*; entrambi interi a 32 bit necessari per ospitare le relative misure dal PTS.
Una volta registrati i dati in una struttura di tipo *RXmsg*, questa viene utilizzata come puntatore in un apposito sistema Mailbox, **RXbox**. In particolare i task interessati inviano le misure così formattate, sempre tramite Mailbox, ad un task generale di ricezione ed elaborazione **ST_ELAB_tsk()**. Il formato per il task di raccolta dati, nello specifico, può assumere la seguente forma:

[8]Nel caso in cui il pacchetto non provenga dalla lettura dei sensori sul Nodo Centrale, ma dai nodi sensori remoti, il numero di ID è automaticamente rappresentato nello header del pacchetto della comunicazione Multipoint [§1.4.3]

```
__task void ST_ELAB_tsk(void){

  RXmsg* RXptr;       //pointer to incoming message

  for(;;){
  os_mbx_wait(RXbox,(void**) &RXptr,INDEF_TIME);   //wait for message
                /* DO SOMETHING */
  _free_box(RXpool,RXptr);         //clear message pointer
  }
}
```

Listato 2.9: ST_ELAB_tsk: raccolta ed elaborazione dati

Come detto l'elaborazione dei dati non è obbiettivo di questo lavoro, per questo nella struttura proposta è presente la sola ricezione dei pacchetti tramite Mailbox. Di seguito è riportato un diagramma che illustra l'idea di funzionamento.

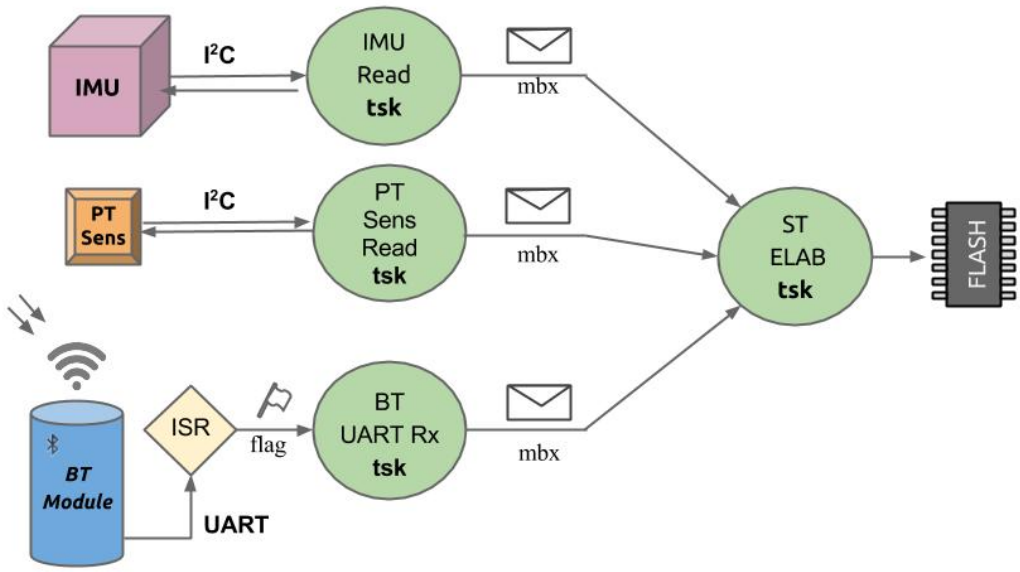


Figura 2.3: Raccolta Dati

2.3.4 Lettura Sensori: IMU e PTS

Vediamo ora i task che interessano la lettura della IMU e del Sensore di Pressione e Temperatura (PTS). Queste istruzioni sono necessarie per la raccolta dei dati che provengono dai dispositivi montati sul Nodo Centrale stesso. L'operazione di campionamento consiste sempre di una lettura via bus I^2C. In particolare sono stati realizzati due task, dalla struttura simile, che agiscono sullo stesso bus ma

indirizzando le due diverse periferiche: IMU e PTS. Vediamo per primo il task di lettura per la IMU.

```
__task void IMU_Read_tsk(void){

    int8_t ACC[6] = {0};
    int8_t GYRO[6] = {0};
    int8_t CMPS[6] = {0};

    int16_t ACC_STACK[3][STACK_SIZE] = {0};
    IM ACC_FILT;
    int16_t GYRO_STACK[3][STACK_SIZE] = {0};
    IM GYRO_FILT;
    int16_t CMPS_STACK[3][STACK_SIZE] = {0};
    IM CMPS_FILT;

    RXmsg *RXptr;
    U8 k = 0;

    //periodic execution
    os_itv_set(IMU_SAMP_RATE);

    for(;;){
      os_itv_wait();

      os_mut_wait(I2C1_MUTX,INDEF_TIME);

      for(k = (STACK_SIZE - 1) ; k > 0   ; k--){
        ACC_STACK[0][k] = ACC_STACK[0][k-1];
        ACC_STACK[1][k] = ACC_STACK[1][k-1]; // stack shift FIFO
        ACC_STACK[2][k] = ACC_STACK[2][k-1];
      }

      //trigger Compass
        I2C_ByteWrite(MPU9150_COMPASS_I2C_ADDRESS, 0x01, 0x0A,I2C1);
        I2C_BufferRead(MPU9150_COMPASS_I2C_ADDRESS,CMPS,
  MPU9150_CMPS_XOUT_L,6,I2C1); //Compass
        I2C_BufferRead(MPU9150_MPU_I2C_ADDRESS,ACC,
  MPU9150_ACCEL_XOUT_H,6,I2C1);    // Accel
        I2C_BufferRead(MPU9150_MPU_I2C_ADDRESS,GYRO,
  MPU9150_GYRO_XOUT_H,6,I2C1);    // Gyro

        os_mut_release(I2C1_MUTX);

      // change endianess and save in stack
      ACC_STACK[0][0] = (ACC[0]<<8|ACC[1]);
      ACC_STACK[1][0] = (ACC[2]<<8|ACC[3]);
```

```
        ACC_STACK[2][0] = (ACC[4]<<8|ACC[5]);

        GYRO_STACK[0][0] = (GYRO[0]<<8|GYRO[1]);
        GYRO_STACK[1][0] = (GYRO[2]<<8|GYRO[3]);
        GYRO_STACK[2][0] = (GYRO[4]<<8|GYRO[5]);

        CMPS_STACK[0][0] = (CMPS[1]<<8|CMPS[0]);
        CMPS_STACK[1][0] = (CMPS[3]<<8|CMPS[2]);
        CMPS_STACK[2][0] = (CMPS[5]<<8|CMPS[4]);

        // running average, example on acceleration
        ACC_FILT.x = ACC_FILT.x +((ACC_STACK[0][0] - ACC_STACK[0][
    STACK_SIZE-1])>>MEAN_FRACTION);
        ACC_FILT.y = ACC_FILT.y +((ACC_STACK[1][0] - ACC_STACK[1][
    STACK_SIZE-1])>>MEAN_FRACTION);
        ACC_FILT.z = ACC_FILT.z +(ACC_STACK[2][0] - ACC_STACK[2][
    STACK_SIZE-1])>>MEAN_FRACTION);

        GYRO_FILT.x = GYRO_STACK[0][0];
        GYRO_FILT.y = GYRO_STACK[1][0];
        GYRO_FILT.z = GYRO_STACK[2][0];

        CMPS_FILT.x = CMPS_STACK[0][0];
        CMPS_FILT.y = CMPS_STACK[1][0];
        CMPS_FILT.z = CMPS_STACK[2][0];

          RXptr = _alloc_box(RXpool);

          RXptr->ID = Master;
          RXptr->acc  = ACC_FILT;
          RXptr->gyro = GYRO_FILT;
          RXptr->cmps = CMPS_FILT;

        os_mbx_send(RXbox,RXptr,TIMEOUT_MBX);

      os_tsk_pass();
      }
}
```

Listato 2.10: IMU_Read_tsk: lettura sensori inerziali

Questo task esegue, ad ogni esecuzione, la lettura delle grandezze attuali derivanti da **Accelerometro**, **Giroscopio** e **Magnetometro**. Ognuno di questi tre fornisce tre separate misure, una per ogni direzione nello spazio (di una terna idealmente ortogonale). Ad ogni chiamata viene utilizzata la funzione I2C_BufferRead vista in precedenza. In particolare, come visto precedentemente, la singola misura è costi-

tuita da 16 bit ed è espressa in complemento a due. Poichè i registri interni della IMU sono ordinati, secondo quanto riportato nell'analisi dell'hardware, la lettura di ogni grandezza consta di una singola transazione di 6 Bytes adiacenti, a partire dall'indirizzo del primo Byte del primo asse di ciascuna grandezza.
In particolare è da ricordare che, a differenza dell'accelerometro e del giroscopio, nel caso del magnetometro la comunicazione avviene verso un diverso indirizzo I^2C, per questioni già descritte nella struttura e inizializzazione della MPU-9150 [§1.3.1]. Inoltre sempre nel caso del solo Magnetometro, ogni lettura dei nuovi dati deve essere preceduta da una scrittura del registro di controllo (0x0A) nel bit denominato Single Measurement Mode.
Per garantire una frequenza di campionamento fissa, il task è eseguito ad alta priorità, periodicamente, secondo le apposite funzioni *os_itv_set(IMU_SAMP_RATE)* e *os_itv_wait()*. In particolare IMU_SAMP_RATE è il numero di tick che lo scheduler fa trascorrere tra due chiamate successive del task.
Da notare, inoltre, l'ultilizzo del Mutex sull'uso del bus I^2C tramite l'istruzione *os_mut_wait(I2C1_MUTX,INDEF_TIME)* che mette il task in attesa nel caso in cui una transizione sul bus sia già in corso da parte di altri task.
A titolo di esempio, in quanto non oggetto di questo testo, in questo task è stato eseguito un possibile filtraggio. Come filtro è stata scelta una semplice Media Mobile (Simple Moving Average) a 16 misure sui soli dati di Accelerazione. La forma matematica di questo filtro è la seguente:

$$m_{n+1} = m_n + \frac{A_{n+k+1} - A_n}{k} \qquad con \qquad m_n = \sum_{t=n}^{n+k} A_t$$

Dove m_{n+1} è la media corrente che, in ogni istante, è funzione della media all'istante precedente m_n, della misura corrente A_{n+k+1} e della misura meno recente A_n, dove k è il numero di misure considerate.
Concludendo per quanto riguarda questo processo, si nota dalle ultime righe del codice come, in linea con quanto detto riguardo alla registrazione dei dati [§2.3.3], vengano passate le letture del sensore in un apposito puntatore **RXptr**, che viene poi inviato tramite sistema Mailbox dalla istruzione **os_mbx_send()**.

Vediamo ora il secondo task dedicato alla lettura dei sensori sul Nodo Centrale. La struttura di questo task è simile a quella del primo. Viene riportato di seguito il codice:

```
__task void PTSens_Read_tsk(void){

    U8 P_DATA[3] = {0};
    U8 T_DATA[3] = {0};
    U32 PRESS,D2 = 0;
    U32 TEMP,D1  = 0;

    RXmsg *RXptr;

    //periodic execution
    os_itv_set(PTSENS_SAMP_RATE);

    for(;;){
      os_itv_wait();

      //-----------PRESS Read-----------------
      os_mut_wait(I2C1_MUTX,INDEF_TIME);
        I2C_CmdSend(MS5611_I2C_ADDRESS, MS5611_CONVERTD1_OSR_4096,
   I2C1);  //trigger Press Cmd
      os_mut_release(I2C1_MUTX);

        OS_SLEEP_MS (MS5611_CONV_DELAY);              //wait for
   conversion to complete

      os_mut_wait(I2C1_MUTX,INDEF_TIME);
        I2C_BufferRead(MS5611_I2C_ADDRESS,P_DATA, MS5611_ADC_READ,3,
   I2C1);
        //read Pressure
      os_mut_release(I2C1_MUTX);

      //-----------TEMP Read -----------------
        I2C_CmdSend(MS5611_I2C_ADDRESS, MS5611_CONVERTD2_OSR_4096,
   I2C1);  //trigger Temp Cmd

      os_mut_release(I2C1_MUTX);

        OS_SLEEP_MS (MS5611_CONV_DELAY); //wait for conv to complete

      os_mut_wait(I2C1_MUTX,INDEF_TIME);
        I2C_BufferRead(MS5611_I2C_ADDRESS,T_DATA, MS5611_ADC_READ,3,
   I2C1);  //read Temperature
```

```
        os_mut_release(I2C1_MUTX);

        // merge data bytes
        D2 = (P_DATA[0]<<16)|(P_DATA[1]<<8)|(P_DATA[2]);
        D1 = (T_DATA[0]<<16)|(T_DATA[1]<<8)|(T_DATA[2]);

        PT_Convert(D1,D2,MS5611_COEFF,&TEMP,&PRESS);

      RXptr = _alloc_box(RXpool);

        RXptr->ID    = Master | 0xC0;
        RXptr->temp  = TEMP;
        RXptr->press = PRESS;

      os_mbx_send(RXbox,RXptr,TIMEOUT_MBX);

    os_tsk_pass();
      }
}
```

Listato 2.11: PTSens_Read_tsk: lettura sensore pressione e temperatura

Come nel caso precedente, questo task è soggetto ad una chiamata periodica, la cui frequenza è definita dal valore di *PTSENS_SAMP_RATE*. Le differenze rispetto alla lettura della IMU appena descritta si notano della procedura di lettura, caratteristica per il sensore, che richiede in primo luogo un comando per iniziare la conversione, dunque un tempo di attesa ed infine un comando di lettura del valore richiesto. Queste operazioni rispecchiano la procedura di lettura per il PTS, descritta nel primo capitolo [§1.3.2]. Dopo le letture, avviene la conversione dei valori (a 24 bit) letti tramite la apposita **PT_Convert**, riportata nella descrizione dell' Hardware [§1.1.3]. Da sottolineare l'utilizzo dello stesso Mutex del quale il task precedente fa già ricorso; questo accorgimento infatti si rende necessario per non creare conflitti nell'occupazione della risorsa bus *I2C1* della MCU.
Come nel caso precedente al termine della raccolta e conversione dei dati, questi vengono scritti nel puntatore **RXptr** utilizzandone i campi della seconda delle possibili configurazioni consentite dal costrutto *union*, come visto precedentemente [§2.3.3].

2.3.5 Ricezione via Bluetooth

Vediamo ora la parte del software riguardante la raccolta dei dati provenienti dai nodi sensori remoti tramite la WBAN basata su Bluetooth Multipoint.

La parte del firmware che gestisce queste operazioni è supportata da un **interrupt handler** e da un **task connesso** a questo. Secondo quanto già riportato rispetto alla gestione degli interrupt in un RTOS [§1.2.3], l'utilizzo di un task di appoggio si rende necessario per non appesantire il carico della ISR (handler) che non deve causare stalli nella rotazione dei task sottostante.
Iniziamo quindi con l'analisi della ISR per la periferica USART6, inizializzata come descritto in precedenza [§2.1.3]. Viene riportato qui il codice dello handler:

```
void ISR_USART6_IRQHandler() {

  if(USART6->SR & USART_SR_RXNE){

    USART_ITConfig(USART6,USART_IT_RXNE,DISABLE); //disable IT
      isr_evt_set(BT_RX_FLAG,tsk5);       //BT_RX_tsk trigger

    USART_ClearITPendingBit(USART6,USART_IT_RXNE);
  }
}
```

Listato 2.12: ISR_USART6_IRQHandler: handler ricezione Bluetooth

La prima operazione eseguita nella procedura di *handling* è un controllo sul tipo di interrupt. Di fatto avviene quindi una sorta di filtraggio al fine di eseguire il codice interno solo nel caso di una ricezione sulla periferica USART6; in particolare all'attivazione della flag *USART_IT_RXNE*.
Per ragioni già affrontate nel precedente capitolo [§1.2.3], all'interno dell'interrupt handler il codice è mantenuto all'essenziale. In particolare viene prima disabilitata la possiblità di ulteriori interrupt[9], quindi viene "alzata" la flag per l'attivazione del task di ricezione ed infine viene fatto il "clear" del *pending bit* dell'interrupt di ricezione.

Vediamo ora il task di appoggio che contiene il codice per servire effettivamente l'interrupt di ricezione. La sua struttura è la seguente:

```
__task void BT_RX_tsk(void){
  RXmsg *RXptr;

  USART_ITConfig(USART6,USART_IT_RXNE,ENABLE);
```

[9]Una volta entrati nella ISR l'interrupt viene disabilitato al fine di non avere chiamate multiple del task di ricezione prima che questa sia "servita". Tutto ciò è solamente a fine precauzionale in quanto il tempo di elaborazione del task di ricezione è comunque molto minore del tempo che intercorre tra due interrupt successivi

```
  Rx_Buffer[0] = 0;
  RxCnt = 0;

  for(;;){
    os_evt_wait_or(BT_RX_FLAG,INDEF_TIME);

        Rx_Buffer[RxCnt] = USART_ReceiveData(USART6);

   if(Rx_Buffer[RxCnt] == ESC_CHAR){

      RXptr = _alloc_box(RXpool);

      RXptr->ID = Rx_Buffer[0] - 0x30;

      if(RXptr->ID & 0xC0){       // IMU data packet
          RXptr->temp  = Rx_Buffer[4]<<24|Rx_Buffer[5]<<16|Rx_Buffer
   [6]<<8|Rx_Buffer[7];
          RXptr->press = Rx_Buffer[8]<<24|Rx_Buffer[9]<<16|Rx_Buffer
   [10]<<8|Rx_Buffer[11];
        }
      else{                       // PTS data packet
          RXptr->acc.x = Rx_Buffer[4]<< 8| Rx_Buffer[5];
          RXptr->acc.y = Rx_Buffer[6]<< 8| Rx_Buffer[7];
          RXptr->acc.z = Rx_Buffer[8]<< 8| Rx_Buffer[9];

          RXptr->gyro.x = Rx_Buffer[10]<< 8| Rx_Buffer[11];
          RXptr->gyro.y = Rx_Buffer[12]<< 8| Rx_Buffer[13];
          RXptr->gyro.z = Rx_Buffer[14]<< 8| Rx_Buffer[15];

          RXptr->cmps.x = Rx_Buffer[16]<< 8| Rx_Buffer[17];
          RXptr->cmps.y = Rx_Buffer[18]<< 8| Rx_Buffer[19];
          RXptr->cmps.z = Rx_Buffer[20]<< 8| Rx_Buffer[21];
        }

      RxCnt = 0;
    }
    else RxCnt++;

    USART_ITConfig(USART6,USART_IT_RXNE,ENABLE);
  }
}
```

Listato 2.13: BT_RX_tsk: Service Routine per interrupt di ricezione Bluetooth

Descriviamo quindi le istruzioni di questo processo: alla attivazione del task vengono eseguite per una sola volta le istruzioni prima del loop *for(; ;)*. Tali istruzioni abilitano l'interrupt di ricezione della USART6 ed inizializzano il primo elemen-

to del buffer di ricezione **Rx_Buffer** ed il relativo contatore. Dopo questo passo il task entra nel loop ed attende (in stato di *WAIT*) l'attivazione della flag **BT_RX_FLAG** gestita dalla ISR appena commentata.
Per meglio descrivere ciò che accade all'arrivo dell'opportuno interrupt, riportiamo la sequenza degli eventi nel seguente elenco puntato:

- Occorre l'interrupt *USART_IT_RXNE* su USART6
- La ISR si auto disabilita ed alza il flag *BT_RX_FLAG* che causa il "trigger" del task si supporto
- Il task, con priorità IRQ, entra in stato *READY* e quindi RUNNING[10]
- Dentro il loop *for(;;)* del task il Byte in arrivo sulla UART, viene letto, controllato e salvato nel buffer *Rx_Buffer*
- Il contatore del buffer viene incrementato e la ISR riabilitata per poter ricevere un nuovo interrupt.
- Il task termina

Questa catena di eventi si ripete fino a che il controllo sul Byte ricevuto non riconosce una coincidenza con un definito carattere di uscita; nel nostro caso un carattere di *newline* "\n". L'arrivo di tale carattere indica per convenzione il termine del pacchetto dati di interesse.
A questo punto viene riempito un puntatore di tipo RXmsg a partire dai dati nel buffer di ricezione. In particolare viene subito letto il campo ID del puntatore al fine di discriminare una scrittura di letture IMU o da una di PTS, come visto in precedenza [2.3.3].
Una volta scritti opportunamente i campi del puntatore, esso viene trasmesso tramite sistema MailBox con le stesse modalità utilizzate per la gestione delle letture dei sensori On-Board; inoltre il contatore del buffer è riazzerato e la ISR riabilitata per consentire la lettura delle ricezioni successive. Riportiamo nella seguente figura un diagramma del funzionamento appena descritto:

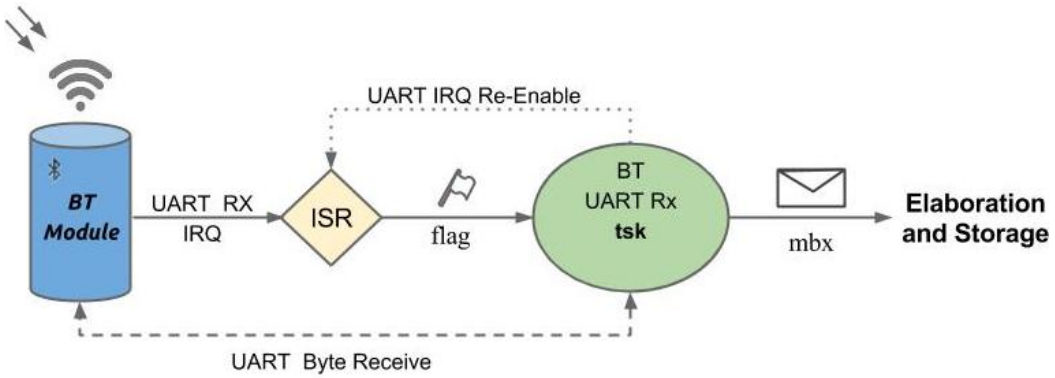


Figura 2.4: Ricezione dai Nodi Sensori Remoti

[10]Il passaggio da READY a RUNNING è praticamente immediato in quanto massima la priorità del task di ricezione (*TSK_PRIO_IRQ*)

In particolare il messaggio Mailbox punta al tasl di elaborazione e salvataggio dati del quale si è data precedentemente una struttura di base [§2.3.3].

2.4 Gestione delle priorità

Vediamo ora i livelli di priorità dei task presenti nel progetto e come lo scheduler di RTX gestisce le code di esecuzione questi processi.

2.4.1 RoundRobin Pre-emptive Scheduling

La politica di *scheduling* scelta per il progetto è quella adottata di default per RTX (e configurabile nel file RTX_CONFIG.c). Questa modalità di utilizzo è definita **Round Robin Pre-emptive Scheduling**. Tramite questa gestione, di base, è attuato quello che si definisce "*Round Robin*" tra i task di background di pari priorità. Nello specifico un modello *Round Robin* consiste in una esecuzione ciclica (a turno) dei task, ai quali è assegnato un dato intervallo di tempo per l'esecuzione. Una volta esaurito tale intervallo, il controllo passa ai task successivi di pari priorità, o , nel caso non ve ne siano, di priorità inferiore (scalando fino eventualmente a giungere all'esecuzione di *os_idle_demon(), un task di sistema definito a minima priorità*. Nell'opzione aggiuntiva di **Pre-emption**, il Round Robin viene interrotto nel caso di ingresso di un task a priorità definita più alta (ad esempio un handler di un interrupt) che prende subito la posizione di Running. Questa politica di scheduling rende quindi necessaria l'eliminazione o la pausa (WAIT) dei task a più alta priorità nei quali si entra, una volta eseguita la loro routine, così da consentire la ripresa dell'esecuzione ciclica di background, interrotta precedentemente. Questi concetti possono essere meglio compresi nel seguente grafico:

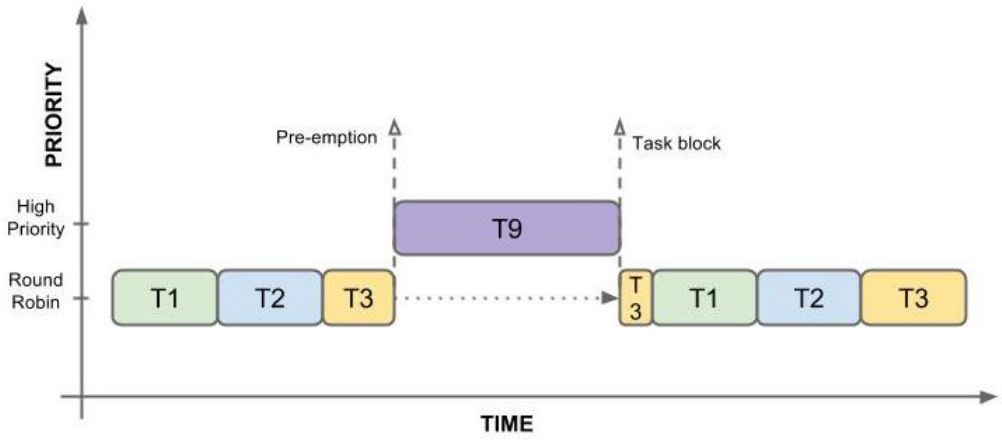


Figura 2.5: Round Robin Pre-emptive Scheduling

Sulle ascisse è posto il tempo (generalmente misurato in *system ticks*), mentre sulle ordinate la priorità. Nel grafico, con colori differenti, sono indicati i task che

passano man mano in esecuzione per mano della gestione a Round Robin.
Il task T9 interrompe l'esecuzione ciclica dei task a bassa priorità, subentra come RUNNING e, dopo aver eseguito la sua routine, cede nuovamente il controllo ai processi meno prioritari. Questo è ad esempio il caso della gestione delle ricezioni dei pacchetti dai nodi sensori, dove T9 rappresenterebbe quindi il task di handling associato alla *Service Routine* dell'interrupt sulla ricezione dell'interfaccia UART.

2.4.2 Livelli di Priorità

Ogni task, al momento della creazione, entra in gestione allo scheduler con una certa priorità. Tale priorità può essere cambiata dinamicamente, durante l'esecuzione della task stessa, oppure da una seconda task. Per questo lavoro sono stati definiti tre[11] livelli di priorità differenti che elenchiamo ora per priorità crescente:

- TASK_PRIO_RR
- TASK_PRIO_HI
- TASK_PRIO_IRQ

Il livello di priorità **TASK_PRIO_RR** viene attribuito alle task di background in esecuzione secondo un Round Robin scheduling. Sotto di essa c'è solo la priorità nulla dell **IDLE_DEMON**, una task che viene eseguita quando nessuna task utente è in esecuzione (spesso mai). Sopra questa priorità troviamo la **TASK_PRIO_HI**, che essendo maggiore, consente l'interruzione del Round Robin pur non essendo la priorità massima delle tre; questo per consentire una gestione degli interrupt tramite un più elevato livello di priorità. L'ultimo livello infatti è costituito dalla **TASK_PRIO_IRQ**, che viene utilizzato solamente per la gestione delle IRQ con una task, di questa priorità, come Handler.

2.4.3 Priorità dei principali Task

Riassumiamo ora nella seguente tabella le condizioni di attesa e l'assegnazione di priorità utilizzata per le task principali utilizzate nel firmware.
Lo schema riportato non è tuttavia fisso. RTOS fornisce delle istruzioni per cambiare la priorità dei vari task sia all'avvio che dinamicamente a seconda dell'occorrenza. In particolare, tuttavia, il modello che si tende a mantenere è quello della attribuzione delle alte priorità ai processi più importanti o non gestibili completamente dal CN, come ad esempio la ricezione dai nodi sensori.

[11] A questi tre in realtà si aggiunge un quarto livello TASK_PRIO_INIT, più elevato tra tutti, che è tuttavia utilizzato solamente per il task di init in quanto questo deve mantenere l'esecuzione fino alla completa inizializzazione di tutte le periferiche e tutti gli altri task

Task	Default Prio	Wait type
Init	PRIO_INIT	x
LED_tsk	PRIO_RR	WAIT_DLY
IMU_Read_tsk	PRIO_HI	WAIT_INT,WAIT_MUT
PTSens_Read_tsk	PRIO_HI	WAIT_INT,WAIT_MUT
BT_SlaveXCmd_tsk	PRIO_HI	WAIT_FLAG
BT_TX_tsk	PRIO_RR	WAIT_MBX,WAIT_MUT
BT_RX_tsk	PRIO_IRQ	WAIT_FLAG

Tabella 2.1: Task Priority

Si conclude qui l'esposizione della realizzazione effettiva del progetto. Nell'appendice è riportato un diagramma complessivo della struttura del firmware e delle interazioni tra i processi e tra questi e le periferiche.

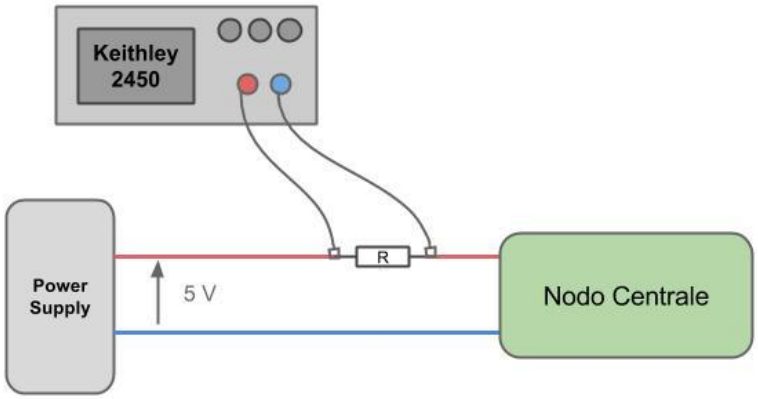
3

Misure e Risultati

In questo ultimo capitolo vengono brevemente analizzati i consumi del Nodo Centrale, secondo varie modalità operative. In particolare l'obbiettivo è quello di stimare gli assorbimenti in corrente per poter aprire a considerazioni su efficienza energetica e tempi di autonomia del dispositivo nel caso di alimentazione a batteria. Inoltre in questi parametri si vuole dare un riferimento di massima per paragoni con future implementazioni di modalità low power, che non sono state analizzate in questo testo.

3.1 Consumi

Per l'analisi dei consumi sono state prese in considerazione alcune modalità operative che verranno descritte in seguito. Le misure sono state ricavate, per motivi pratici, dalla lettura delle cadute di potenziale a cavallo di una piccola resistenza di valore noto, in serie con uno dei terminali di alimentazione. Di conseguenza i consumi si riferiscono al sistema Nodo Centrale nel complesso e non alle sue singole parti. Lo schema del setup di misura è quindi il seguente:



Tutte le misure che seguono sono state effettuate con lo strumento *Keithley 2450*, specifico per misure di precisione di voltaggi, correnti, resistenze e potenze. Per

conoscere con precisione e accuratezza il valore della resistenza, è stata effettuata una misura a 4 fili di questa. I valori di corrente sono quindi ottenuti dividendo le letture di tensione per tale resistenza.
Il nodo centrale durante i test è stato alimentato tramite una presa USB 2.0 standard con tensione di uscita nominale di 5 Volt e misurata effettiva $(4957 \pm 1) * 10^{-3}\ V$. Non essendo l'obbiettivo di questo testo, la misura di precisione dei consumi, i valori che seguono sono da considerarsi solamente una stima di quelli effettivi come già accennato nell'introduzione.
Sono state considerate per il Nodo Centrale come esempio di utilizzo tre differenti modalità:

- IDLE, nessun task in esecuzione, tutte le periferiche attive ed alimentate
- IDLE LED, nessun task in esecuzione, tutte le periferiche attive ed alimentate, feedback visivo tramite accensione di uno dei due led
- FULL, tutti i task standard in esecuzione (campionamento, ricezione, elaborazione dati), feedback visivo tramite accensione alternata dei due led

I risultati ottenuti sono riportati in tabella. Essendo interessati ad un valore rappresentativo delle correnti assorbite, riferite in tutti i casi a regimi in buona approssimazione stazionari, come statistica è stata scelta la media alla quale viene attribuita l'incertezza di 3 volte la deviazione standard del campione.

Regime	Corrente [mA]	Dev. Std. [mA]	Risultato [mA]
IDLE	58,2960	0,1664	58,3 ± 0,5
IDLE LED	62,2243	1,3916	62 ± 4
FULL	65,5880	3,1411	66 ± 9

Tabella 3.1: Assorbimenti di corrente

Come è logico aspettarsi i consumi sono ordinati in modo crescente a seconda dell'utilizzo più o meno intensivo delle periferiche. In pieno regime di utilizzo (FULL) si osserva dalle letture (in particolare dalla deviazione del campione) un assorbimento di corrente meno stabile e più oscillante rispetto agli altri due casi.
Per quanto riguarda la modalità definita IDLE il consumo è inferiore e l'assorbimento di corrente più stabile. Di seguito è riportato il grafico della modalità IDLE per una finestra di osservazione di qualche secondo.

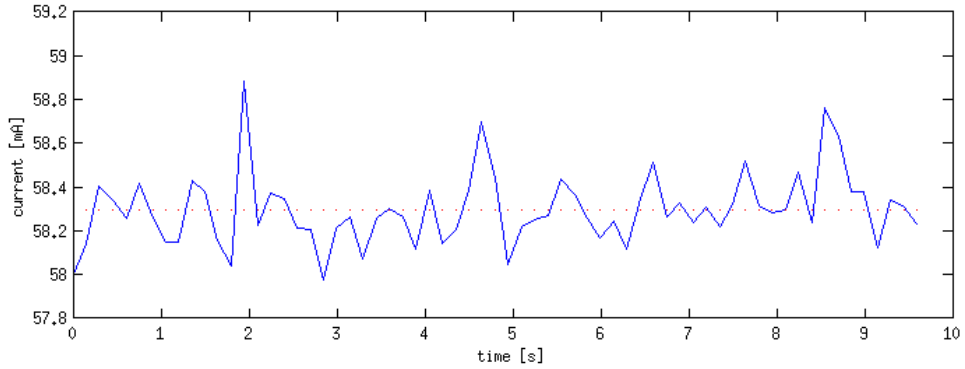


Figura 3.1: Fase IDLE, assorbimento di corrente

L'accensione di un led (IDLE LED) comporta infine un aumento dei consumi di circa 4mA con oscillazioni nell'assorbimento non troppo significative.

In ultima analisi è stato analizzato il profilo di assorbimento di corrente della fase di inizializzazione, in particolare questo include essenzialmente il setup della connessione Multipoint con i nodi sensori e la configurazione dei sensori tramite bus I^2C. Riportiamo l'andamento nella seguente figura:

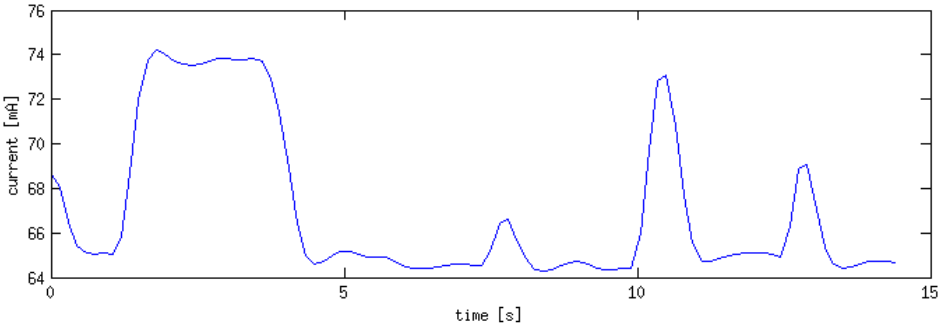


Figura 3.2: Inizializzazione, assorbimento di corrente

Dal grafico si osservano dei consumi più accentuati, fino a circa 73 mA durante la fase di inizializzazione. Questi picchi corrispondono all'utilizzo in trasmissione del modulo Bluetooth del CN. In particolare dall'andamento si distinguono l'alternanza tra l'invio dei comandi di connessione e le attese di risposta da parte dei nodi sensori. Ad avvalorare questa interpretazione, il tempo di inizializzazione osservato sul grafico è inoltre compatibile con quello stimato tramite *breakpoint* software.

Nel complesso, il consumo per il CN, esclusa la fase di inizializzazione, può essere stimato come la media tra gli assorbimenti di corrente in modalità IDLE e FULL. Considerando nel calcolo stabile il voltaggio di alimentazione (Va) si ottiene:

$$P = \frac{I\_Idle + I\_Full}{2} * Va = 307\ mW$$

Questo consumo non è eccessivamente elevato, tuttavia nel firmware realizzato non è stata implementata alcuna politica di risparmio energetico, come modalità low power o sleep mode (disponibili sia per la MCU che per il modulo Bluetooth). Di conseguenza l'efficienza energetica è sicuramente un parametro migliorabile in vista di sviluppi futuri del progetto.
In ultima analisi va specificato che per problematiche relative all'hardware (che risulta già assemblato e non scomponibile) non è stato possibile approfondire l'indagine verso il consumo effettivo dei singoli componenti, in particolare MCU e modulo Bluetooth. Di conseguenza non è semplice stabilire quale sia il consumo effettivo dei singoli e non tramite approssimazioni e considerando valido per ogni misura il principio di sovrapposizione per i consumi.

3.2 Acquisizioni, Prestazioni e Trasmissione

Vediamo ora alcune osservazioni riguardanti la comunicazione Bluetooth, l'acquisizione dei dati dai sensori e il carico computazionale a cui è sottoposta la MCU del Nodo Centrale.
Per quanto riguarda la trasmissione è stata simulata con successo la ricezione e trasmissione di pacchetti tramite l'utilizzo di due USB dongle Bluetooth con lo stesso modulo radio del CN. Tuttavia non è stato ancora testato il sistema nel complesso e quindi la qualità della comunicazione con i nodi sensori veri e propri, il cui utilizzo non rientrava negli obbiettivi di questo testo.
Sono state effettuate tuttavia altre misure, tra le quali il tempo necessario per stabilire la connessione nella WBAN tra Nodo Centrale e nodi sensori. Approssimativamente si è trovato che una soglia sicura sui tempi di attesa per la risposta dei nodi porta complessivamente ad un'attesa di circa **10-11 secondi** per stabilire la connessione Multipoint. Questo tempo potrebbe essere ridotto introducendo al posto dei tempi di attesa un più complesso sistema di acknowledge sulle risposte ai comandi di setup. Con questo sistema sarebbe possibile inoltre una gestione più dinamica in caso di errori di connessione. Da stime eseguite sui tempi veri di risposta dei nodi sensori si è trovato che il valore potrebbe effettivamente essere abbassato fino a circa 6 secondi mantenendo lo stesso un buon margine di sicurezza sul tentativo.
Per quanto riguarda le acquisizioni invece, il sistema è stato testato con successo rispetto a letture di IMU (Accelerometro, Giroscopio, Magnetometro) e PTS (Barometro, Termometro) a **50** $\frac{sample}{s}$ ciascuno. Questo non è ancora il limite superiore raggiungibile tuttavia occorre tener conto della effettiva necessità di campionamenti più frequenti; in particolare considerando che i nodi sensori dovranno inviare le loro acquisizioni con la stessa densità. Per quanto riguarda la bontà dei valori

acquisiti, non è oggetto di questo testo trattare l'effettiva precisione e accuratezza dei valori letti dai sensori sul CN, tuttavia è bene tener presente (in termini di carico computazionale) che sarà necessaria l'implementazione di un efficiente filtraggio (qui solo abbozzato a titolo di esempio sulle accelerazioni) sui valori e di una fase iniziale di calibrazione a seconda della precisione che si intende ottenere.

Tramite appositi strumenti software è stata inoltre analizzata l'occupazione del **task stack** per ciascun task. In particolare questo fattore è collegato alla occupazione totale di RAM del microprocessore durante l'esecuzione. Di default ad ogni task eseguito è stato associato uno stack della dimensione di 1024 Byte. Nella seguente tabella sono riportati i risultati dei test sull'occupazione media dei vari task stack.

Task	Stack Size [Byte]	Stack Load [% of Stack Size]
BT_TX_tsk	1024	7
BT_RX_tsk	1024	9
LED_tsk	1024	6
IMU_Read_tsk	1024	50
PTS_Read_tsk	1024	18
BT_SlaveX_Cmd_tsk	1024	5

Come si osserva dai valori in tabella l'uso di 1024 Byte come dimensione standard per lo stack è più che sufficiente per i task sviluppati, che al massimo arrivano ad un'occupazione del 50% di questo (caso di lettura della IMU). Secondo quanto riportato nel manuale di utilizzo di RTX, ogni task stack richiede l'allocazione di 52 Byte aggiuntivi per la gestione. Di conseguenza con questa configurazione la RAM preallocata per i task è $(1024 + 52) * 6 = 6456$ *Byte* ovvero circa **6 KByte**. A questo valore vanno aggiunte le allocazioni di memoria dovute alla gestione di RTX e quindi del suo scheduler e dei sistemi come Mailbox, Mutex, etc (secondo i valori riportati in appendice). Considerando anche questi ultimi fattori l'occupazione di RAM stimata in fase di esecuzione[1] va aumentata di circa 700 Byte (valore sovrastimato); nel complesso quindi ammonta a poco più di **7 KByte**. Questo valore è ragionevole se si pensa che quella sviluppata in questo testo è solamente la base del firmware. Compilando il firmware tuttavia si nota un forte aumento della richiesta di RAM, di circa 10 KByte. Questa discrepanza è circoscritta allo spazio richiesto dalle *Zero-Initialized Variables* ovvero dalla maggior parte delle variabili e buffer dichiarate nel programma, delle quali non si era tenuto precedentemente conto nel calcolo dell'occupazione di memoria. Tuttavia la MCU scelta rende di-

[1] La RAM richiesta durante la fase di inizializzazione delle variabili al

sponibili fino a 192 KByte di RAM e si rivela quindi sufficiente per lo scopo.
In vista di sviluppi futuri, l'aggiunta di moduli software porterà all'incremento dello spazio in RAM necessario. Questa tendenza potrebbe rendere indispensabile un' allocazione del task stack più specifica, ovvero riservando ad ogni processo solamente lo spazio realmente necessario alla sua esecuzione.

In ultima analisi, è stato considerato qualitativamente il carico computazionale al quale la MCU è sottoposta. Per fare ciò è stato implementato un semplice contatore software all'interno del task di sistema *os_idle_demon*. Questo processo è eseguito quando nessun altro task è in stato RUNNING e non vi sono task in condizione READY pronti ad essere attivati; per questo l'esecuzione di *os_idle_demon* può essere considerata come un indice di inattività della MCU. Osservando l'evoluzione del contatore, che si incrementa in queste condizioni di inattività, si è osservato come in effetti il microprocessore trascorra porzioni significative di tempo in questo processo. Pur non essendo possibili analisi quantitative di questo comportamento, si può tuttavia affermare che il carico computazionale a cui il processore è sottoposto è lontano dal suo limite superiore.
Infine per quanto riguarda la dimensione del programma vero e proprio (codice + moduli RTOS + costanti), dalla compilazione risulta necessario uno spazio ROM pari a circa **68 KByte** dei 1024, ovvero 1 MByte disponibili nella MCU scelta.

4

Osservazioni e Conclusioni

Dopo aver effettuato il test del sistema sotto differenti profili, possiamo commentare i risultati come segue. Per quanto riguarda l'utilizzo di un RTOS, la scelta si è rivelata vantaggiosa sotto vari aspetti. In primo luogo la gestione a task dei processi pur essendo più complessa dal punto di vista della sincronizzazione, si è rivelata vincente sotto gli aspetti dell'efficienza, della manutenzione del firmware e della modularità e riutilizzabilità delle singole parti. Il carico computazionale aggiuntivo che comporta l'utilizzo di un RTOS non ha gravato in maniera apprezzabile sulle prestazioni, almeno per quanto riguarda il microcontrollore utilizzato. In particolare sono stati effettuati test rispetto all'occupazione di memoria RAM che è risultata inferiore a 6 Kbyte (sui 192 disponibili) per l'intero progetto. La disponibilità di un sistema multitasking inoltre sembra essere la scelta più indicata per applicazioni complesse e ricche di funzionalità, come è questo il caso, rispetto ad una programmazione classica a "Super-Loop" (dove le istruzioni sono eseguite ciclicamente secondo un ordine fisso). In ultimo luogo l'utilizzo di task per buona parte indipendenti tra loro fornisce vantaggi anche dal punto di vista energetico e del carico computazionale, consentendo di attivare e disattivare processi in un'ottica di gestione dinamica delle varie componenti software del firmware.

Per quanto riguarda l'utilizzo di una connessione Multipoint Bluetooth, invece, possiamo concludere quanto segue: sicuramente utilizzando un protocollo Bluetooth la modalità Multipoint risulta essere l'unica opzione valida per la condivisione real-time di dati tra un numero di dispositivi superiore a due. Come visto infatti è decisamente inattuabile l'impiego di una connessione Point-to-Point alternata tra un Master e due nodi remoti; questo per i tempi di connessione decisamente elevati (secondi) rispetto alla comunicazione dei dati (millisecondi). L'utilizzo della modalità Multipoint è quindi l'unica soluzione per la creazione della WBAN per lo scambio di dati real-time. Non è stato possibile testare la connessione con i nodi sensori effettivi in condizioni di trasmissione e ricezioni intensive, tuttavia in base

a test di scambio di singoli pacchetti dati tramite dongle USB Bluetooth con modulo radio identico al Nodo Centrale, si è verificato l'effettivo funzionamento del firmware rispetto agli obbiettivi prefissati. Inoltre la scelta di scrivere procedure dove i BDaddress dei nodi erano già noti non è casuale in quanto, in questo modo, non è richiesto una scansione iniziale per la ricerca degli slave e questo comporta un abbassamento dei tempi di inizializzazione del dispositivo come analizzato nel terzo capitolo. Eventualmente quindi una scelta logica potrebbe essere quella di mantenere le procedure di connessione esposte nel testo ed affiancare a queste una terza procedura da eseguire solamente al primo "*binding*" dei dispositivi; al fine di scansionare e salvare i BDaddress degli slave solamente una volta.

Per quanto riguarda i consumi, dalle misure di assorbimento in corrente effettuate si è stimato un consumo di potenza medio di circa 300 mW con picchi di 370 mW in fase di inizializzazione o uso intensivo della trasmissione Bluetooth. Inoltre la potenza di trasmissione del modulo Bluetooth è eccessiva rispetto alla effettiva necessità e questo provoca consumi aggiuntivi in trasmissione. Da specificare tuttavia il fatto che non sono state implementate routine di risparmio energetico o sleep mode e che quindi questo valore risulta comunque indicativo e sovrastimato rispetto a quello che sarà il prodotto finale.
Con questo si conclude il lavoro esposto nel testo. Quello portato a termine è la base per il futuro sviluppo del progetto. La speranza è che siano stati dimostrati l'affidabilità e potenzialità del sistema hardware e i vantaggi dell'applicazione di un sistema operativo real-time per il firmware. I moduli software da sviluppare in futuro saranno dedicati alla gestione delle periferiche di feedback, in particolare di tipo uditivo e alla implementazione delle routine di analisi dati per il riconoscimento delle varie condizioni cliniche del paziente a partire dalle letture dei sensori. Inoltre guardando sempre ad obbiettivi a lungo termine, sarà utile lo sviluppo di routine software per consentire l'interfacciamento USB del dispositivo con altri dispositivi per il salvataggio e l'analisi dei dati raccolti. Da ricordare inoltre che uno dei principali obbiettivi del progetto CuPiD è quello di fornire un servizio personalizzato per il paziente, di conseguenza sarà necessaria un ampia fase di test per consensitre la calibrazione del sistema e il corretto sviluppo degli algoritmi di feedback adattati a casi reali di utilizzo.
CuPiD si pone quindi come un progetto impegnativo a lungo termine che possa fornire uno strumento utile per la terapia di patologie come il morbo di Parkinson e non solo, nell'ottica di poter migliorare le condizioni di vita dei pazienti.

Appendice

4.1 Registri interni di interesse MPU9150 e MS5611

IMU MPU9150

Register	Address	Type
MPU9150_MPU_I2C_ADDRESS7	0x68	R
MPU9150_SELF_TEST_X	0x0D	R/W
MPU9150_SELF_TEST_Y	0x0E	R/W
MPU9150_SELF_TEST_Z	0x0F	R/W
MPU9150_SELF_TEST_A	0x10	R/W
MPU9150_SMPLRT_DIV	0x19	R/W
MPU9150_CONFIG	0x1A	R/W
MPU9150_GYRO_CONFIG	0x1B	R/W
MPU9150_ACCEL_CONFIG	0x1C	R/W
MPU9150_MOT_DUR	0x20	R/W
MPU9150_FIFO_EN	0x23	R/W
MPU9150_I2C_MST_CTRL	0x24	R/W
MPU9150_I2C_MST_STATUS	0x36	R
MPU9150_INT_PIN_CFG	0x37	R/W
MPU9150_INT_ENABLE	0x38	R/W
MPU9150_INT_STATUS	0x3A	R
MPU9150_ACCEL_XOUT_H	0x3B	R
MPU9150_ACCEL_XOUT_L	0x3C	R
MPU9150_ACCEL_YOUT_H	0x3D	R
MPU9150_ACCEL_YOUT_L	0x3E	R
MPU9150_ACCEL_ZOUT_H	0x3F	R
MPU9150_ACCEL_ZOUT_L	0x40	R
MPU9150_TEMP_OUT_H	0x41	R
MPU9150_TEMP_OUT_L	0x42	R
MPU9150_GYRO_XOUT_H	0x43	R
MPU9150_GYRO_XOUT_L	0x44	R
MPU9150_GYRO_YOUT_H	0x45	R
MPU9150_GYRO_YOUT_L	0x46	R
MPU9150_GYRO_ZOUT_H	0x47	R
MPU9150_GYRO_ZOUT_L	0x48	R
MPU9150_MOT_DETECT_STATUS	0x61	R
MPU9150_I2C_MST_DELAY_CTRL	0x67	R/W
MPU9150_SIGNAL_PATH_RESET	0x68	R/W
MPU9150_MOT_DETECT_CTRL	0x69	R/W
MPU9150_USER_CTRL	0x6A	R/W
MPU9150_PWR_MGMT_1	0x6B	R/W
MPU9150_PWR_MGMT_2	0x6C	R/W
MPU9150_WHO_AM_I	0x75	R
MPU9150_CMPS_I2C_ADDRESS7	0x0C	R
MPU9150_CMPS_XOUT_L	0x03	R
MPU9150_CMPS_XOUT_H	0x04	R
MPU9150_CMPS_YOUT_L	0x05	R
MPU9150_CMPS_YOUT_H	0x06	R
MPU9150_CMPS_ZOUT_L	0x07	R
MPU9150_CMPS_ZOUT_L	0x08	R
MPU9150_CMPS_CTRL	0x0A	R/W

PTS MS5611

Register	Address	Type
MS5611_I2C_ADDRESS	0xEE	R
MS5611_RESET	0x1E	R/W
MS5611_CONVERTD1_OSR_256	0x40	W
MS5611_CONVERTD1_OSR_512	0x42	W
MS5611_CONVERTD1_OSR_1024	0x44	W
MS5611_CONVERTD1_OSR_2048	0x46	W
MS5611_CONVERTD1_OSR_4096	0x48	W
MS5611_CONVERTD2_OSR_256	0x50	W
MS5611_CONVERTD2_OSR_512	0x52	W
MS5611_CONVERTD2_OSR_1024	0x54	W
MS5611_CONVERTD2_OSR_2048	0x56	W
MS5611_CONVERTD2_OSR_4096	0x58	W
MS5611_ADC_READ	0x00	R
MS5611_PROM0_READ	0xA0	R
MS5611_PROM1_READ	0xA2	R
MS5611_PROM2_READ	0xA4	R
MS5611_PROM3_READ	0xA6	R
MS5611_PROM4_READ	0xA8	R
MS5611_PROM5_READ	0xAA	R
MS5611_PROM6_READ	0xAC	R
MS5611_PROM7_READ	0xAE	R

4.2 Struttura funzionale del Firmware

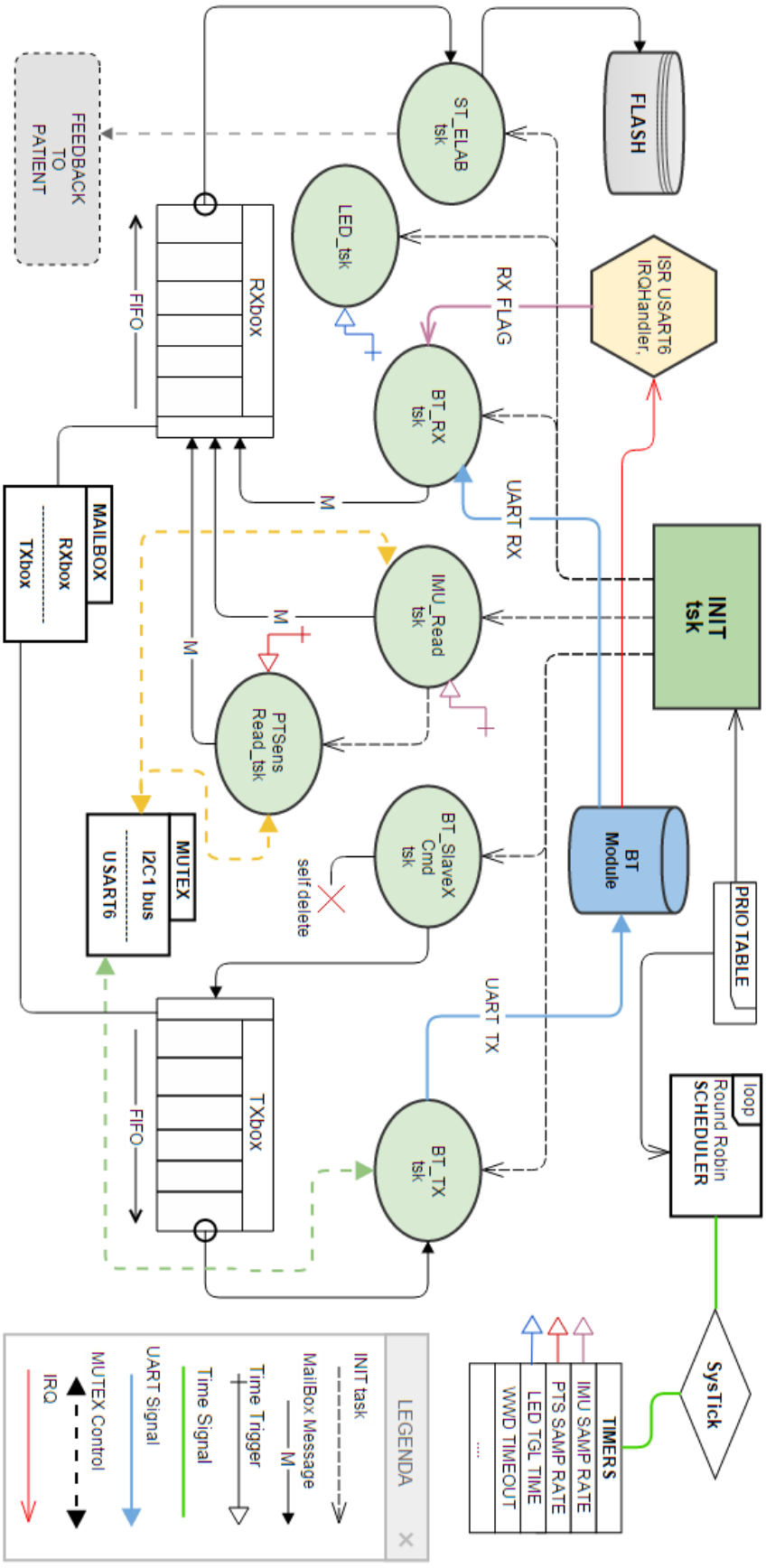


Figura 4.1: Struttura Funzionale del Firmware

4.3 Requisiti Hardware per RTX

CODE Size	< 4.0 KBytes
RAM Space for Kernel	< 300 Bytes + 128 Bytes User Stack
RAM Space for a Task	TaskStackSize + 52 Bytes
RAM Space for a Mailbox	MaxMessages*4 + 16 Bytes
RAM Space for a Semaphore	8 Bytes
RAM Space for a Mutex	12 Bytes
RAM Space for a User Timer	8 Bytes
Hardware Requirements	SysTick timer

Figura 4.2: Occupazione di memoria per RTX

4.4 Listati secondari

```
void USART_TX(USART_TypeDef* USARTx, volatile char *c){

  while(*c){
    // aspetto che il buffer di invio sia vuoto
  while( !(USARTx->SR & 0x00000040) );
    USART_SendData(USARTx, *c);
    *c++;
  }
}
```

Listato 4.1: USART_TX: trasmissione con interfaccia USART

```
void BT_Init_PTP_Connection(const char* BDAddr){
    char temp[RX_BUFF_SIZE];

    strcpy(temp,AT_SPP_CONN);
    strcat(temp,BDAddr);
    strcat(temp,CRLF);

    USART_TX(USART6,AT_PTP_MODE);
    OS_SLEEP_MS(WAIT_SHORT);

    USART_TX(USART6,"AT+AB Config Var05 = 0\r\n"); //Point To Point
    OS_SLEEP_MS(WAIT_SHORT);

    USART_TX(USART6,temp);

    OS_SLEEP_MS(WAIT_LONG);
}
```

Listato 4.2: BT_Init_PTP_Connection: connessione Point to Point

```
void FormatString(char* dest,const char* cmd,U8 SlaveID){
          U8 dim;

          dim = strlen((char*)cmd);
          sprintf(dest,"%u%u%u%u",SlaveID,(dim/100),((dim/10)- (dim
    /100)*10),(dim%10));

          strcat(dest,(char*)cmd);
}
```

Listato 4.3: FormatString: aggiunta header ai pacchetti Multipoint

```
I2CStatus I2C_ByteWrite(u8 slAddr, u8 pBuffer, u8 WriteAddr,
    I2C_TypeDef* I2C_Periph)
{
  volatile uint16_t Timeout = 0xFFFF;
  /* Send START condition */
  I2C_GenerateSTART(I2C_Periph, ENABLE);

  /* Test on EV5 and clear it */
  while(!I2C_CheckEvent(I2C_Periph, I2C_EVENT_MASTER_MODE_SELECT))
    {
      if (Timeout-- == 0)
      return Error;
    }

  /* Send address for write */
  I2C_Send7bitAddress(I2C_Periph, slAddr, I2C_Direction_Transmitter);

  Timeout = 0xFFFF;
  /* Test on EV6 and clear it */
  while(!I2C_CheckEvent(I2C_Periph,
   I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED))
    {
      if (Timeout-- == 0)
      return Error;
    }

  /* Send the internal address to write to */
  I2C_SendData(I2C_Periph, WriteAddr);

  Timeout = 0xFFFF;
  /* Test on EV8 and clear it */
  while(!I2C_CheckEvent(I2C_Periph, I2C_EVENT_MASTER_BYTE_TRANSMITTED
   ))
    {
```

```
      if (Timeout-- == 0)
      return Error;
    }

  /* Send the byte to be written */
  I2C_SendData(I2C_Periph, pBuffer);

  Timeout = 0xFFFF;
  /* Test on EV8 and clear it */
  while(!I2C_CheckEvent(I2C_Periph, I2C_EVENT_MASTER_BYTE_TRANSMITTED
   ))
    {
      if (Timeout-- == 0)
      return Error;
    }

  /* Send STOP condition */
  I2C_GenerateSTOP(I2C_Periph, ENABLE);

  Timeout = 0xFFFF;
    while(Timeout--);

  return Success;
}
```

Listato 4.4: I2C_Byte_Write: scrittura I^2C

```
I2CStatus I2C_BufferRead(u8 slAddr, u8* pBuffer, u8 ReadAddr, u16
    NumByteToRead, I2C_TypeDef* I2C_Periph)
{
  volatile uint16_t Timeout = 0xFFFF;

  /* While the bus is busy */
  while(I2C_GetFlagStatus(I2C_Periph, I2C_FLAG_BUSY))
    {
      if (Timeout-- == 0)
      return Error
      ;
    }

  /* Send START condition */
  I2C_GenerateSTART(I2C_Periph, ENABLE);

    Timeout = 0xFFFF;
  /* Test on EV5 and clear it */
  while(!I2C_CheckEvent(I2C_Periph, I2C_EVENT_MASTER_MODE_SELECT))
    {
```

```
    if (Timeout-- == 0)
    return Error;
  }

/* Send address for write */
I2C_Send7bitAddress(I2C_Periph, slAddr, I2C_Direction_Transmitter);

  Timeout = 0xFFFF;
/* Test on EV6 and clear it */
while(!I2C_CheckEvent(I2C_Periph,
 I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED))
  {
    if (Timeout-- == 0)
    return Error;
  }

/* Clear EV6 by setting again the PE bit */
I2C_Cmd(I2C_Periph, ENABLE);

/* Send the internal address to write to */
I2C_SendData(I2C_Periph, ReadAddr);

  Timeout = 0xFFFF;
/* Test on EV8 and clear it */
while(!I2C_CheckEvent(I2C_Periph, I2C_EVENT_MASTER_BYTE_TRANSMITTED
 ))
  {
    if (Timeout-- == 0)
    return Error;
  }

/* Send STRAT condition a second time */
I2C_GenerateSTART(I2C_Periph, ENABLE);

  Timeout = 0xFFFF;
/* Test on EV5 and clear it */
while(!I2C_CheckEvent(I2C_Periph, I2C_EVENT_MASTER_MODE_SELECT))
  {
    if (Timeout-- == 0)
    return Error;
  }

/* Send address for read */
I2C_Send7bitAddress(I2C_Periph, slAddr, I2C_Direction_Receiver);

  Timeout = 0xFFFF;
/* Test on EV6 and clear it */
```

```
  while(!I2C_CheckEvent(I2C_Periph,
   I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED))
    {
      if (Timeout-- == 0)
      return Error;
    }
    Timeout = 0xFFFF;
  /* While there is data to be read */
  while(NumByteToRead)
  {
    if (Timeout-- == 0)
    return Error;
    if(NumByteToRead == 1)
    {
      /* Disable Acknowledgement */
      I2C_AcknowledgeConfig(I2C_Periph, DISABLE);

      /* Send STOP Condition */
      I2C_GenerateSTOP(I2C_Periph, ENABLE);
    }

    /* Test on EV7 and clear it */
    if(I2C_CheckEvent(I2C_Periph, I2C_EVENT_MASTER_BYTE_RECEIVED))
    {
      /* Read a byte from the LIS3LV02DL */
      *pBuffer = I2C_ReceiveData(I2C_Periph);

      /* Point to the next location where the byte read will be saved
    */
      pBuffer++;

      /* Decrement the read bytes counter */
      NumByteToRead--;
    }
  }

  /* Enable Acknowledgement to be ready for another reception */
  I2C_AcknowledgeConfig(I2C_Periph, ENABLE);

  return Success;
}
```

Listato 4.5: I2C_BufferRead: lettura I^2C

```
I2CStatus I2C_CmdSend(u8 slAddr, u8 cmd, I2C_TypeDef* I2C_Periph){

  volatile uint16_t Timeout = 0xFFFF;

  /* Send START condition */
  I2C_GenerateSTART(I2C_Periph, ENABLE);

  /* Test on EV5 and clear it */
  while(!I2C_CheckEvent(I2C_Periph, I2C_EVENT_MASTER_MODE_SELECT))
    {
      if (Timeout-- == 0)
      return Error;
    }

  /* Send address for write */
  I2C_Send7bitAddress(I2C_Periph, slAddr, I2C_Direction_Transmitter);

  Timeout = 0xFFFF;
  /* Test on EV6 and clear it */
  while(!I2C_CheckEvent(I2C_Periph,
   I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED))
    {
      if (Timeout-- == 0)
      return Error;
    }

  /* Send the internal address to write to */
  I2C_SendData(I2C_Periph, cmd);

  Timeout = 0xFFFF;
  /* Test on EV8 and clear it */
  while(!I2C_CheckEvent(I2C_Periph, I2C_EVENT_MASTER_BYTE_TRANSMITTED
   ))
    {
      if (Timeout-- == 0)
      return Error;
    }

      /* Send STOP condition */
  I2C_GenerateSTOP(I2C_Periph, ENABLE);

  return Success;

}
```

Listato 4.6: I2C_CmdSend: indirizzamento I^2C

Bibliografia

[1] Building Applications with RL-ARM, ARM Ltd and ARM GmbH, 2009

[2] http://www.arm.com/products/tools/software-tools/mdk-arm/middleware-libraries/rtx-real-time-operating-system.php,
ARM Ltd, 2014

[3] Using the firmware of the AT command set, STMicroelectronics, 2013

Ringraziamenti

Le persone che vorrei ringraziare per il traguardo raggiunto probabilmente occuperebbero più spazio della tesi stessa se ne scrivessi il nome e la ragione. Tuttavia, come si conviene, un piccolo promemoria spero possa fare le veci.

Ringrazio il mio relatore Prof. Luca Benini per l'opportunità che mi è stata data di unirmi a questo progetto interessante. Inoltre ringrazio per l'aiuto il mio correlatore Dott. Filippo Casamassima e il Dott. Simone Benatti e la Dott.ssa Elisabetta Farella per la disponibilità e gli utili consigli.

Una Laurea tuttavia è un percorso del quale questa tesi rappresenta solamente l'ultimo gradino.

Per aver potuto percorrere tutta la scala fino alla cima:

Grazie soprattutto ai miei genitori, a Roberta e Stefano, Francesco e Marco.

Grazie a Ilaria, Angio, Irene, Fabio, Michelangelo, Daniele, Mattei, Gimmy, Tommy, Guenda, Ari, Mati, Michele, Puffo, Ceci, al Lama, a entrambi i Pasqui, a Giulio, Pippo, Gavio, Laurina, alla Gas, Carolina, Magda, Beatrice, Mary, Pate, Marty, Sere, Pelo, la Vale, Nic, Rocco e tutti quelli che "non sono del mestiere" ma mi hanno sempre appoggiato in questo percorso.
Grazie a tutti quelli che invece "sono del mestiere" e hanno condiviso questi anni con me: Grazie ad Alberto, Armando, Pascucci, Bruno, (DJ)Seiss, l'Otty, la Laura, grazie a Michi, ed Andre.
Grazie a tutti i ragazzi dell' Unibo Motorsport per quello che mi hanno trasmesso in questi anni.
Grazie a tutti quelli che hanno condiviso le mie passioni, mi hanno aiutato ad andare avanti e spinto a dare il meglio. Per tutti voi

```
while(1) printf("Grazie!\n");
```