Università degli Studi di Padova

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Realizzazione di Transducer Interface Module (IEEE 1451) su sistemi a microcontrollore

Relatore: Prof.ssa Giada Giorgi

Laureando: Salvatore Brundo

24 Ottobre 2011

Questo documento é stato scritto in LaTeX su Debian GNU/Linux.

Indice

Elenco delle figure

Listings

Sommario

In questo lavoro di tesi si mira a realizzare un'implementazione lato TIM dello standard IEEE 1451.0 su di una scheda di fabbricazione Microchip® dotata di un microcontrollore della famiglia PIC18 della stessa casa produttrice. La scheda è dotata inoltre di transceiver radio per la comunicazione mediante protocollo 802.15.4.

Capitolo 1

Overview del problema

In questo capitolo viene offerta una rapida overview che spiega lo scopo della tesi, l'ambiente operativo in cui si colloca, le problematiche principali.

1.1 Ambito operativo

L'ambito operativo in cui si inserisce questo lavoro è quello delle reti embedded orientate alla misura di grandezze. A tal proposito l'IEEE lavora ormai da una decade alla standardizzazione di una realtà, quella delle reti embedded, assai variegata e che presenta sovente problemi di incompatibilità e scarsa interoperabilità.

Reti embedded per la misurazione

Lo scopo dello standard IEEE 1451 è infatti la realizzazione di un modello teorico molto ampio che copra tutti i possibili problemi che una rete di questo genere possa presentare. Gli aspetti coperti dallo standard sono molteplici: dalla definizione di un modello comune per il concetto di nodo, passando poi per le possibili forme di comunicazione intra-nodo senza tralasciare problematiche inerenti alla rappresentazione dei dati raccolti.

Il 1451 introduce pertanto il concetto basilare di *smart sensor*, raffinando la ben più comune definizione di sensore *general purpose*. Tale estensione fornisce al sensore elementare alcune peculiarità che lo rendono assai più potente: esso è infatti fornito di uno strato software di appoggio in grado di processare i dati grezzi acquisiti e renderli poi disponibili in modo standardizzato.

Smart sensor

Altra brillante intuizione dell'IEEE è l'idea dei TEDS, ovvero *Transducer Electronic Data Sheet*. Tale concetto consiste nella abilità dello smart sensor di contenere al proprio interno diversi data sheet codificati elettronicamente in cui sono riportate le informazioni descrittive principali dello stesso, le relative modalità di calibrazione, le diverse modalità in cui può essere utilizzato e così via.

Datasheet elettronico

Oltre a ciò l'IEEE descrive minuziosamente le modalità di comunicazione fra nodi, senza tralasciare alcun mezzo di comunicazione: è presente la connessione cablata, quella wireless, quella RFID e così via.

Per fornire modularità allo standard esso è stato suddiviso in sottostandard la cui nomenclatura è nella forma 1451.0, 1451.1 e così via.

1.2 Lo standard 1451.0

Questo lavoro di tesi si è concentrato specificamente sul layer 1451.0 dell'ampissimo standard 1451. Esso si occupa della descrizione di uno strato di servizi e specifiche comuni atti all'interfacciamento dei sensori, al fine di garantire l'interoperabilità, indipendentemente dallo strato di comunicazione scelto.

All'interno dello standard sono descritte minuziosamente le specifiche di rappresentazione dei dati acquisiti, comprensivi di unità di misura, la varietà di TEDS previsti e le caratteristiche di ciascuno nonchè le modalità con le quali un nodo può accedere ai servizi di rete forniti da un altro sottostandard.

Oltre a quanto detto è definita una sequenza molto ampia di comandi che permettono la gestione complessiva dei nodi della rete. I comandi permettono di verificare le funzionalità degli smart sensor sulla rete, la loro configurazione e le loro caratteristiche tecniche.

Sono definiti i formati di due pacchetti atti a contenere i comandi 1451.0 e le relative risposte. Il tutto è corredato di API scritte in IDL in modo da essere *language-indipendent*.

Nodo TIM e NCAP

Per potenziare ulteriormente le possibilità di questo layer, lo standard descrive due tipologie di nodi distinti:

TIM (Transducer Interface Module) ovvero dispositivi cui spetta la gestione del sensore/trasduttore, la raccolta dei dati e la trasmissione di essi. Sono considerati dei nodi terminali di rete e si appoggiano direttamente ai dispositivi della seconda categoria, quella degli

NCAP (Network Capable Application Processor) ai quali spetta la gestione della rete e l'inoltro dei messaggi che circolano in essa, alla stregua dei router. Questi dispositivi operano anche da gateway tra gli utenti della rete standard ed i TIM, nel caso fosse previsto l'accesso dall'esterno.

1.3 Il nodo TIM

In questo lavoro si è cercato di realizzare l'implementazione di un nodo TIM dello standard 1451.0. Come piattaforma di sviluppo si è utilizzata una *evaluation board* prodotta dalla Microchip®, la PicdemZ.

Essa è un esempio perfetto di un potenziale nodo TIM reale, è infatti dotata di un microcontrollore della famiglia PIC18, un transceiver radio 802.15.4 per la comunicazione wireless (descritta nello standard 1451.5) e un sensore di temperatura. PicdemZ

1.4 Problematiche principali

Il problema principale è di natura pratica: esso proviene dal fatto che lo standard 1451.0, essendo corredato di API assai complesse, da definizioni di tipi di dato piuttosto elaborate, nonchè da meccanismi di funzionamento assai articolati si scontra con le limitate caratteristiche hardware di un microcontrollore il cui handicap principale è la taglia delle memorie, sia quella programma, sia quella RAM.

Altro problema non banale è la realizzazione di una architettura software che sia strutturalmente in grado di supportare tutte le peculiarità previste dallo standard.

Da non sottovalutare comunque le problematiche relative alla gestione degli eventi che concernono la ricezione e trasmissione di pacchetti 802.15.4 e le condizioni di sincronizzazione fra questi e l'esecuzione *normale* del codice.

Nel proseguio di questo lavoro si cercheranno di presentare con maggiore dettaglio le problematiche prese in esame e verrà esposta al lettore una possibile strategia implementativa.

Capitolo 2

Standard IEEE 1451

In questo capitolo viene presentato lo standard IEEE 1451, le sue finalità e ambito operativo e i suoi componenti. Si illustrano poi le caratteristiche di IEEE 1451.0, argomento centrale all'interno di questo lavoro di tesi.

Prima di intraprendere la lettura di questo capitolo, si consiglia vivamente di leggere il documento dello standard IEEE 1451.0 (si veda [14507a]) in modo da comprendere in dettaglio tutti gli aspetti principali. Altra lettura consigliata dall'autore è lo standard IEEE 1451.5 (si veda [14507b]), poichè saranno più chiare le interazioni che sussistono fra i due layer della stessa famiglia di standard.

2.1 Overview dello standard 1451

Un trasduttore è un sistema (nel senso della teoria dei segnali) che converte

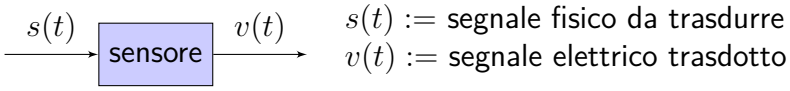

Figura 2.1: Il sistema *sensore*

l'energia da un dominio ad un altro. Qualora la trasformazione avvenga da un segnale di tipo fisico, chimico, biologico verso un segnale elettrico, si parla speficamente di sensore. Come è possibile dedurre dalla figura 2.1, un sensore è pertanto in grado di *acquisire* e dunque *misurare* una grandezza, come la pressione, la temperatura o la velocità di rotazione, ecc.. e convertirla in un segnale elettrico di tensione o di corrente.

sensore: definizione qualitativa

attuatore: definizione qualitativa

Un attutatore è invece il sistema in grado di effetuare il *mapping* inverso ovvero da grandezza elettrica a grandezza fisica (vedi figura 2.2). Esso converte dunque un segnale elettrico in un'altra grandezza fisica proporzionale al segnale elettrico stesso: ad esempio una tensione elettrica in velocità di rotazione dell'albero di un motore.

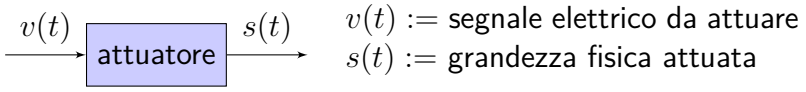

Figura 2.2: Il sistema *attuatore*

Il dato o più propriamente *campione* fornito dai sensori è sovente un dato *raw* che richiede un' ulteriore elaborazione per poter essere utilizzato in un sistema di controllo o misura. A questo proposito, viene spesso inserito un blocco di post-elaborazione, che esegue alcune operazioni di condizionamento sulla grandezza restituita dal sensore in modo da fornire in uscita un valore più significativo della grandezza misurata.

Smart sensor

Questa cascata di *atomic building-block* (vedi figura 2.3) costituisce un cosiddetto

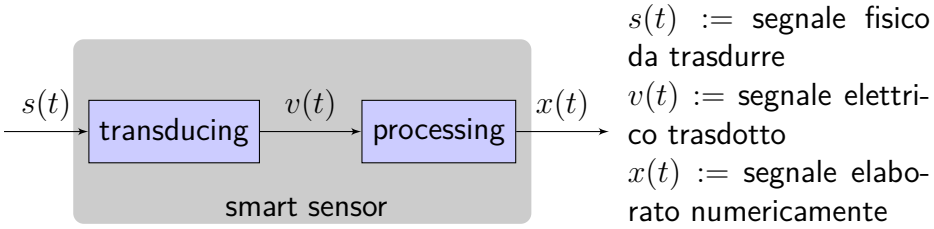

Figura 2.3: Schema a blocchi di uno smart sensor: cascata di un modulo di trasduzione e da uno di condizionamento numerico del segnale

sensore intelligente, in quanto unione di un trasduttore tradizionale con una unità di elaborazione numerica (ad esempio un microcontrollore o un microprocessore) in grado di convertire il dato grezzo, solitamente corrispondente ad una grandezza elettrica (tensione) proporzionale alla grandezza fisica misurata, in un valore numerico processato in maniera opportuna.

Un sensore classico che misura la pressione fornisce in uscita un valore di tensione proporzionale al valore misurato, un sensore intelligente invece fornirà un

valore numerico elaborato, già normalizzato, corrispondente al valore di pressione rilevato.

Lo standard IEEE 1451 fornisce una standardizzazione degli *smart sensor*, in modo tale da rendere sistemi basati su sensori diversi, forniti da costrutttori diversi, interoperabili tra loro.

Un IEEE 1451 *smart transducer* è un trasduttore intelligente standardizzato che affianca alla semplice capacità di misura o di attuazione, tutte quelle funzionalità che permettono al trasduttore di rielaborare i dati acquisiti, gestire autonomamente la propria descrizione, calibrazione, diagnosi e comunicazione con altri dispositivi.

Un trasduttore viene ritenuto intelligente (smart) e standardizzato IEEE 1451 se risponde a tre fondamentali caratteristiche:

- è descritto da un datasheet elettronico (detto TEDS nello standard)
- il controllo e i dati associati al trasduttore sono digitali
- il trigger, lo stato del trasduttore e il controllo sono forniti per mantenere il corretto funzionamento del sensore stesso.

Per fare questo, lo standard prevede l'inserimento all'interno di ciascun trasduttore smart di un vero e proprio datasheet elettronico e della logica necessaria per comunicare i propri dati.

1451 Environment: NCAP & TIM

Lo standard descrive inoltre uno scenario operativo in cui mira a collocarsi, assai complesso e variegato.

Tale *environment* è costituito da una rete di nodi embedded di due varietà: il Network Capable Application Processor (NCAP) e il Transducer Interface Module (TIM). Per la comprensione di cosa si intende per rete embedded si veda la definizione fornita [TS07].

Il nodo TIM contiene l'interfaccia di comunicazione con gli altri nodi (NCAP), il regolatore di segnale, i convertitori analogico-digitale o digitale-analogico (ADC o DAC) ed il trasduttore stesso. Esso può essere semplice, avendo a bordo un solo trasduttore, ma può ospitarne anche molti, accrescendo il suo grado di complessità e funzionalità.

L'NCAP è un'altra tipologia di nodo che funge da *gateway* per la rete di nodi TIM. Esso è un dispositivo costituito da un processore con due o più interfacce di comunicazione, una rivolta verso la rete 1451 e l'atra verso una rete TCP/IP più tradizionale. Il nodo NCAP fornisce quindi funzionalità di appoggio alla rete di nodi TIM consentendo quindi ad un utente di una rete tradizionale di comunicare con i nodi TIM via nodo NCAP. Un grafico chiarirà senz'altro la topologia della rete ipotizzata nello standard e le sue caratteristiche, si veda pertanto la figura 2.5.

Da notare che uno smart transducer può comprendere al suo interno sia le caratteristiche del nodo TIM che quelle del nodo NCAP, realizzando così un nodo *stand-alone* in grado di autointerfacciarsi ad una rete.

2.2 I componenti dello standard 1451

Gli standard della famiglia 1451

Vista la vastità di elementi che lo standard si propone di descrivere e standardizzare, esso è stato suddiviso in porzioni più piccole, ciascuna deputata alla descrizione di porzioni operative distinte dello standard. Si riporta in figura 2.4 uno schema rappresentante i vari componenti e relativo ambito di applicazione della famiglia IEEE 1451. La figura 2.4 è assai utile per la comprensione dei

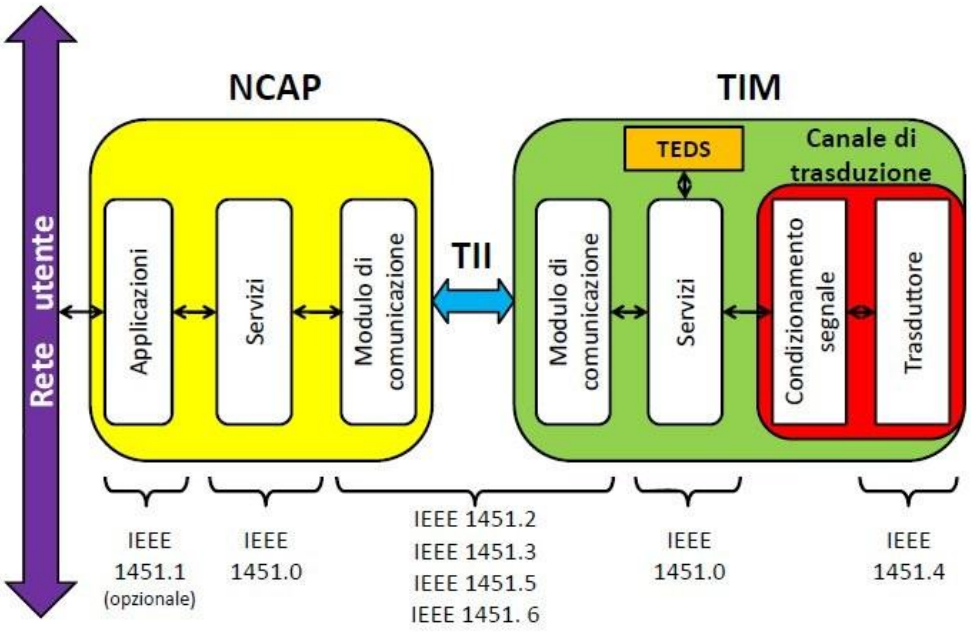

Figura 2.4: I componenti dello standard IEEE 1451 (prospettiva tra nodo TIM e nodo NCAP)

ruoli dei componenti dello standard. Come si vede, è rappresentato (schematicamente) un nodo NCAP, connesso (non importa il come) ad una rete di tipo più tradizionale. Esso è a sua volta interfacciato ad un nodo TIM. Il nodo NCAP come prima annunciato è dotato di due interfacce di comunicazione: una verso la rete utente e una verso i nodi TIM.

IEEE 1451.X

Lo standard IEEE 1451 è composto da diversi *sottostandard* che di fatto sono

layer, con un nome nella forma 1451.X con $X = 0, 1, \ldots, 7$. Ciascuno ha uno scopo differente che andiamo brevemente a riportare.

IEEE 1451.0

.0: comandi interoperabili di scambio dati, TEDS e configurazione

Il layer IEEE 1451.0 definisce un insieme di istruzioni e comandi che sono comuni a tutti gli elementi di IEEE 1451 e indipendenti dal mezzo di comunicazione (1451.X) tra il trasduttore e l'NCAP. Tra questi ci sono i comandi di base per leggere e scrivere i dati del trasduttore/attuatore, leggere e scrivere i TEDS, per inviare comandi di configurazione e controllo al TIM. L'obiettivo principale di IEEE 1451.0 è quello di ottenere una interoperabilità a livello dati e controllo all'interno di reti di sensori connessi in varie modalità, cablate o wireless.

IEEE 1451.1

.1: modello di smart transducer

Il layer IEEE 1451.1 definisce un modello comune per i componenti di uno smart transducer inserito in una rete, oltre alle specifiche riguardanti l'interfaccia. L'architettura software di questo layer è definita attraverso i tre modelli seguenti:

- un modello dati specifica il tipo e la forma delle informazioni che vengono comunicate attraverso l'interfaccia tra gli oggetti definiti da IEEE 1451.1, sia nel caso di comunicazioni locali che remote;
- un modello oggetto specifica il tipo delle componenti software usate per progettare ed implementare il sistema: in pratica tale modello fornisce una serie di blocchi con i quali costruire l'applicazione di controllo o misura;
- due modelli di comunicazione definiscono la sintassi delle interfacce software tra gli oggetti (o blocchi software) dell'applicazione e la rete. Di fatto, questo layer si concentra principalmente sulla comunicazione tra gli NCAP e tra un NCAP e altri nodi della rete.

IEEE 1451.2

.2: connessione wired point-to-point

Il layer IEEE 1451.2 definisce l'interfaccia di comunicazione e il formato dei TEDS nel caso di connessione *point-to-point* tra NCAP e trasduttore. Lo standard originale prevede l'utilizzo dell'interfaccia SPI (Serial Peripheral Interface) con l'aggiunta di alcune linee ulteriori per il controllo di flusso e la temporizzazione. Esso è in modifica per renderlo interfacciabile con IEEE 1451.0 e compatibile con altre interfacce comuni, tra cui UART (ovvero seriale).

IEEE 1451.3

.3: connessione multi-drop

Il layer IEEE 1451.3 definisce l'interfaccia di comunicazione e il formato dei TEDS nel caso di connessione *multidrop* tra NCAP e trasduttore. Esso permette di configurare i trasduttori di una rete come nodi che condividono la stessa linea di comunicazione.

IEEE 1451.4

.4: trasduttori analogici

Il layer IEEE 1451.4 definisce un'interfaccia mista per supportare trasduttori analogici che possano operare in modalità sia analogica che digitale. Lo scopo di questo layer è quello di aggiungere il TEDS ai vecchi sensori con interfaccia analogica. All'accensione del sistema, il TEDS viene inviato tramite interfaccia digitale ad un unico filo (one-wire) dopodichè, l'interfaccia viene commutata in modalità analogica in modo da poter essere utilizzata per trasportare l'informazione analogica dal trasduttore al sistema di misura.

IEEE 1451.5

.5: comunicazione wireless

Il layer IEEE 1451.5 definisce l'interfaccia di comunicazione e il formato dei TEDS nel caso di connessione wireless. Diverse soluzioni wireless sono supportate, come 802.11 (WiFi), 802.15.1 (Bluetooth) e 802.15.4 (ZigBee).

La rete wireless tra i TIM e l'NCAP può anche essere eterogenea: l'NCAP può implementare diversi tipi di interfaccia senza fili, per comunicare con i TIM. Ciascun TIM, invece, comunica con un solo NCAP e necessita quindi solamente dell'interfaccia appropriata. Un esempio di connessione wireless eterogenea tra NCAP e TIM è mostrata in figura 2.5.

IEEE p1451.6

.6p: CANopen

Il layer IEEE p1451.6 (proposto) definisce l'interfaccia di comunicazione e il formato dei TEDS nel caso di connessione tramite *CANopen*.

IEEE p1451.7

.7p: RFID

Il layer IEEE p1451.7 (proposto) definisce l'interfaccia di comunicazione e il formato dei TEDS nel caso di connessione tra trasduttori e dispositivi *RFID (Radio Frequency Identification)*.

Network 1451

Il lettore è ora pronto a comprendere il complesso environment di rete proposto nello standard 1451 e rapresentato in figura 2.5. Come si capisce dall figura, i

nodi NCAP sono connessi ai nodi TIM mediante porzioni distinte dello standard a seconda delle necessità dell'ambiente. In questo lavoro ci si è concentrati sullo

Rete 1451 eterogenea

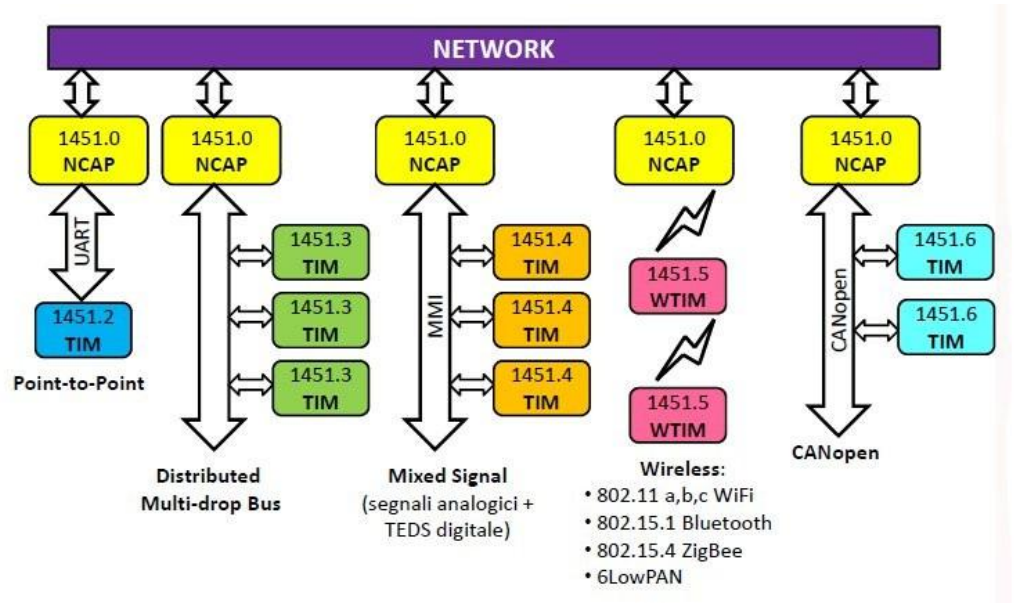

Figura 2.5: Il network 1451: diversi canali di trasmissione fisica accomunati da servizi interoperabili fra loro

standard IEEE 1451.0, di cui si tratta nel seguito.

2.3 1451.0

Lo standard IEEE 1451.0 descrive al suo interno tutte le parti fondamentali e comuni della famiglia 1451.

In esso vengono descritti nomi e sigle di uso comune dello standard, i tipi di dato, le convenzioni utilizzate e la struttura del messaggio e dei comandi. Una porzione rilevante del 1451.0 è dedicata alla standardizzazione dei TEDS.

I comandi sono divisi in due categorie: standard e definiti dal produttore. Indipendentemente dalla classe, ogni comando è composto da due byte: il più significativo definisce la classe, il meno significativo, chiamato funzione, identifica il comando specifico all'interno della classe.

Comandi 1451

La tabella 2.6, corrispondente alla tabella 15 del 1451.0 definisce le classi dei comandi. Per ognuna delle classi di comandi specificate sono presenti in [14507a] anche le tabelle con i relativi codici dei comandi, non riportate quì per brevità

cmdClassId	Attribute name	Category
0	Reserved	Reserved
1	CommonCmd	Commands common to the TIM and TransducerChannel
2	XdcrIdle	Transducer idle state
3	XdcrOperate	Transducer operating state
4	XdcrEither	Transducer either idle or operating state
5	TIMsleep	Sleep state
6	TIMActive	Tim active state commands
7	AnyState	Any state
8—127	ReservedClass	Reserved
128—255	ClassN	Open for manufacturers – N = class number

Figura 2.6: Le classi di comando del protocollo 1451.0

ma che si consiglia caldamente di visionare per comprendere gli scenari operativi presentati nel capitolo sul design del 1451.0.

2.3.1 1451.0: TEDS

I TEDS (Transducer Electronic Data Sheets) corrispondono alla versione elettronica dei data sheets degli stessi trasduttori e contengono tutte le informazioni necessarie per la loro gestione.

TEDS e Virtual TEDS

Essi sono concepiti per essere memorizzati in una parte di memoria non volatile del TIM. Se per alcuni motivi non possono essere memorizzati direttamente nel TIM, essi possono essere allocati in altre memorie e in questo caso si parla di TEDS virtuali.

Un TIM può avere un insieme di TEDS sia nella propria memoria, sia virtuali. E' disponibile un meccanismo per sapere se un TEDS è o meno virtuale, utilizzando il comando `Query TEDS` si ottiene in risposta un indicatore che specifica la tipologia di TEDS.

Generalmente un TEDS non viene programmato una sola volta dal produttore o dall'utente. Il TEDS viene continuamente cambiato e aggiornato durante lo stato operativo del sensore. Lo stesso trasduttore può modificare il TEDS in base ad un evento determinato.

Adaptive TEDS

Per rendere questo possibile, viene abilitato il bit `Adaptive` che indica se il TEDS può essere modificato dalla logica interna al TIM oppure no. I TEDS definiti in questo standard sono molteplici, ma quattro sono quelli da implementare obbligatoriamente in ogni TIM:

- Meta-TEDS
- TransducerChannel-TEDS
- User's Transducer Name-TEDS

- Phy-TEDS

Meta-TEDS

caratteristiche temporali (timeout)

Il Meta-TEDS fornisce alcuni parametri temporali, al caso peggiore, che sono utilizzati dall'NCAP per settare i propri timeout e capire quando il TIM non risponde o scadono le sue richieste. Questo TEDS mette a disposizione le informazioni comuni di tutti i trasduttori disponibili (trasducer channel) e le informazioni necessarie per l'accesso ai trasduttori. Esso viene letto con un comando `Query TEDS` o `Read TEDS`. L'argomento del comando identifica il TEDS che deve essere interrogato.

Solitamente questo TEDS è in sola lettura per evitare modifiche impreviste, per questo TEDS i comandi `Write TEDS` e `Update TEDS` non vengono abilitati.

TransducerChannel-TEDS

Datasheet del trasduttore

Il TransducerChannel-TEDS fornisce informazioni dettagliate per ogni singolo trasduttore. Esso fornisce le grandezze che sono misurate o controllate dal trasduttore, i *range* in cui il trasduttore opera, le caratteristiche di I/O digitali, le informazioni sui tempi di interrogazione.

Anch'esso viene interrogato tramite i comandi `Query TEDS` o `Read TEDS` ed è implementato in modaltà sola lettura per non ottenere effetti indesiderati da un'incoretta interrogazione.

User's Transducer Name-TEDS

Questo TEDS mette a disposizione un'area dove viene memorizzato il nome che l'utilizzatore fornisce e utilizza per identificare il trasduttore.

Phy-TEDS

Questo TEDS dipende dal mezzo di comunicazione utilizzato per la comunicazione tra il TIM e NCAP.

Esso non viene specificato nel 1451.0, bensì essendo legato alla tecnologia di trasmissione, viene definito dagli altri componenti della famiglia 1451.X.

Ogni TEDS ha un suo Data Block. Il data block comprende tutti i campi previsti per il TEDS in questione. Ogni campo è descritto in maniera particolareggiata nello standard. Le tabelle riassuntive come la tabella 43 di [14507a] mostrano come per ogni campo sia specificato il tipo, in cui è inserito il numero identificativo, il nome associato al campo, la descrizione, il tipo di dato contenuto e il numero massimo di ottetti (*aka* byte) previsti, ovvero la dimensione massima del campo.

Il nome associato al campo (*field name*) è il nome utilizzato quando deve essere trasmesso il TEDS o una parte di esso ad un'altra applicazione. Il solo numero identificativo è poco caratterizzante, mentre il nome del campo rende più chiari e comunque univocamente distinti i valori trasmessi.

Struttura TLV

Il TEDS utilizza la struttura dati `Type, Lenght, Value (TLV)`: in questo modo ogni valore del TEDS è contenuto in una ennupla di TLV. In un TLV il campo `Type` contiene un numero identificativo. Il valore è contenuto nel campo `Value` (i numeri identificativi sono mostrati nei data block e a parte 2 e 3 gli altri codici cambiano in ogni TEDS). Nel campo `Length` è specificata la lunghezza del campo Value in numero di ottetti.

2.3.2 1451.0: API

Lo standard 1451.0 definisce due tipi di API: la prima - *Trasducer Service Interface* - definisce una API del solo NCAP, utilizzata dalle applicazioni di misura e controllo lato utente, per accedere al layer IEEE 1451.0.

1451.0 TSI API

Questa API contiene i metodi per leggere e scrivere i TEDS, per gestire i TrasducerChannels, e infine per inviare comandi di configurazione e controllo ai TIM. Ovviamente quest'ultimo, equipaggiato della controparte di questa API deve essere in grado di processare ed eseguire i comandi richiesti dal nodo NCAP.

Inoltre è definita un'interfaccia opzionale da implementare, contenente le applicazioni per la lettura di stream di dati dagli apparati di misura e altre possibili operazioni.

1451.0 MCI API

L'altra API - *Module Communication Interface* - si colloca tra lo standard 1451.0 e un'altro membro della famiglia 1451. Essa è un'interfaccia simmetrica che va implementata sia sull'NCAP che sul TIM.

Questa API contiene metodi che dovrebbero essere implementati dal layer IEEE 1451.X e chiamati per iniziare le operazioni di comunicazione. Similarmente ci sono metodi che devono essere implementati dal 1451.0 e invocati dal 1451.X per consegnare le informazioni inviate.

Interface Definition Language

Per mantenere una neutralità di linguaggio, le funzioni e i parametri delle API sono descritte con il linguaggio *Interface Definition Language (IDL)*.

Al livello superiore si trova il modulo `IEEE1451Dot0`, il quale a sua volta contiene i moduli:

- `TransducerServices`
- `ModuleCommunications`
- `Args`
- `Util`.

Nel modulo TrasducerServices sono specificate le API del lato NCAP necessarie all'utente per accedere al 1451.0. Esso contiene le classi e le interfacce per la scoperta e la registrazione dei TIM, per l'accesso ai Trasducer Channel, per effettuare misurazioni o comandare gli attuatori, per gestire l'accesso ai TIM e per leggere e scrivere sui TEDS. TransducerServices

In ModuleCommunications sono definite le API con la quale gestire la comunicazione tra NCAP e TIM. Sono disponbili interfacce sia per la comunicazione point-to-point sia per le reti. ModuleCommunications

Args e Util costituiscono la porzione dello standard dove sono descritte le tipologie di dato supportate e le specifiche di *marshalling* e *unmarshalling* dello standard.

2.3.3 1451.0: Indirizzamento e messaggi

Lo standard 1451 definisce anche una modalità di indirizzamento fra nodi e i relativi pacchetti di comunicazione. Per gli indirizzi sono disponibili 16 bit, così utilizzati: Messaggi 1451.0

0x0000 significa che il mesaggio è diretto all'intero nodo TIM

0x0001 - **0x7FFF** sono gli indirizzi possibili dei transducer channel

I restanti indirizzi sono riservati ai gruppi virtuali di transducer channel. Si veda [14507a] per maggiori informazioni. . Per quanto riguarda i pacchetti di

1-Octet							
7	6	5	4	3	2	1	0
Destination TransducerChannel Number (most significant octet)							
Destination TransducerChannel Number (least significant octet)							
Command class							
Command function							
Length (most significant octet)							
Length (least significant octet)							
Command-dependent octets . . .							

Figura 2.7: Il formato di un pacchetto di comando 1451

comunicazione fra nodi, riportiamo il packet definito all'interno dello standard per le richieste (vedi figura 2.7) e il formato delle risposte (vedi 2.8). .

1-Octet							
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Success/Fail Flag							
Length (most significant octet)							
Length (least significant octet)							
Reply-dependent octets . . .							

Figura 2.8: Il formato di un pacchetto di risposta 1451

Capitolo 3

TIM: Picdem Z e *framework*

In questo capitolo presentiamo le caratteristiche della scheda PicdemZ e degli strumenti ausiliari (sia software che hardware) che si sono utilizzati nello sviluppo di questo lavoro. Dopo una overview dell'hardware, vengono fornite alcune informazioni addizionali per comprenderne meglio le peculiarità. La lettura di questo capitolo non sostituisce assolutamente quella dei manuali a corredo e dei datasheet.

3.1 Overview del materiale

Nel seguito vengono illustrati gli elementi che compongono l'ambiente di lavoro in cui questo progetto si è collocato. Trattiamo le caratteristiche fondamentali della scheda PicdemZ, del microcontrollore di cui è dotata (PIC18LF4620) e del *programmer* che si è utilizzato per lo sviluppo del codice (MPLAB ICD2).

Per completezza vengono riportati inoltre gli altri strumenti a corredo: MPLAB IDE e MPLAB C18 per la stesura del codice, il Digitus Serial-USB Adapter per l'interfacciamento seriale alla scheda e il software fornito da Microchip® da cui è stato evinto qualche suggerimento per la stesura del progetto. Si faccia inoltre riferimento al lavoro di [Ber08] per ulteriori utili spunti.

3.1.1 PicdemZ

PicdemZ board

La scheda PicdemZ è una scheda per sviluppo/testing prodotta dalla azienda Microchip®. È equipaggiata di:

- un microcontrollore (d'ora innanzi spesso *uC*) Microchip® PIC serie 18 (PIC18F4620) su zoccolo da 40 pin;

- un sensore di temperatura Microchip® TC77 a 5 pin connesso al microcontrollore tramite Serial Peripheral Interface (SPI);
- due LED collegati alle uscite digitali del microcontrollore;
- due pulsanti collegati agli ingressi digitali del microcontrollore;
- un pulsante di reset;
- un connettore ICSP RJ-11 a 6 pin per la programmazione del firmware;
- un connettore RS-232 per la comunicazione seriale;
- un connettore a 12 pin per l'alloggiamento del modulo transceiver radio MRF24J40MA (802.15.4™);
- un connettore per l'alimentazione da batteria (e relativo switch);
- un connettore jack femmina per l'alimentazione esterna;
- un'area di prototyping dotata di tensione +3.3V e relativa linea GND.

Si veda [Mic08c] per maggiori informazioni. Nella figura 3.1 è possibile prendere visione della scheda nella sua interezza e della posizione dei componenti sopraccitati.

3.1.2 MPLAB®ICD2

Insieme con la scheda è fornito uno strumento hardware (detto *programmer*) per la programmazione della memoria del uC Chiameremo frequentemente nel seguito tale operazione (di programmazione del uC) *flashing*.

MPLAB ICD2

Il modello in questione è il MPLAB®ICD2: esso è idoneo a molti altri tipi di schede dotate dello stesso connettore ICSP RJ-11 a 6 pin, molto comune su *board* Microchip®. Come è possibile leggere nel manuale [Mic07a], esso è un modello assai avanzato, dotato infatti della funzionalità di *in-circuit-debugging*.

L'*in-circuit-debugging* consiste nella possibilità di inserire dei *breakpoint* nel codice, letteralmente punti di interruzione, che sono di fatto iniettati all'interno del microcontrollore durante il *flashing* in modo da poter arrestare l'esecuzione del codice direttamente *nel* microcontrollore. Tale caratteristica lo rende assolutamente idoneo nello sviluppo di software e rende il processo di *debugging* molto più indolore.

Il Microchip MPLAB ICD 2 riunisce pertanto le caratteristiche di un In-Circuit Debugger (ICD) e di un In-Circuit Serial Programmer (ICSP) nello stesso dispositivo.

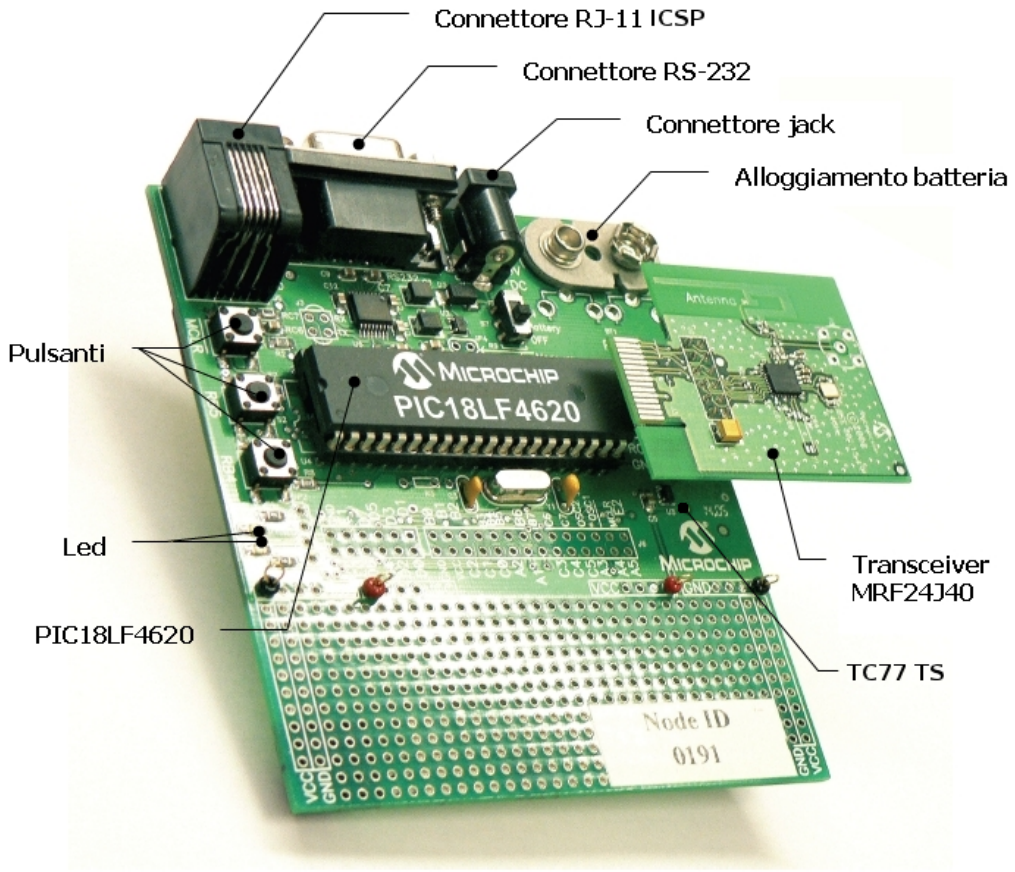

Figura 3.1: La scheda Microchip® PicdemZ

Esso è studiato per funzionare da supporto nelle fasi di valutazione, debug e programazione dei dispositivi in un ambiente di laboratorio.

Esso fornisce diverse funzionalità come l'esecuzione del codice in tempo reale e passo-passo, il monitoraggio e la modifica di variabili e registri, il debug direttamente nel circuito di prova, il monitoraggio della tensione di alimentazione del circuito, led di diagnostica, un'interfaccia utente con Microchip® MPLAB IDE e una connessione RS-232 o USB per il collegamento con il pc.

Questo dispositivo permette di programmare il Microchip PIC18LF4620 senza dover togliere il microcontrollore dalla scheda. Tale fatto comporta un notevole risparmio di tempo e una maggiore manegevolezza del dispositivo.

Si veda la guida [Mic07a] per il setup del programmer e per altre informazioni utili.

Figura 3.2: Il programmer MPLAB ICD2

3.1.3 Strumenti ausiliari

USB2Serial Adapter

Figura 3.3: Digitus USB/Serial Adapter

Per monitorare lo stato di esecuzione del PicdemZ si è utilizzata estensivamente la comunicazione seriale. Essendo ormai tutti i computer di fabbricazione recente sprovvisti della vetusta porta DB-9 per comunicazione seriale, si è utilizzato il convertitore usb-seriale della Digitus®. È riportata in figura 3.6 una immagine di quest'ultimo.

ZENA^{TM} Wireless Network Analyzer board

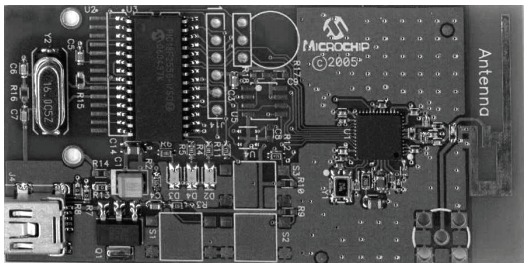

Figura 3.4: ZENA^{TM} Wireless Network Analyzer board

Per verificare il funzionamento del modulo transceiver MRF24J40MA si è utilizzata la scheda di fabbricazione Microchip®, ZENA. Quest'ultima, corredata di un programma di acquisizione, anch'esso fornito dalla Microchip®, consente di catturare i pacchetti (anche malformati) dello standard 802.15.4 presenti entro una certa distanza da essa stessa.

Mediante il software e la scheda è pertanto molto semplice monitorare tutto il traffico 802.15.4 prodotto dal(i) transceiver radio e analizzarne le problematiche. I canali supportati dall'analyzer sono quelli nello spettro di frequenze dei 2.4 GHz e in particolare quelli dall'11 al 26, secondo la definizione del protocollo.

È connessa al pc mediante cavo usb. Si veda il manuale [Mic06b] a corredo per l'installazione e utilizzo.

Oscilloscopio Agilent MSO6012A

Strumento che si è reso utilissimo per verificare il corretto funzionamento della porzione di codice che gestisce l'hardware della scheda è l'oscilloscopio Agilent MSO6012A. Esso, corredato di una sonda per acquisizione digitale da 16 bit è stato applicato ai pin del microcontrollore destinati alla comunicazione sul bus SPI.

3.2 Hardware onboard

Prendiamo ora in analisi con maggiore attenzione i componenti presenti sulla scheda di nostro interesse. Si veda il manuale [Mic08c] per le connessioni.

3.2.1 uC PIC18F4620

PIC18F4620

Il uC montato a bordo della PicdemZ è il PIC18F4620. Facilmente intuibile dal nome dello stesso, la famiglia di appartenenza, ovvero la PIC18. Con tale dicitura la casa produttrice identifica una famiglia di microcontrollori *middle-range* dal punto di vista delle periferiche in dotazione e dalla taglia della memoria. Si veda la sezione 3.3 per maggiori informazioni sull'architettura.

Riportiamo le caratteristiche essenziali del PIC18F4620:

Caratteristiche principali

- memoria programma 64 kB di tipo Flash
- CPU Speed fino a 10 MIPS (massima frequenza supportata 40MHz)
- memoria RAM di 3968 byte
- memoria eeprom *non-volatile* di 1024 byte
- interrupt a bassa e ad alta priorità
- oscillatore interno a disposizione
- *WatchDog Timer*
- 3 porte digitali da 8 bit ciascuna (PORT A, B, C)
- 2 porte digitali da 4 bit ciascuna (PORT D, E)
- Tutte le 5 porte sono *multiplexate* con altre periferiche:

Periferiche onboard

- 3 interrupt esterni programmabili
- 1 ECCP Enhanced Capture/Compare/PWM module
- 1 CCP Capture/Compare/PWM module
- 1 MSSP Master Synchronous Serial Port (3-Wire SPI and I^2C)
- 1 USART module (RS-232, RS-485, ecc..)
- ADC a 13 canali da 10 bit
- 2 Analog comparator
- 1 timer da 8 bit
- 3 timer da 16 bit

Per conoscere nella sua interezza le molteplici caratteristiche del uC si veda il *datasheet* [Mic08b].

3.2.2 LED

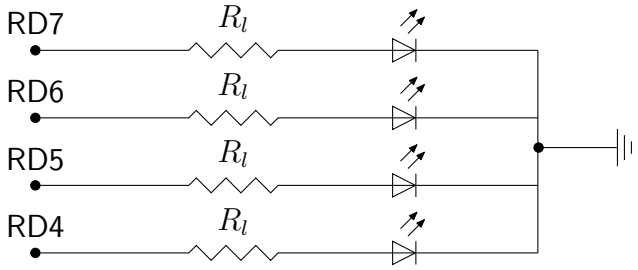

Figura 3.5: Diagramma di connessione dei led addizionali al PIC18LF4620

La scheda PicdemZ è dotata di due led verdi connessi ai pin RA0 e RA1 del PIC18F4620. Per aumentare il feedback visivo della scheda durante tutta la fase di sviluppo, si è ritenuto utile connettere altri quatrro led alla PORTD del uC, in particolare ai pin D7, D6, D5, D4 secondo lo schema di figura 3.5.

3.2.3 Temperature Sensor TC77

Sulla scheda è presente un sensore di temperatura, anch'esso di fabbricazione Microchip®, come è possibile vedere in figura 3.1 (TC77 TS). Il TC77 comunica con il uC mediante bus SPI e pertanto condivide quest'ultimo con la scheda radio. È dotato di 5 pin anche se inserito in un package da 8. La temperatura acquisita dal sensore è resa disponibile in una parola binaria in complemento a due da 13 bit. La sensibilità dello stesso è variabile a seconda del range di temperatura:

Sensore di temperatura

- da $+25°$ C a $+65°$ C accuratezza di $\pm$ 1° C
- da $-40°$ C a $+85°$ C accuratezza di $\pm$ 2° C
- da $-55°$ C a $+125°$ C accuratezza di $\pm$ 3° C

Importante precisare come sia possibile ottenere la lettura della temperatura esclusivamente mediante *polling* essendo il TC77 sprovvisto di una linea TTL che vari stato, al completamento dell'acquisizione di un nuovo campione. Si faccia riferimento a [Mic02] e a [BL04] per maggiori informazioni.

3.2.4 Daughter board MRF24J40MA

Insieme con la scheda PicdemZ sono fornite in dotazione alcune schede wireless 802.15.4 note con il nome di MRF24J40MA. Esse sono corredate del transceiver

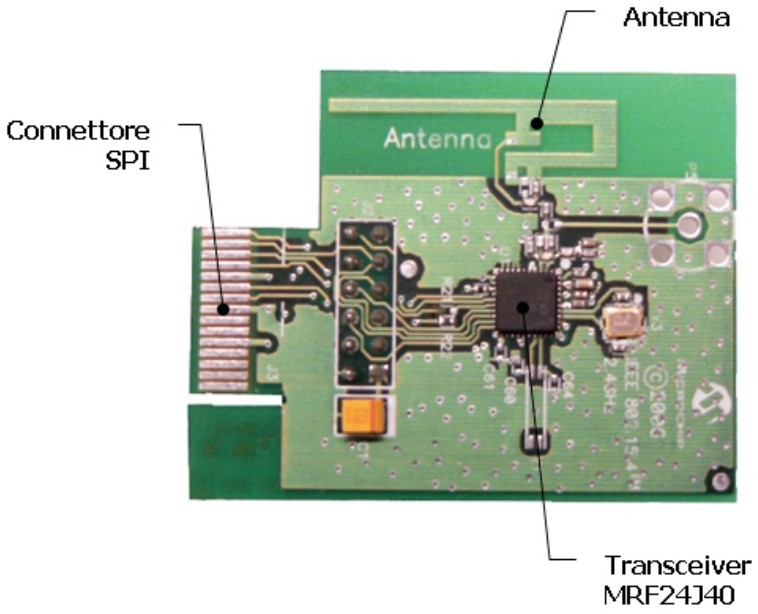

Figura 3.6: Microchip® MRF24J40MA 2.4 GHz IEEE Std. 802.15.4™ RF Transceiver Module

Modulo radio

radio MRF24J40. *(Nota bene: la scheda ha un nome che termina in MA, il transceiver no),* Il modulo radio è provvisto di un'antenna stampata incorporata che garantisce una copertura di qualche decina di metri. È tuttavia predisposto un alloggiamento dove installare un connettore per un'eventuale antenna esterna. Il modulo MRF24J40MA è alloggiabile sulla PicdemZ mediante il connettore a 12 pin visibile in figura 3.1 (perciò detta *daughter board*). Il transceiver comunica con il uC mediante bus SPI™ e invia al PIC un segnale di interrupt al verificarsi di determinati eventi (come ad esempio la ricezione di un messaggio).

Si veda il manuale della PicdemZ [Mic08c] e il datasheet della daughter board [Mic08a] per maggiori informazioni. La MRF24J40MA è connessa al PIC18F4620 secondo lo schema di figura 3.7 tramite il connettore a 12 pin presente sulla scheda. I connettori denominati MISO, MOSI, SCK formano l'interfaccia di comunicazione SPI col uC, mentre il pin RB0 è deputato alla ricezione degli interrupt dal transceiver verso il PIC. Sono ovviamente presenti le linee di alimentazione.

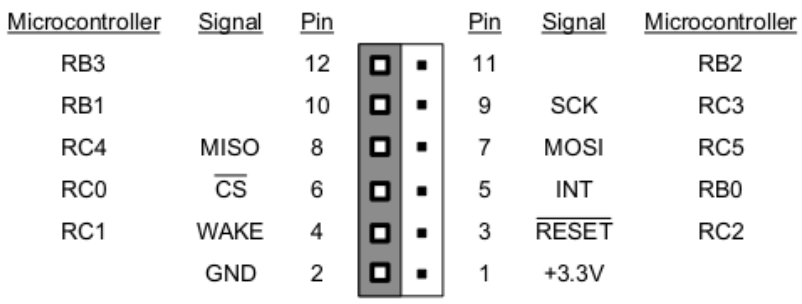

Figura 3.7: Il connettore a 12 pin che permette l'interfacciamento fra MRF24J40MA al PIC18F4620

3.3 Architettura PIC18

Le informazioni riportate nel seguito non hanno alcuna presunzione di completezza. Un microcontroller è in generale un dispositivo assai complesso e nella fattispecie, essendo il PIC18F4620 assai ricco di funzionalità, la complessità aumenta ulteriormente. Nel seguito sono presenti solamente alcune nozioni fondamentali atte alla comprensione delle problematiche presentate e analizzate più oltre nel testo.

Fondamentale ausilio alla comprensione delle caratteristiche è l'insostituibile datasheet (si veda [Mic08b]). Utile inoltre la lettura di [Hua05]; il testo è un ottimo punto di partenza per comprendere in maniera più ampia e organica le caratteristiche della famiglia di microcontrollori PIC18. Esso è stato utilizzato estensivamente per la stesura di questo lavoro e perciò se ne ringrazia l'autore. Di ausilio inoltre per la comprensione dell'hardware del uC il [Ibr08], pur essendo scritto con in mente un compilatore diverso da quello usato in questo progetto (in particolare mikroC®, sviluppato da mikroElektronika®).

Fatte dunque le premesse del caso, procediamo con la trattazione.

3.3.1 Memoria

Memoria PIC18

Il uC è dotato sostanzialmente di tre memorie distinte:

- data RAM (3968 byte)
- program memory (64 kB)
- data eeprom (1024 byte)

La prima di esse, sicuramente la più famigliare a qualunque informatico è la memoria RAM. Essa consta di 16 banchi di 256 byte ciascuno ed è indirizzabile in rappresentazione decimale con il range `0x000 - 0xFFF`. Come facilmente immaginabile, il primo *nibble* (ovvero 4 bit) è l'indirizzo del banco, i restanti due, fissato il banco sono l'indirizzo della specifica cella di memoria. Gli indirizzi sono pertanto a 12 bit.

Non si giunge però al totale di 4096 byte (256 byte $\times$ 16) poichè una porzione di questi è riservata a scopi diversi che all'allocazione delle variabili durante l'esecuzione. Nella fattispecie, gli utlimi 128 byte del banco 15 (`0xF80 - 0xFFF`) sono destinati agli *Special Function Registers o (SFR)*. Tali registri sono destinati al mantenimento dello stato e del controllo del microcontrollore e delle periferiche. I registri di memoria rimanenti sono invece denominati *GPR*, ovvero *General Purpose Register*.

La program memory invece è destinata a contenere le istruzioni che il uC dovrà eseguire non appena alimentato. Le istruzioni del programma sono pertanto contenute in uno *storage* differente rispetto a quello delle variabili. Questa peculiarità, ovvero la separazione tra programma e dati, va sotto il nome di architettura Harvard.

La program memory è composta di 65536 byte indirizzabili nel range `0x0000 - 0xFFFF`. Fanno eccezioni alcune locazioni di memoria destinate ad un uso specifico:

0x0000 reset vector

0x0008 high-priority interrupt vector

0x0018 low-priority interrupt vector

Sono di grande interesse i vector per gli interrupt. Essi contengono infatti l'indirizzo della routine cui saltare qualora si verifichi un interrupt. Ovviamente, a seconda della priorità dell'interrupt, si userà il contenuto della cella `0x0008` piuttosto che `0x0018`; di fatto è possibile impostare il microcontrollore affinchè gestisca gli interrupt ignorandone la priorità.

La data eeprom è destinata ad archiviare dati *non-volatile* ovvero persistenti nonostante la rimozione di tensione dal microcontrollore. Tale memoria è una sorta di micro-hard-disk a disposizione dell'utente. È possibile effettuare letture e scritture su tale memoria sfruttando alcuni SFR dedicati.

Le informazioni quì riportate sono state vitali per la scrittura del linker script del progetto. Si faccia riferimento al capitolo sull'implementazione per ulteriori informazioni.

3.3.2 Periferiche

Il PIC18F4620 è assai ricco di funzionalità come si avrà avuto modo di capire. Le periferiche importanti ai nostri scopi sono comunque sostanzialmente due: la porta seriale e il bus SPI™.

È oltre lo scopo di questo documento fornire una trattazione organica della comunicazione seriale; si rimanda pertanto al capitolo 18 del datasheet [Mic08b] dove vengono spiegati tutti i dettagli e gli SFR utilizzati dal modulo *enhanced universal synchronous receiver transmiter (EUSART)*, ovvero il modulo deputato alla comunicazione seriale. Si noti comunque che la PicdemZ dispone delle piste necessarie per porre in connessione i pin deputati alla comunicazione mediante EUSART con la porta DB-9.

Nel capitolo sui driver verrà illustrato al lettore il codice che consente di inviare e ricevere caratteri su tale porta.

Modulo MSSP: bus SPI

Di maggiore interesse probabilmente il modulo MSSP, ovvero *master synchronous serial port*. Tale modulo, la cui spiegazione si trova nel capito 17 del datasheet, supporta la comunicazione mediante differenti tipi di bus:

- SPI™
- I^2C

Rilevante nel nostro scenario operativo il bus SPI poichè mediante quest'ultimo il microcontrollore si può interfacciare al transceiver MRF24J40. Tale bus è comunque molto popolare anche per la comunicazione verso memorie eeprom esterne e molti altri tipi di circuiti integrati.

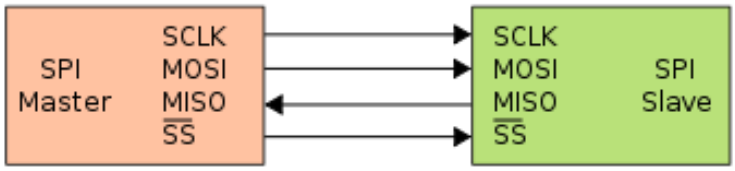

Figura 3.8: Interfaccia di comunicazione SPI: dispositivi master e slave

Interfaccia SPI

In un tipico scenario operativo il microcontrollore ricopre il ruolo di *master* nella comunicazione, mentre un altro dispositivo (potenzialmente un altro microcontroller) riveste il ruolo di *slave*.

Il master si occupa di inviare su di una linea il segnale SCK (spesso noto anche come SCLK), ovvero il clock della comunicazione, al dispositivo slave.

Master e slave sono poi connessi con altre due linee note come SDI e SDO (note altresì come MISO e MOSI) mediante le quali i due dispositivi si scambiano in modo sincrono un byte alla volta. Si veda lo schema di figura 3.8 per maggiore chiarezza.

Spesso alle tre linee ne è aggiunta una quarta, detta CS (o SS) il cui scopo è fungere da selettore di slave device, negli ambienti cosiddetti *multislave*.

Si immagini dunque un microcontrollore che funge da master sia connesso a più dispositivi slave condividento le linee SCK, SDI e SDO. Desiderando che

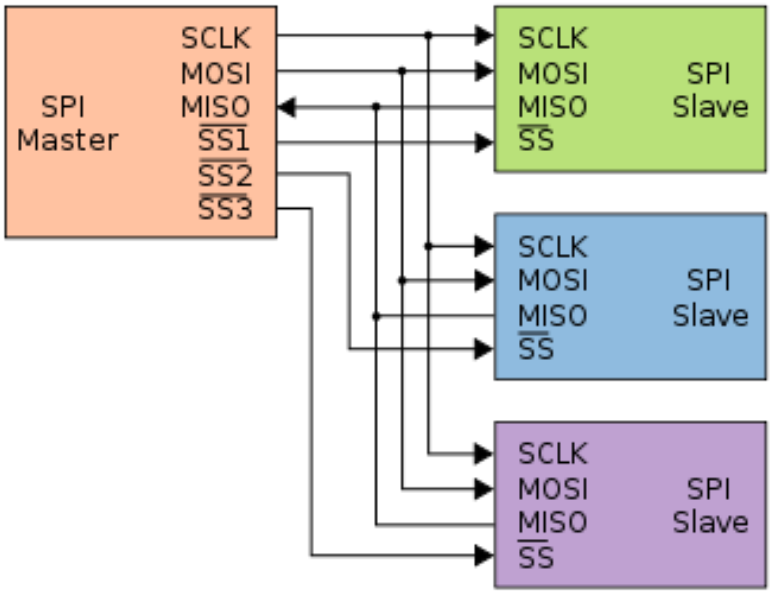

Figura 3.9: Interfaccia di comunicazione SPI multislave; si notino le multiple linee SS

Interfaccia SPI multi-slave

la comunicazione avvenga in mutua esclusione con ciascuno dei dispositivi, si sfrutta la linea CS nel modo seguente: essa è posta nello stato *active* solo verso il dispositivo con cui si vuole comunicare e *idle* verso tutti gli altri. L'immagine 3.9 farà senza dubbio chiarezza. Ovviamente lo stato *active* e *idle* sono tensioni TTL opposte. È ovvio dunque come il numero di linee CS del dispositivo master debba essere uguale al numero di dispositivi slave che si vogliono controllare. Completata la comunicazione con il dispositivo, la linea active può ritornare idle anch'essa di modo che se ne possa selezionare un'altra per un'altra trasmissione. Tipicamente si considera la linea CS idle se allo stato TTL alto.

Da menzionare inoltre i moduli Timer e Capture/Compare/PWM: essi non verrano direttamente sfruttati nel progetto, ma il sistema operativo ne farà uso per funzionare, esso stesso.

3.3.3 Interrupt

Risparmiamo al lettore una trattazione su cosa sia il meccanismo di interrupting, sulle politiche di *handling* di quest'ultimo e sulle insidie che la *priority inversion* possa generare.

Nozione fondamentale da conoscere per continuare la lettura, è che il meccanismo dell'interrupt è utilizzato per alterare il normale flusso di esecuzione delle istruzioni, qualora si verifichino delle condizioni straordinare.

Interrupt handling

Le periferiche sopra menzionate ad esempio utilizzano il meccanismo dell'interrupt per comunicare al microcontrollore (e al codice al suo interno) variazioni di stato, ovvero ad esempio la presenza di un byte nel buffer del bus SPI, piuttosto che il completamento dell'invio di un carattere sulla EUSART o l'esaurimento di uno dei timer.

In tutta la famiglia PIC18 è prevista una gestione dell'interrupt organizzata su due distinti livelli di priorità: *high* e *low*. Gli interrupt ad alta priorità, possono interrompere quelli a bassa; il viceversa non è ovviamente possibile.

Nel capitolo 10 di [Mic08b] sono riportati gli SFR atti alla configurazione e gestione degli interrupt.

3.4 Tool di sviluppo e librerie

Ogni lavoro che pertenga allo sviluppo di software è sempre collegato ad alcuni elementi: un linguaggio, un compilatore, un editor ed ad eventuali altre librerie a corredo. Nel seguito sono trattati gli elementi che hanno composto il nostro ambiente di sviluppo.

3.4.1 Microchip® MPLAB IDE

Microchip® MPLAB IDE è la suite per PC che serve a sviluppare applicazioni per microcontrollori Microchip.

Questo programma è un *Integrated Development Environment (IDE)*: esso permette di realizzare tutte le fasi di sviluppo dell'applicazione in una sola applicazione.

È dotato infatti di:

- un editor per la stesura del codice (un po' scandente *ndr.*)

- un project manager per gestire i file sorgente e le impostazioni di compilazione
- si integra perfettamente con tutti i tool di sviluppo di casa Microchip®, ovvero:
  - MPLAB C Compiler (PIC18, PIC24, PIC32, dsPIC)
  - MPASM™Assembler
  - MPLINK™Object Linker
  - MPLIB™Object Librarian
- un simulatore software per il microcontroller target (MPLAB SIM)
- *Watch Window* per monitorare lo stato dei registri del micro
- pemette l'esecuzione *step-by-step* se connesso ad un debugger come l'MPLAB ICD2
- consente di effettuare il flashing del uC in modo molto semplice

Quelle riportate sono solo alcune delle principali caratteristiche di MPLAB IDE. Si vedano i manuali [Mic06a] e [Mic09b] e la guida [Mic05c] per scoprire altre funzionalità, comprendere le potenzialità del simulatore e capire come creare un *Workspace* ove contenere un'applicazione.

3.4.2 Compilatore Microchip® C18

Informazione che certamente il lettore a questo punto necessiterà di conoscere è quella relativa al linguaggio: in che linguaggio si programma il PIC18 ? Le risposte sono due: si può optare per l'assembly e sfruttare l' MPASM™Assembler oppure sviluppare in linguaggio C e utilizzare il compilatore MPLAB®C18. Nel nostro caso si è optato per lo sviluppo in linguaggio C. È oltre lo scopo di questo scritto analizzare le caratteristiche del linguaggio C e tantomento la sua sintassi. Si riportano per completezza alcuni riferimenti a testi la cui lettura è fortemente consigliata prima di intraprendere lo sviluppo. Si vedano la guida introdutti al C18 [Mic05c], il manuale del linguaggio [Mic05b] e la guida alle librerie fornite dal linguaggio [Mic05a]. Si veda ovviamente [Bri88] per una introduzione al linguaggio C.

Unica precisazione doverosa è il fatto che, come si aspetterebbe il lettore più smaliziato, il codice C per i PIC 18 si allontana leggermente da [Bri88] ed è esteso mediante una grande varietà di direttive `#pragma` il cui scopo è fornire al

compilatore delle istruzioni strettamente legate alla configurazione hardware del uC.

Mediante le direttive `#pragma` è possibile ad esempio segnalare al compilatore una ISR (interrupt service routine), descrivere porzioni statiche di dati in memoria programma o impostare i cosiddetti *configuration-bits*. Questi ultimi consentono di gestire le modalità di funzionamento del uC, come ad esempio la tipologia e velocità di clock, l'utilizzo di talune porte del microcontrollore e altre periferiche come il *WatchDog Timer*. Si vedano [Mic07b] e [Mic09c] per la lista completa dei configuration bits e per esempi di utilizzo delle periferiche onboard mediante le librerie del linguaggio.

3.4.3 Software applicativo Microchip®

Per comprendere il funzionamento della radio e per agevolare lo sviluppo si sono sfruttate alcune applicazioni fornite direttamente da Microchip®. Ciascuna di esse è corredata di una cosiddetta *application note* ovvero una relazione in cui vengono illustrati per sommi capi i principi di funzionamento del software.

Microchip Stack for the ZigBee™Protocol

Descritto nell'application note AN965 (si veda [FOR06]), tale software compone uno stack ZigBee™completo; Non ci si dilunga ulteriormente poichè tale stack è stato oggetto di numerosi altri lavori, fra i qualli ricordiamo il lavoro [Ber08].

MRF24J40 Radio Utility Driver Program

È un semplice driver *raw* che mostra tecniche di interfacciamento e utilizzo del transceiver a noi in dotazione. Quest'ultimo documento è stato molto utile per comprendere le dinamiche di comunicazione mediante bus SPI tra il microcontrollore e il transceiver. Il programma a corredo permette all'utente di familiarizzare inoltre con le modalità di funzionamento della radio e permette una più facile comprensione degli elementi presentati nel datasheet del transceiver.

Si veda l'application note [MM09] per maggiori informazioni.

Capitolo 4

FreeRTOS

In questo capitolo si accennano rapidamente alcune caratteristiche di FreeRTOS che consentono di far capire al lettore perchè si è optato per esso e che benefici si hanno di conseguenza.

4.1 Cos'è

FreeRTOS è un sistema operativo real-time, comunemente in inglese RTOS, ovvero *real-time operating system*. Esso è progettato dal suo autore (*Richard Barry*) con l'ambizione di essere semplice, portabile e conciso.

Real Time Operating System

È scritto in C, con le porzioni più delicate in codice assembly. Al momento della stesura di questo documento, esso supporta 28 architetture differenti (quasi tutti microcontrollori e dispositivi embedded), con compilatori diversi per ciascuna architettura ed è rilasciato corredato di più di 50 applicazioni di demo differenti. Ciascuna di esse ha come target un'architettura, un dato compilatore e una scheda di sviluppo ed è sovente identificata con il nome di *porting*

Multipiattaforma

Interessante ai fini del nostro progetto è il fatto che di casa Microchip® sono supportate tutte le famiglie dalla *middle-range* in su, ovvero:

- PIC32
- PIC24 e dsPIC
- PIC18

Le applicazioni demo sono inoltre testate sui compilatori ufficiali Microchip®.

L'applicazione demo di nostro interesse nello specifico è scritta per il PIC18F452 su di una scheda della Forest Electronic Developments. Tale fatto ha richiesto un lavoro preliminare di adattamento per il PIC18F4620 sulla PidemZ che verrà esaminato più oltre.

FreeRTOS è innanzitutto un sistema operativo: è perciò dotato di un kernel in grado di mandare in esecuzione diversi task fornendo funzionalità di *multitasking*.

Un processore convenzionale (non *multicore*) non può eseguire più di una istruzione alla volta, ma eseguendo in successione istruzioni provenienti da diversi programmi, fornisce la sensazione che più di un programma alla volta sia in esecuzione.

Multitasking

Il RTOS fornisce pertanto delle API che consentono allo sviluppatore di creare task (dei veri e propri programmi *stand-alone*), mandarli in esecuzione e rimuoverli dal sistema. Quando un task è in esecuzione esso ha a disposizione una copia delle proprie variabili e nessuno dei task sospesi può invadere (e manomettere) le variabili del task.

Essendo inoltre real-time, FreeRTOS ci fornisce anche diverse politiche di scheduling atte a stabilire quando ciascuno dei programmi (o task) andrà in esecuzione e per quanto.

Oltre alle funzionalità di base appena descritte, vengono fornite allo sviluppatore altre feature in modo da rendere facilmente praticabili operazioni quali la comunicazione inter-task o l'isolamento di porzioni di codice che accedano a risorse ad accesso limitato.

Per una introduzione alla teoria dei sistemi operativi si veda [CFM11].

4.2 Come funziona

FreeRTOS è diviso in due porzioni principali: la prima contiene l'implementazione delle API fornite allo sviluppatore per utilizzare il sistema operativo e le sue funzionalità accessorie. Tutto il codice che lo compone è comune a tutte le demo e costituisce il *core* di FreeRTOS

La seconda parte invece, è specifica per ogni architettura. Tale sezione costituisce un layer di adattamento che nasconde al core le peculiarità dell'architettura sulla quale in quel momento è in esecuzione il sistema operativo.

Questa scelta del progettista si è rilevata piuttosto azzeccata poichè consente di aumentare le funzionalità del sistema incrementalmente e per giunta rende possibile lo sviluppo di nuovi *porting* in maniera relativamente semplice ed elegante.

API essenziali

Nel seguito vengono menzionati solo aluni degli elementi essenziali del RTOS. Si consiglia vivamente, per avere una panoramica più ampia del sistema, la lettura dell'ottimo [Bar09b].

Le API sono suddivise in diversi file e hanno diversi ambiti applicativi; esse consentono di:

- gestire i task (crearli, rimuoverli, variarne la priorità, ecc...)

- avviare lo scheduler del sistema operativo
- gestire le code di comunicazione fra task (leggere e scrivere messaggi sulla coda)
- gestire semafori binari e semafori con counter
- gestire coroutine (una versione *light* dei task a memoria condivisa)
- creare mutex per la sincronizzazione
- utilizzare un heap per l'allocazione dinamica della memoria

Esaminiamo innanzitutto le funzionalità inerenti i task. Innanzitutto un task è semplicemente che una funzione C. Quest'ultima però non viene invocata direttamente nel codice, ma viene passata mediante un puntatore a funzione ad un'altra funzione delle API che si occupa di trasformarla in un task. Tale funzione è `xTaskCreate(....)`. Si veda la documentazione delle API di FreeRTOS ([Bar09a]) per esempi di utilizzo delle funzioni e per capirne i parametri di ingresso e i valori di ritorno.

Essa sarà poi mandata in esecuzione dallo scheduler di FreeRTOS non appena si verifichino le condizioni idonee.

La funzione che si occupa invece dell'avvio dello scheduler del sistema operativo è `vTaskStartScheduler()`. Dopo l'invocazione di questa funzione, il sistema operativo FreeRTOS è avviato: i task varieranno il loro stato a seconda delle decisioni dello scheduler o mediante esplicita esecuzione delle chiamate riportate in figura 4.1.

FreeRTOS scheduler

Un task pronto per l'esecuzione è quindi in stato *ready* e diventa *running* solamente quando l'algoritmo di scheduling lo ritiene opportuno. FreeRTOS supporta due diverse tipologie di scheduling: *round-robin* cooperativo e *preemtive*. Nel primo caso, viene assegnato ad ogni task un numero fissato di quanti temporali durante i quali un task è fisicamente in esecuzione; trascorsi questi, la CPU viene ceduta ad un altro task ready. Qualora non vi siano altri processi ready la CPU viene passata ad un task particolare, sempre presente dopo l'avvio dello scheduler: il task idle. Come si immaginerà, il task *idle* non fa nulla e cede il controllo al prossimo task che si dichiarerà ready.

Con la modalità di schedule *preemptive* un task può sottrarre il controllo della cpu qualora si verifichi una gerarchia di priorità fra task che lo consenta. Tale meccanismo è noto in letteratura come *preemption*.

Essendo ben oltre lo scopo di questo lavoro affrontare il complesso problema dello scheduling di task, e ancor peggio nel caso di scheduling real-time, si consiglia per approfondire il tema, la consultazione di [Liu00].

Context-switch

Il meccanismo di funzionamento di FreeRTOS è concettualmente piuttosto

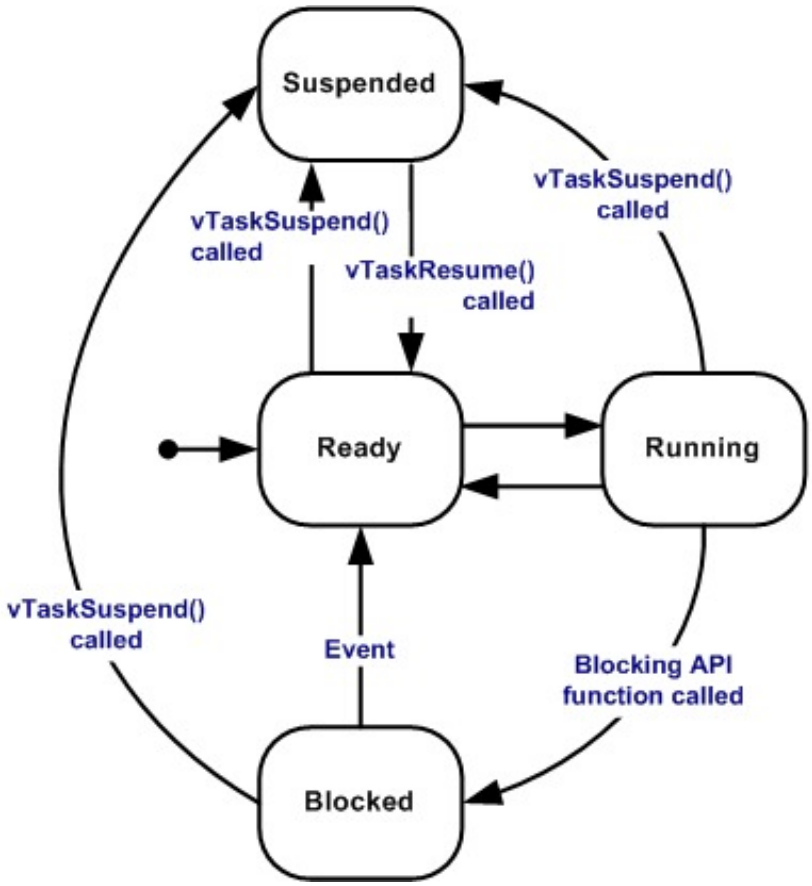

Figura 4.1: Gli stati possibili di un task FreeRTOS e funzioni per la variazione di stato

semplice. Esso si basa sostanzialmente sull'utilizzo di un timer hardware. Allo scadere di questo, viene lanciato un interrupt e questo fatto consente la messa in esecuzione dell'opportuna *interrupt service routine* (ISR) che lo gestisce.

La routine reimposta il timer in modo da poter far verificare di nuovo l'interrupt e avvia poi il kernel del sistema operativo. Il kernel analizza poi lo stato dei task del sistema.

Se viene verificata la necessità di passare la CPU ad un task differente rispetto a quello trovato in esecuzione, il kernel avvia la procedura di *context-switch*. Un context-switch consiste nel copiare in una porzione di memoria *riservata* al sistema operativo il valore delle dei registri dello stack del task in esecuzione. Insieme con la copia delle variabili vengono memorizzate anche le informazioni relative al program counter (PC) che è una locazione di memoria in cui è presente l'indirizzo dell'istruzione correntemente in elaborazione. Tale operazione è nota

con il nome di *context-save*.

Selezionato il nuovo task da mandare in esecuzione, se ne recupera lo stato precedentemente archiviato, lo si ripristina all'interno dello stack e lo si rimanda in esecuzione impostando opportunamente il PC. Questa fase è il *context-restore*.

L'interrupt del timer che permette questa sequenza di operazioni è noto all'interno di FreeRTOS con il nome di *tick-interrupt*. Il quanto temporale minimo all'interno di FreeRTOS è pertanto la distanza fra due tick successivi. La frequenza del tick è inoltre regolabile mediante i file di configurazione del sistema operativo.

4.3 Feature interessanti

Oltre al fondamentale contex-switch fra task, FreeRTOS mette a disposizione un ben nutrito gruppo di funzioni per gestire funzionalità accessorie che si rendono però assai utili, nello sviluppo di applicazioni non banali.

Code FIFO

Di grande interesse sono senza dubbio le code. Con queste ultime è possibile definire un canale di comunicazione con politica FIFO. In una coda è possibile inserire un qualunque tipo di dato definibile in C, anche un tipo complesso costruito mediante `struct`. Da notare che tutti gli oggetti nella coda devono essere dello stesso tipo.

Nella sostanza, un qualunque task può creare una coda mediante la chiamata `xQueueCreate(..)`.

Si supponga che la coda creata da un primo task supporti il tipo di dato `unsigned char`. Dopo la creazione di tale coda, un altro task può inserire ad esempio il `char 'a'` nella coda creata dal primo task mediante la primitiva `xQueueSend(..)`. Un terzo task poi potrà leggere che nella coda è stato inserito il carattere `'a'` mediante la funzione `xQueueReceive(..)`.

Ecco dunque come è possibile dotare i task della possibilità di comunicare fra loro. Tale funzionalità si è resa molto utile nel proseguio di questo lavoro, come verrà riportato estensivamente nel seguito.

Capitolo 5

Design dell'implementazione

In questo capitolo vengono trattate le problematiche evinte che hanno condotto alla struttura finale del codice e alla relativa implementazione. Si ricapitolano brevemente gli ingredienti fin quì presentati e si spiega con quali finalità e principi è stato progettato il software. Dopodichè si entra nel dettaglio dell'architettura software di NCAP e TIM

5.1 La struttura complessiva

Come si avrà ampliamente avuto modo di capire dalla trattazione dei capitoli precedenti, è necessario riflettere bene sulle caratteristiche che l'implementazione deve offrire *prima di passare all'azione*.

Libreria e non applicazione

Va inoltre precisato che si è cercato, ove possibile, di optare sempre per scelte che non fossero troppo limitative dal punto di vista del riutilizzo del codice, in modo da progettare quantomeno l'ossatura di quella che potrebbe diventare a tutti gli effetti una libreria per FreeRTOS che implementi (magari per ora parzialmente) lo standard 1451. Tale scelta, sebbene ammirabile, se non altro nella finalità, ha generato come controparte, ulteriori complicazione progettuali, che si spera trovino nel design prodotto una degna collocazione.

Nella trattazione che segue viene fornita una rapidissima panoramica delle caratteristiche salienti delle due fasce di dispositivi. Ulteriore raffinamento e più dettagliata spiegazione, segue.

5.1.1 I nodi

Ricapitolando brevemente lo scenario fin quì presentato, facciamo mente locale di tutti gli elementi che dobbiamo mettere insieme. Abbiamo il protocollo 1451.0

(si veda 2.2) che si colloca immediatamente sopra alle routine di servizio fornite dal protocollo 1451.5. Esse a loro volta utilizzano (o incapsulano) primitive o chiamate fornite da un livello più basso, che pilota direttamente l'hardware a disposizione.

Il 1451.0 si divide in due rami completamente distinti, l'implementazione per il nodo NCAP e quello per il nodo TIM.

1451.0 NCAP querying

Nel primo caso le chiamate che è necessario rendere disponibili al programmatore sono in effetti quelle presentate nella sezione 2.3.2 del capitolo sul 1451. Tali primitive sono principalmente chiamate di *querying* verso i nodi TIM e i relativi canali di trasduzione, nonchè alcune chiamate che permettano di supportare l'invocazione di callback alla conclusione dell'acquisizione dei dati lato TIM. Il nodo NCAP deve avere a bordo del codice che consenta quindi al designer di un'applicazione che fruisce dello standard 1451 di invocare le chiamate previste dallo stesso, con relativi parametri (di ingresso e di uscita) e eventualmente di *callback*.

1451.0 TIM replying

Per quanto riguarda il nodo TIM invece è necessario implementare la controparte delle pritive lato NCAP, ovvero è necessario dotare il nodo della capacità di *replying* alle richieste NCAP. Il nodo TIM deve essere quindi equipaggiato di codice in grado di invocare gli strati inferiori dello standard 1451, ovvero il 1451.5 nel nostro caso che a sua volta chiamerà librerie più vicine all'hardware per l'effettivo invio dei messaggi attraverso la rete.

Il codice deve consentire di incorporare librerie esterne (ovviamente progettate e scritte con opportune specifiche di compatibilità) fornite da uno sviluppatore che fungano da driver per *qualunque* tipo di sensore. Tali librerie infine devono funzionare in modo da essere indipendenti dall'effettivo mezzo di comunicazione usato per connettere il nodo TIM con la rete cui appartiene e deve fornire la possibilità di stoccare le informazioni dei TEDS su "supporti" differenti.

5.1.2 Hardware e software

Ovviamente, come sempre succede in questi casi, soddisfare completamente le specifiche e costruire del buon codice, comporta fare una buona modellizzazione degli oggetti coinvolti nella realtà analizzata e tradurli opportunamente in codice. Tale scopo è *ottenibile più facilmente* quando si ha a disposizione hardware e software di appoggio molto diverso da quello a noi in dotazione.

x86: Sistema operativo e toolchain

Disporre infatti di un sistema operativo (così come l'accezione comune del termine prevede) comporta avere a disposizione un strato software che fornisce trasparentemente allo sviluppatore una moltitudine di facilitazioni. Innanzitutto il meccanismo di multitasking e relativa politica di scheduling, eventualmente real-time. Seguono poi la gestione della memoria dei relativi programmi in esecuzione senza che si verifichino conflitti o sovrapposizioni fra le porzioni di ciascuno e

soprattutto la possibilità, non avendo grossi limiti dal punto di vista dell'hardware, di sviluppare codice con astratti e sofisticati linguaggi orientati agli oggetti che rendono il *modelling* della realtà, una vera passeggiata.

Picdem Z: PIC18, FreeRTOS, MPLAB e C18

Noi per converso abbiamo uno scenario operativo molto differente: (si veda la panoramica dell'hardware del PIC18F4620 per maggiore precisione)

- scarsissima capacità di processing
- scarsa memoria programma
- memoria RAM molto piccola
- memoria *non-volatile* di ridotte dimensioni
- bassa espressività (capacità di rappresentare concetti astratti) della *toolchain*
- difficoltà di debugging

Si badi bene che questa carrellata non è assolutamente atta a scoraggiare chiunque si voglia avvicinare a questo tipo di ambiente di sviluppo, quanto piuttosto a valutare con obbiettività cosa ci si trova davanti, in modo da sfruttare al meglio le risorse e progettare il codice di conseguenza.

Embedded network: scenario perfetto per il 1451

È doveroso inoltre precisare come una rete 1451 sarà quasi certamente costituita da molti nodi embedded; da qui la necessità di realizzare una implementazione in un hardware come quello del PIC18 dove sono presenti parecchie limitazioni.

La controparte infatti, come si si può facilmente immaginare, è la maneggevolezza che una scheda del genere (e relativo uController) offre:

- controllo *totale* di cosa viene eseguito e quando
- ampie possibilità nella scelta di cosa implementare e come
- ottimizzazione del codice (ambiente di sviluppo e compilazione dedicato)
- ridottissimi costi di produzione dell'hardware (specie su larga scala)

5.1.3 Perchè FreeRTOS

Questa panoramica dovrebbe motivare almento in parte, perchè si è deciso di appoggiare l'implementazione della libreria 1451.0 su FreeRTOS.

Lo scenario che ci troviamo davanti coinvolge un complesso sistema di livelli software, ciascuno con compiti e dipendenze non banali. È necessario permettere al software di generare o gestire messaggi compatibili con lo standard 1451,

pilotare l'hardware della PicdemZ in modo da utilizzare il transceiver radio, comprendere la varie tipologie di comandi ricevuti e disporre le opportune azioni per fornire una replica, gestire l'acquisizione o l'attuazione di data set, ecc...

Complesso sistema a *layer*

Avendo pertanto un problema non banale davanti si è ritenuto saggio fruire di uno strato software posto al livello più basso che desse uniformità a tutto il codice prodotto e garantissime un minimo di caratteristiche per rendere più agevole lo sviluppo oltre a garantirne una minima portabilità.

Portabilità

Usare FreeRTOS significa avere (si veda 4.1) a disposizione alcune pritive per creare task e metterli in esecuzione concorrentemente secondo la politica di scheduling scelta, separando il meccanismo dalla politica di utilizzo dello stesso. Significa inoltre avere a disposizione una struttura centrale per l'assegnazione della memoria ai vari task. Abbiamo inoltre a disposizione alcune librerie che consentono la comunicazione fra task mediante code di messaggi e semplici meccanismi di *lock* per l'accesso in mutua esclusione alle risorse, in porzioni delicate del software.

Vantaggi di FreeRTOS

Altro motivo per il quale si è deciso di appoggiarsi ad un livello software inferiore che funga da piattaforma a tutta la libreria è il problema degli *automi mutuamenti interagenti*.

Il problema degli automi mutuamente interagenti

Lo standard 1451 è rappresentabile (magari con qualche rilassamento sulla definizione teorica tradizionale di [HMU07]) mediante un' automa a stati finiti deterministico o DFA *(deterministic finite automaton)*. Ciò significa che è possibile redigere un diagramma del tipo in figura 5.1 che illustra in maniera visuale in che stato la macchina si trova e in quali stati possa evolvere a seconda delle sollecitazioni esterne.

Il concetto di automa a stati finiti

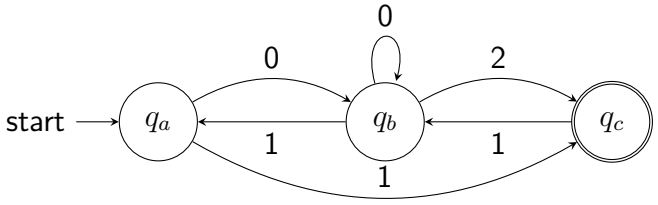

Figura 5.1: Esempio di un DFA: i nodi rappresentano gli stati possibili e gli archi rappresentano l'evento che fa variare lo stato all'automa

Più semplicemente, possiamo immaginare che ciascuno stato rappresenti una condizione possibile, assumibile dal protocollo (o più correttamente da un nodo TIM che implementi lo standard IEEE 1451.0), ad esempio:

il 1451 visto come un automa

- Ricevuto comando di classe `x` e identificativo `y` (in breve <x,y>)

- Inviata risposta a comando <s,t>
- Mi trovo in modalità TIMSleep
- Il TrCh-3 si trova in modalità FreeRunning

e ciascun arco (o freccia) nel diagramma rappresenta le possibili evoluzioni fra stati, ad esempio:

- Sono in stato TIMSleep e mediante il comando che mi manda in stato TIMSleep rimango in stato TIMSleep
- Sono in stato TIMActive e mediante il comando che mi manda in stato TIMSleep evolvo verso lo stato TIMSleep
- Il TrCh-3 da FreeRunning può diventare Sampling mediante il comando opportuno

Automa della radio

Il protocollo di comunicazione sfruttato dallo strato 1451.5, sia che ci si voglia limitare all'802.15.4 e a maggior ragione qualora si desideri utilizzare l'intero Stack Zigbee Microchip, è a sua volta un altro automa a stati finiti e per giunta assai complesso.

automa 1451 $\Longleftrightarrow$ automa radio

Il problema consiste nel fatto che tali automi che richiedono entrambi di essere rappresentati nel funzionamento del codice, devono potersi sollecitare a vicenda. Un esempio può chiarire immediatamente quale sia il nocciolo della questione. Se il codice del 1451 di un nodo TIM deve inviare un messaggio di *reply* ad un nodo NCAP alla fine dell'acquisizione di un *set* di dati, esso deve poter comunicare con il *driver* radio in modo che esso si ponga in modalità di trasmissione e poi proceda all'invio fisico dei dati sul canale e segnali poi al 1451 il successo della trasmissione. Al contrario invece, immaginando il nodo TIM impegnato ad acquisire dati da un TrCh e immaginando poi che un nodo NCAP invii al nodo TIM impegnato, una richiesta di arresto dell'acquisizione, è facile immaginare come il *driver* radio debba ricevere dal canale di trasmissione il messaggio e informare il 1451 della ricezione del messaggio, provvedendo quanto prima ad interrompere l'acquisizione.

Lo scenario presentato, dimostra come le due entità siano entità che devono essere trattate come entità pari, senza dipendenze relative, dotate però di un canale di comunicazione software che consenta la mutua sollecitazione. Tale canale fra gli automi sarà facilmente ottenibile grazie alle primitive di comunicazione fra task fornite nativamente da FreeRTOS.

5.2 Design lato NCAP

Entrando nel merito dell'architettura del software sul versante NCAP, ci si è basati sui capitoli 9, 10 ed 11 di [14507a] per redarre l'analisi seguente. Da precisare inoltre, che per avere una migliore comprensione del problema presentato, è bene procedere alla lettura anche di [14507b].

Goal

Nei capitoli menzionati vengono presentate le API che un'implementazione lato NCAP dovrebbe fornire allo sviluppatore che volesse scrivere un'applicazione basata sul 1451. Le API fornite, espresse in IDL, sono suddivise in moduli in modo da suddividere in porzioni semanticamente coerenti le funzioni da esse raggruppate.

Componenti sotware dello standard

Le componenti costituenti le API sono visibili in figura 5.2 Nello schema

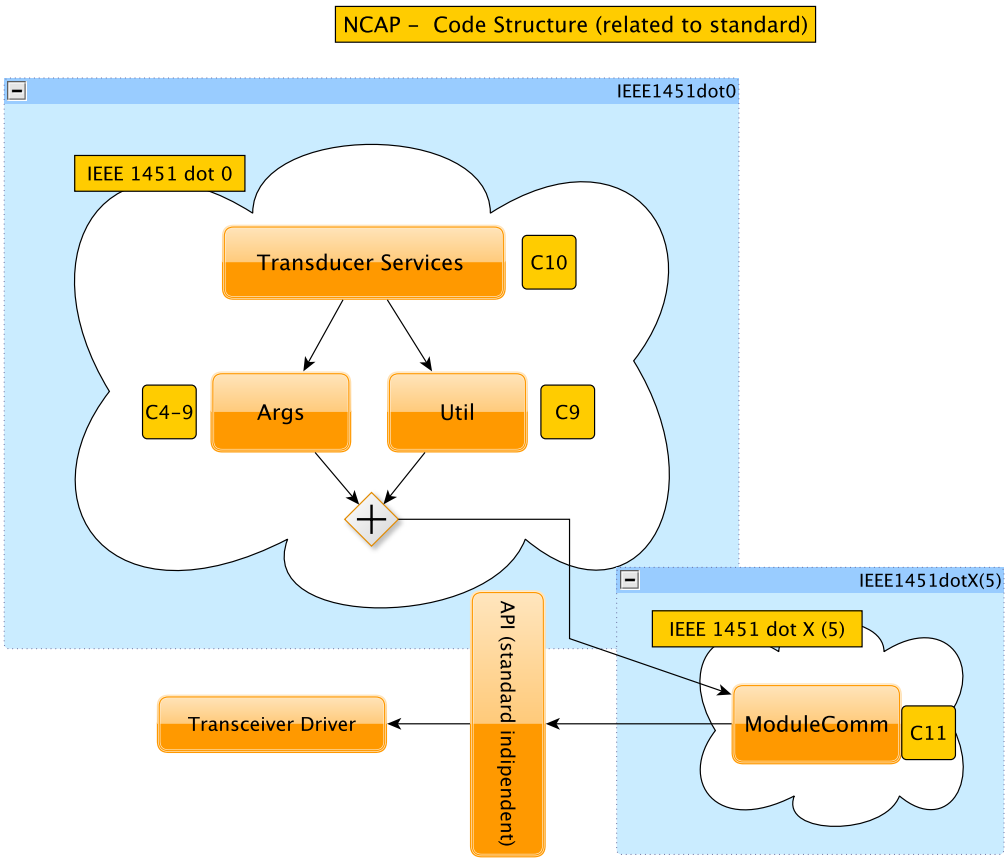

Figura 5.2: Componenti principali dello standard 1451

sono rappresentati in arancio i moduli dello standard e la relativa appartenenza

ai diversi layer dello standard. Accanto a ciascun modulo è presente un'etichetta nella forma `CXX` dove `XX` funge da riferimento ai capitoli di [14507a] dove ciascuna porzione è descritta in dettaglio.

I moduli facenti parte di *IEEE1451dot0* sono

- `Transducer services`
- `Args`
- `Util`

I moduli sw 1451.0

Ciascuno di essi ha un ruolo preciso nella trattazione proposta dall' IEEE. I `Transducer services` sono l'elemento più importante, essi infatti costituiscono l'insieme di funzioni deputate all'interfacciamento remoto coi nodi TIM. Questa la definizione fornita all'interno dello standard:

> *This is the public API that measurement and control applications use to interact with the IEEE 1451.0 layer. It contains classes and interfaces for discovering registered TIMs, accessing TransducerChannels to make measurements or write actuators, managing TIM access, and reading and writing TEDS.*

I moduli `Args` e `Util` contengono funzionalità ausiliarie ai `Transducer Services`. `Args` contiene le definizioni dei tipi di dati supportati dal protocollo: dai più semplici numeri interi fino ad elaborate strutture composite atte a rappresentare ogni tipo di grandezza fisica. Si veda il capitolo 4 di [14507a] per ulteriori informazioni. Il modulo `Util` contiene invece alcune primitive atte ad effettuare conversioni e manipolazioni di *stream* di `Args`, in modo da poter effettuare il *marshalling/unmarshalling* dei dati per la trasmissione. A tal proposito si vedano le tabelle 81 e 82 di [14507a].

Il modulo di comunicazione

Il modulo `ModuleComm` appartiene ad un layer diverso dello standard. Esso infatti dipende dalla tipologia di connessione presente fra il nodo NCAP e i TIM. Nel nostro scenario operativo lo strato di comunicazione è basato sul layer 1451.5, adatto alle trasmissioni su rete wireless. Il modulo contiene le primitive che consentono allo strato 1451.0 di interfacciarsi al modulo di comunicazione implementato nello strato 1451.5 che a sua volta, come si capisce facilmente dalla figura 5.2 si interfaccerà ai *driver* per il controllo del transceiver della scheda.

Per maggiore precisione espositiva, si riportano in figura 5.3 gli elementi che costituiscono una rappresentazione visuale di moduli e interfacce descritte all'interno dello standard.

Accanto a ciascun modulo è riportata un'etichetta nella forma `TXX` dove `XX` rappresente il riferimento alla tabella di [14507a] che contiene il dettaglio delle primitive che dovrebbero essere presenti nell'implementazione di ciascun modulo.

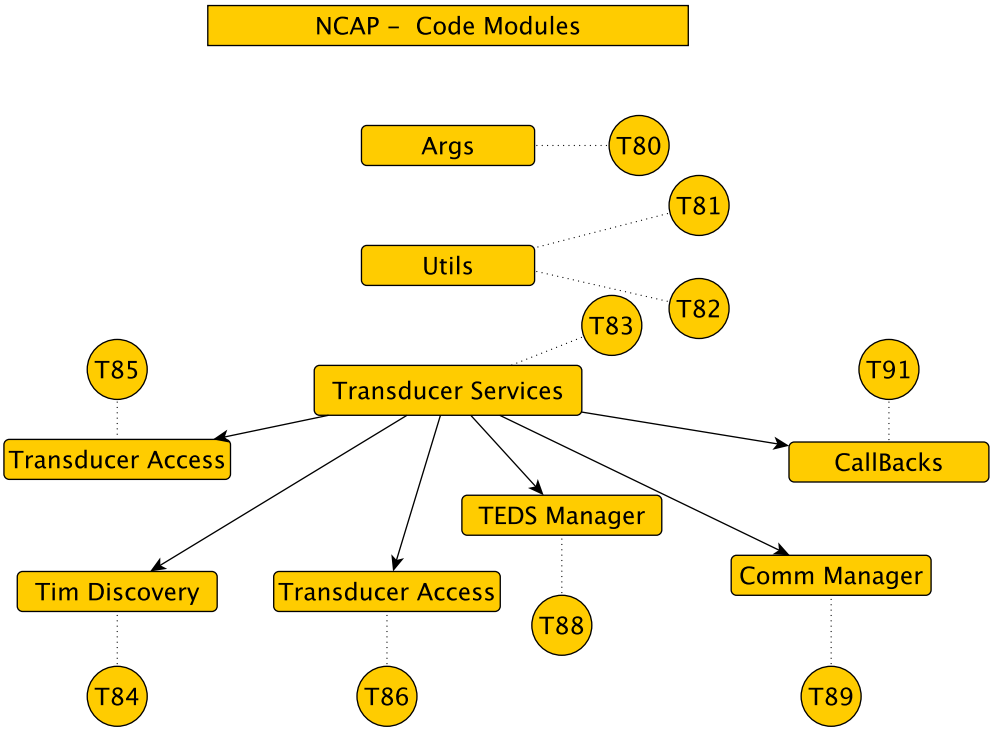

Figura 5.3: `Core modules` dello standard 1451.0 e relative `interfaces`

5.2.1 Utilizzo delle API

In figura 5.4 è rappresentato un diagramma a blocchi che illustra come dovrebbero funzionare le API del 1451.

Primitive per il designer di un'applicazione 1451

Un'applicazione che sfrutti le API del 1451 non farebbe null'altro che invocare una delle possibili primitive precedentemente presentate con gli opportuni parametri passati. Tale fatto causerebbe la creazione di un messaggio 1451 ed il relativo invio verso il nodo target o verso uno specifico canale di un dato nodo.

In caso di chiamata di tipo bloccante, gli argomenti di ritorno della primitiva verranno popolati e restituiti all'applicazione chiamante, non appena il comando verrà processato ed eseguito dal destinatario e ne sarà prodotta la risposta. In caso di chiamata non bloccante invece, la chiamata ritornerà non appena inviato il messaggio verso l'indirizzo desiderato: gli argomenti di output verranno popolati e resi disponibile all'applicazione all'interno di una chiamata di callback invocata direttamente dal 1451 non appena ricevuta risposta dalla destinazione richiesta.

I moduli nei layer di appartenenza

È possibile capire l'interazione delle varie componenti in relazione al loro strato di appartenenza nella figura 5.5. L'implementazione, per avere una certa

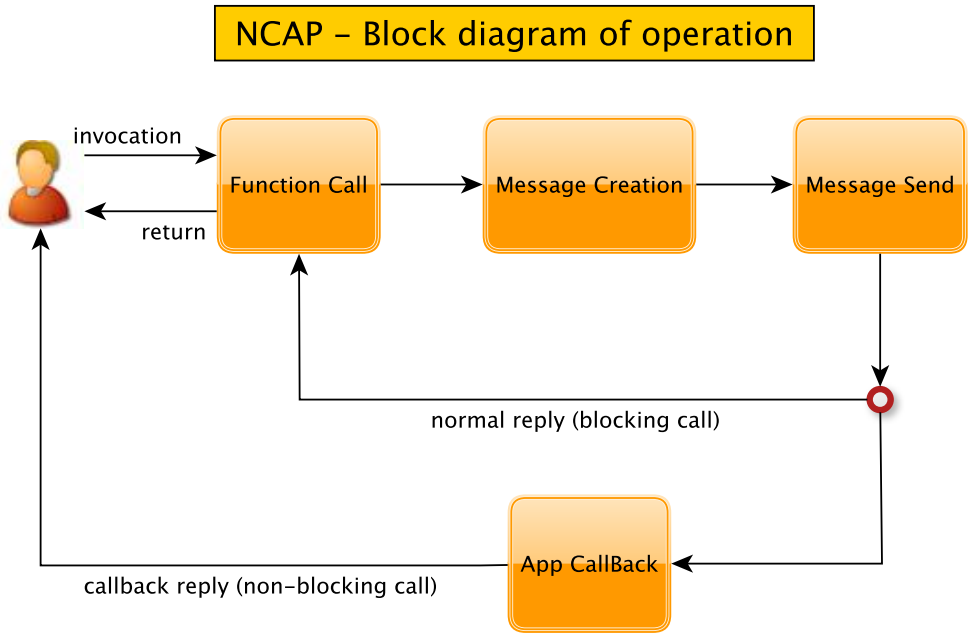

Figura 5.4: Scenario operativo di utilizzo delle API del 1451.0 (lato NCAP)

organicità e robustezza, dovrebbe essere struttura in livelli distinti in modo da inserire all'interno dello strato 1451.0 la realizzazione dei moduli `Transducer services`, `Args eUtil`. Su un layer distinto si dovrebbe trovare il `ModuleComm` che è il componente *core* dello strato intermedio fra l'hardware fisico della scheda e i servizi 1451.0.

Modularità del codice

Tale suddivisione è assolutamente necessaria se si è interessati a realizzare una libreria modulare. Per modularità, intendiamo riferirci al fatto che porzioni del codice possano essere rimpiazzate singolarmente senza compromettere la struttura complessiva dello stesso. Fatto sembra di grande rilevanza poichè il componente che giace sul layer 1451.5 dovrebbe poter esser rimpiazzato con strati di comunicazione diversi a seconda dello strato fisico a disposizione.

5.2.2 Implementazione corrente

Al momento della stesura di questa tesi l'implementazione secondo questo design sul nodo NCAP è ancora mancante. Si è però implementato il codice per un nodo che simula le abilità (in versione assai ridotta) di un nodo NCAP in modo da poter collaudare le funzionalità fornite dal nodo TIM. Tale nodo verrà descritto nel capitolo 8.

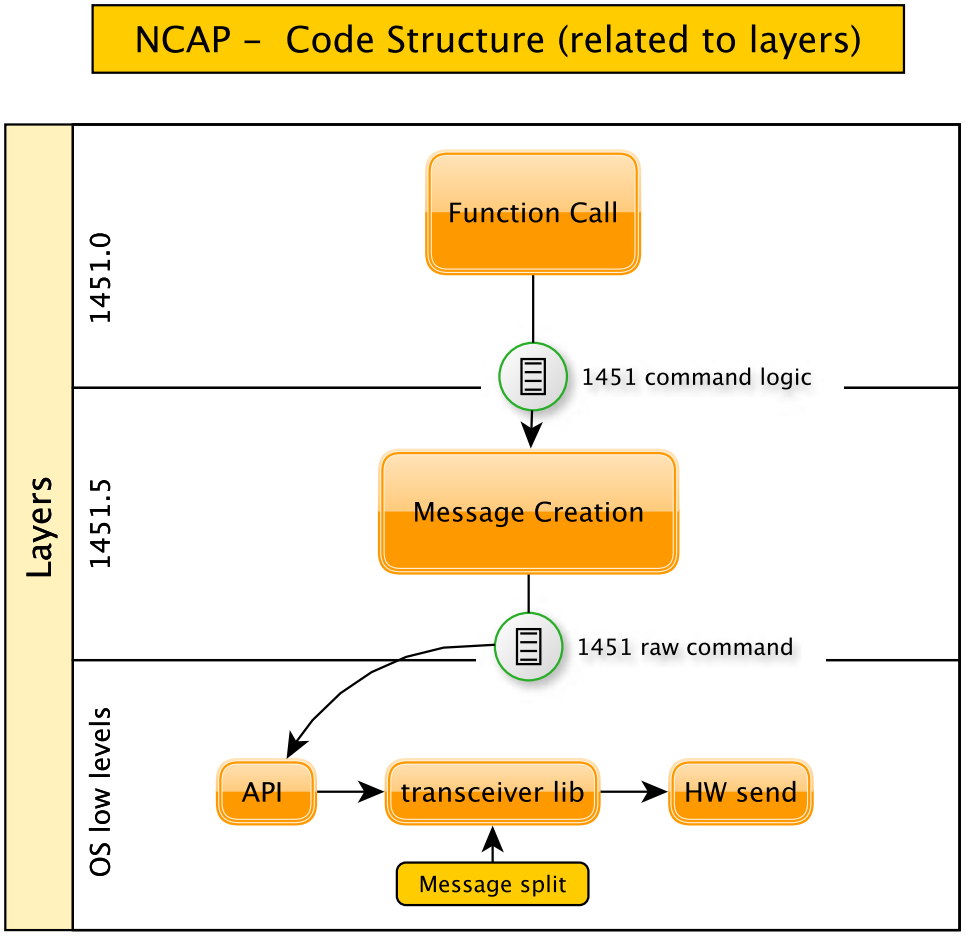

Figura 5.5: Diagramma delle operazione 1451 NCAP - prospettiva layer

5.3 Design lato TIM

Siamo finalmente alla progettazione del software sul nodo TIM che è in effetti il vero e proprio risultato operativo di questo lavoro.

Possiamo supporre che il nodo NCAP sia già dotato della capacità di inviare richieste ai nodi TIM e gestirne le risposte ricevute. Ora dobbiamo capire come realizzare il software del TIM.

In esso si concentrano tutte le problematiche più complesse del protocollo. A differenza del nodo NCAP dove abbiamo dato una rapida overview degli elementi costitutivi di una possibile implementazione, nel caso del TIM entreremo in ogni singolo dettaglio in modo da permettere al lettore di capirne la struttura e la

conseguente implementazione, illustrata nel prossimo capitolo.

Il processo di design

Come è sovente prassi quando si progetta software, cerchiamo di giungere al design conclusivo mediante una sequenza di passi ben ponderati. Cominciamo con l'elencare alcuni scenari di utilizzo (casi d'uso) del TIM, cerchiamo poi di evincere dei requisiti e in seguito concludiamo con la sintesi del design.

5.3.1 Casi d'uso

Riportiamo di seguito alcuni casi d'uso di un ipotetico nodo TIM. Ciascuno scenario presentato è un frammento temporale di eventi immaginari che ci aiutano ad ipotizzare meglio la realtà finale del nodo. Gli eventi sono ipotizzati ovviamente considerando le API e lo scenario presentato in [14507a]

scenario 1

Condizioni iniziali:

- TIM `T` in modalità `active`
- `TrCh-2` in modalità `active`
- radio in modalità `rx`
- indirizzo e numero canali noti ad un nodo NCAP `N`

Sequenza eventi:

- NCAP invia al tempo t_0 segnale di trigger per attivare acquisizioneTrCh-2
- TIM riceve messaggio, elabora e comprende il comando (per brevità `TIM_ACK`)
- TIM inizia l'acquisizione dei campioni dal Transducer Channel
- TIM invia conferma al nodo NCAP
- NCAP invia dopo un tempo t_1 richiesta invio dati
- TIM riceve messaggio, elabora e comprende il comando
- TIM invia i dati acquisiti nell'intervallo $[t_0, t_1]$
- NCAP riceve dati e dispone lo stop del canale del TIM
- `TIM_ACK` e TIM pone in stato IDLE il canale

scenario 2

Condizioni iniziali:

- TIM `T` in modalità `active`
- `TrCh-xx` in modalità `sleep`
- radio in modalità `rx`
- indirizzo e numero canali noti ad un nodo NCAP `N`

Sequenza eventi:

- NCAP richiede attivazione di `TrCh-0`
- `TIM_ACK` e variazione stato di `TrCh-0`
- NCAP invia trigger per `TrCh-0`
- `TIM_ACK` e inizio acquisizione da `TrCh-0`
- NCAP richiede `sleep` di `TrCh-0`
- `TIM_ACK` e messa in `sleep` di `TrCh-0`

scenario 3

Condizioni iniziali:

- scenario precedente

Sequenza eventi:

- NCAP richiede attivazione di `TrCh-0`
- `TIM_ACK` e variazione stato di `TrCh-0`
- NCAP invia trigger per `TrCh-0`
- `TIM_ACK` e inizio acquisizione da `TrCh-0`
- NCAP richiede data-set di `TrCh-0` in modalità asincrona (non bloccante)
- `TIM_ACK` e invio data set e `reference` alla richiesta del nodo NCAP

scenario 4

Condizioni iniziali:

- scenario precedente

Sequenza eventi:

- NCAP richiede attivazione di `TrCh-0`
- `TIM_ACK` e variazione stato di `TrCh-0`
- NCAP richiede `MetaTEDS` al nodo TIM

- `TIM_ACK`, *fetching* e invio del TEDS specifico a NCAP
- NCAP invia trigger per `TrCh-0`
- `TIM_ACK` e inizio acquisizione da `TrCh-0`
- NCAP richiede data-set di taglia n di `TrCh-0` in modalità asincrona (non bloccante)
- `TIM_ACK`
- NCAP richiede nuovamente `MetaTEDS` al nodo TIM
- `TIM_ACK`, *fetching* e invio del TEDS specifico a NCAP (c'è una richiesta non bloccante ancora *pending*)
- invio data set pronto (con n campioni) e `reference` alla richiesta (non bloccante) precentemente già confermata del nodo NCAP

Le situazioni presentate non alcuna presunzione di completezza ma servono solamente per fornire al lettore alcune immagini per capire lo scenario operativo di un nodo TIM al lavoro. Riflettendo su quanto detto è comunque possibile evincere le funzionalità principali del TIM.

5.3.2 Specifiche

Caratteristiche essenziali TIM

Grazie ai semplici scenari presentati, è possibile capire immediatamente alcune delle caratteristiche *fondamentali* del nodo TIM. Esso deve:

- *idealmente* poter ricevere messaggi in qualunque momento
- inviare messaggi in momenti precisi
- capire i comandi ricevuti
- intraprendere opportune operazioni pratiche secondo comando ricevuto
- suddividere comandi e operazioni in base ai diversi ambiti operativi previsti (acquisizione, alterazione stato interno, processing `TEDS`, ecc...)
- sapere in che *stato* si trova e in quale stato si trovano i propri *Transducer Channel*
- poter modificare il proprio stato in base ai comandi ricevuti
- poter avviare azioni e eventualmente stopparle in modo non *naturale*
- gestire coppie di messaggi ricevuti/inviati secondo uno schema temporale non *naive*

Leggendo con attenzione gli scenari possiamo dedurre alcuni vincoli software che prima non erano noti. Uno scenario infatti è composto di operazioni atomiche elementari (che concettualmente sappiamo già di dover supportare) che se disposte su di una base temporale, con un certo ordine, possono enfatizzare *dettagli critici* che ad una prima analisi possono esserci sfuggiti. Ad esempio:

Caratteristiche critiche TIM

- Qualunque sia l'operazione 1451 in esecuzione (*fetching* `TEDS`, acquisizione dati da `TrCh-x`, elaborazione di data-set per l'invio ad una richiesta NCAP *pending*) *la ricezione di un messaggio da parte del transceiver può sempre alterare il funzionamento corrente*
- Detto $\mathtt{m}_{(\mathtt{i,rx})}$ l' i-esimo messaggio arrivato al nodo TIM e detto $\mathtt{m}_{(\mathtt{i,tx})}$ la replica del nodo TIM al precedente messaggio, è necessario tenere traccia dell' indice i per evitare che una replica TIM corrisponda ad un messaggio NCAP diverso.

Elencate le caratteristiche ideali del nodo TIM e evinti alcuni importanti dettagli, passiamo ora ad aggiungere i vincoli che dipendono dalle *feature* della PicdemZ e del uC PIC18F4620. La prospettiva che useremo per definire tali specifiche si concentrerà su memoria e capacità di elaborazione del uC.

Vincoli memoria uC

Ricordando che abbiamo a disposizione 3968 byte di memoria RAM e 64 Kbyte di memoria programma (e dati statici) è utile:

- limitare l'uso di `struct` complesse che sprechino memoria e privilegiare quando possibile i `bitfield` [Bri88]
- fondamentale non usare array di `struct` complesse o quantomeno, limitarne al minimo possibile l'utilizzo
- cercare di allocare tutta la memoria necessaria all'inizio dell'applicazione e poi sfruttare quella già allocata al meglio durante l'esecuzione
- condiderare che un pacchetto 802.15.4 misura al più 128 byte. E' facile capire che dedicando 512 byte della memoria come *buffer* per i messaggi entranti ed uscenti del 1451, è possibile memorizzare al massimo 4 pacchetti contemporaneamente

Vincoli hardware PicdemZ

Per quando riguarda le capacità elaborative della scheda si può dire:

- il TC77 non è dotato di un pin di interrupt perciò è obbligatorio acquisire i campioni dal sensore in *polling*
- i campioni del sensore non hanno una base temporale regolare, poichè il sensore non ha caratteristiche di acquisizione complesse ed un nuovo campione è disponibile circa ogni 300 ms

- il MRF24J40 è dotato di meccanismo di interrupting verso il uC, pertanto tutti i messaggi in arrivo sulla radio possono sollecitare il uC
- avendo a disposizione la seriale sulla PicdemZ è facile produrre *output* con finalità di *debugging/testing*
- ad una frequenza di 16 Mhz il uC esegue con buona approssimazione al massimo 4 milioni di operazioni al secondo

Un'ulteriore breve riflessione è utile sui vari ambiti funzionali del codice del TIM. Come si avrà ampliamente avuto modo di capire, sono diversi e fra loro alquanto disgiunti gli ambienti operativi del TIM:

- ricezione/invio messaggi 1451 (e relativo *driving* della radio)
- comprensione dei comandi ricevuti e esecuzione di opportune porzioni di codice
- aggiornamento dello stato interno e dei propri transducer channel
- gestione file virtuali `TEDS`
- acquisizione data set dai `TrCh`

Tutti gli elementi finora esposti sono atti a permettere una migliore comprensione della vera e propria architettura del software che viene presentata nel seguito.

5.3.3 Architettura software

Nella figura 5.6 possiamo vedere gli elementi che comporranno la struttura del codice. Si è deciso di suddividere il software in porzioni differenti a seconda dell'ambito operativo. Gli elementi che lo costituiscono sono:

CommMng Communication Manager (1451.5)

IntMng Internal Manager (1451.5 *connection layer* 1451.0)

CmdInt Command Interpreter (1451.0)

CmdEx Command Executer (1451.0)

TEDSMng TEDS Manager (1451.0)

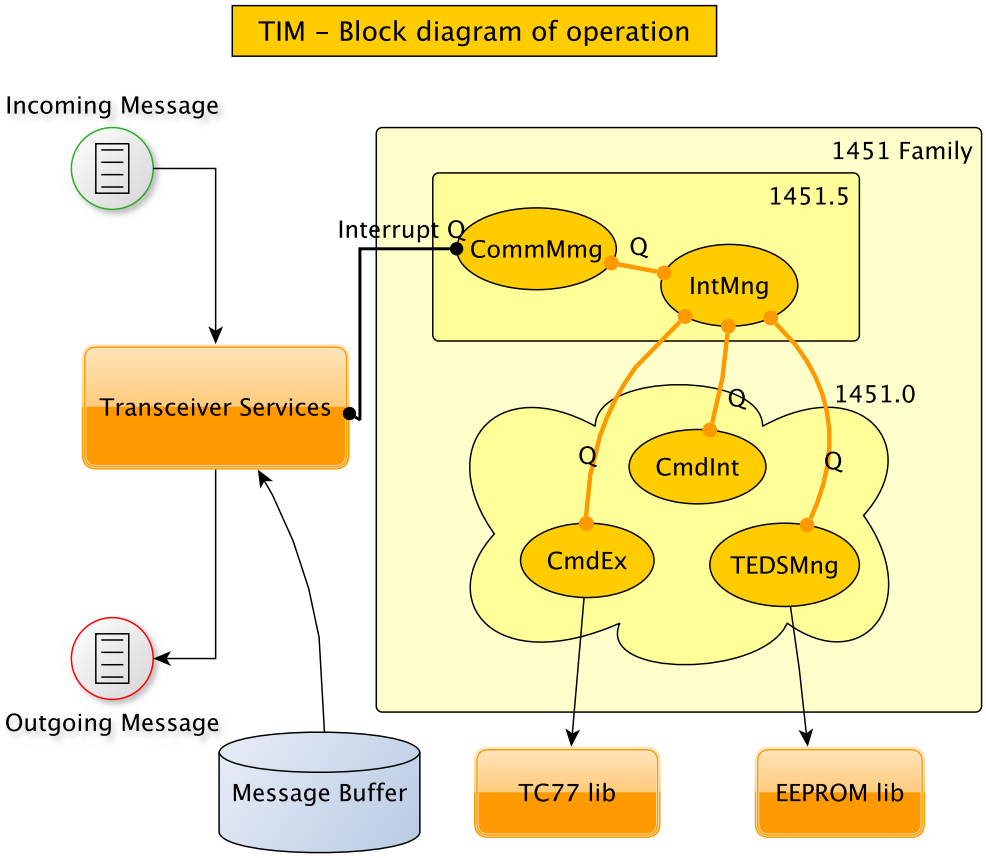

Figura 5.6: Diagramma dei componenti software sul nodo TIM

Il sistema è costuito da componenti software in grado di comunicare fra loro mediante un sistema di messaggistica interno.

Come si può vedere in figura infatti, fra i blocchi dei componenti esistono degli archi arancioni che li connettono: tali archi rappresentano un canale di comunicazione software fra i singoli. Da notare che transitando attraverso il nodo dell' IntMng, è possibile trovare un cammino da ogni blocco verso tutti gli altri.

Messaggistica interna

I canali sono delle vere e proprie *tubature* che permetteranno a ciascun componente di inviare un *messaggio* ad un altro componente.

Nel capitolo successivo, capiremo cosa sono veramente i messaggi fra i moduli, cosa contengono e a cosa servono.

CommMng

I componenti sono classificabili in due gruppi: il CommMng appartiene propriamente al layer del 1451.5 poichè il suo scopo è pilotare il driver della radio e ricevere notifiche dalla radio alla ricezione di un messaggio nuovo, mediante il canale di comunicazione denominato `Interrupt Q`. Esso fa uso di una porzione ausiliaria di memoria RAM che funge da buffer per un messaggio in entrata e uno in uscita.

IntMng

L'IntMng invece è un componente intermedio che permette di interfacciare il CommMng con gli altri tre che sono la vera e propria implementazione del 1451.0 sul nodo. Tra le funzionalità dell'IntMng figura anche il *routing* della messaggistica interna.

CmdInt

Il CmdInt è il vero e proprio interprete dei comandi 1451: suo compito sarà riconoscere e comprendere i vari comandi ricevuti dal nodo e disporre le opportune operazioni conseguenti: principalmente sollecitare il TEDSMng e il CmdEx. Esso è inoltre responsabile di tenere traccia dello stato del nodo TIM nonchè di tutti i suoi Transducer Channel (vedi figura 5.7 per maggiori informazioni).

CmdEx

Il CmdEx è responsabile della maggior parte di azioni previste dallo standard, ovvero la gestione dei Transducer Channel: interfacciamento verso il sensore TC77, variazione modalità di acquisizione, trasferimento data-set dal nodo TIM verso il buffer per i messaggi in uscita, ecc...

TEDSMng

Il TEDSMng è il componente deputato alla gestione (lettura/scrittura/update) dei datasheet virtuali, denominati TEDS in [14507a]. Tale componente è scorporato dal CmdEx poichè si interfaccia ad una porzione hardware-dependent del codice, infatti i TEDS saranno memorizzati nella memoria eeprom della PicdemZ.

Scelte implementative preliminari

Elenchiamo nel seguito alcune delle scelte implementative fondamentali: ciascun

Modulo software = FreeRTOS Task

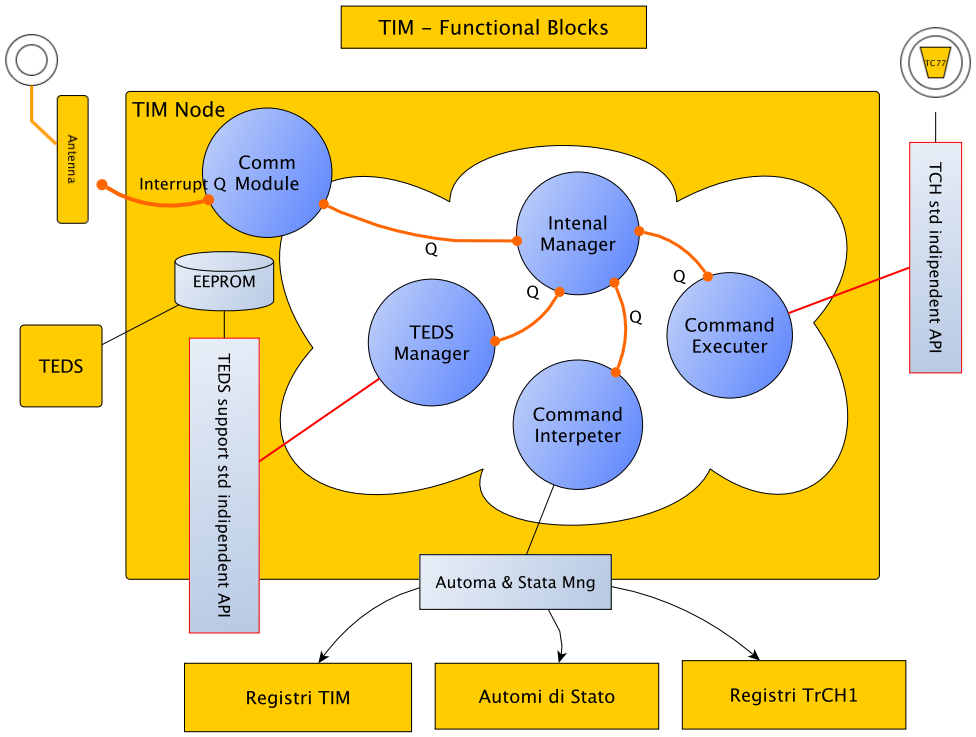

Figura 5.7: Rappresentazione accurata delle componenti software del nodo 1451. Da notare l'automa principale del nodo e quelli dei singoli canali

componente sarà implementato come un task FreeRTOS. Ciò significa che ciascun componente avrà una struttura indipendente dagli altri. Ogni task, verrà pertanto mandato in esecuzione, dallo scheduler di FreeRTOS, secondo la politica di scheduling scelta.

Canale di comunicazione = FreeRTOS Queue

I canali di comunicazione sfrutteranno le code fornite dal sistema operativo (ecco perchè la dicitura *Q* nella figura 5.6): ogni componente avrà pertanto una coda di messaggi in ingresso da servire; ogni qualvolta un task verrà mandato in esecuzione si occuperà di elaborare il più vecchio dei messaggi in coda. Ogni task durante l'esecuzione può a sua volta generare uno o più messaggi verso gli altri membri del sistema. Una volta riesumato dal sistema operativo esso procederà al *processing* del messaggio successivo e così via. All'interno delle code saranno inseriti messaggi nella forma $< \texttt{from}, \texttt{to}, \texttt{opCode} >$ il cui significato è: task mittente, task destinatario, operazione richiesta al task destinatario.

Sistema modulare con messaggistica

Caso leggermente diverso è la Interrupt Q: essa ha la peculiarità di essere popolata esclusivamente da una ISR invece che da un task. Lo scenario operativo è sostanzialmente il seguente: il transceiver radio puà ricevere un messaggio

mentre un task è impegnato ad elaborare un messaggio in coda. Tale fatto comporta il *firing* di un interrupt verso il microcontroller: esso risponderà immediatamente con una interrupt service routine che invierà al CommMng una notifica della ricezione di un nuovo messaggio dalla rete. Ora il sistema viene in questo modo informato della ricezione di un messaggio nuovo e può riprendere l'esecuzione consapevole di quanto è appena accaduto e gestire il nuovo messaggio non appena possibile.

La gestione degli interrupt radio

Il buffer di memoria presente in figura serve per la memorizzazione in RAM (in modo da essere disponibile a tutti i task) del messaggio incoming correntemente in elaborazione e per lo storaging dell'eventuale messaggio outcoming in composizione dai vari task. Per far sì che non si verifichino incongruenze nella gestione del buffer, esso è protetto da due semafori FreeRTOS in modo da impedire la sovrascrittura di messaggi non ancora processati.

Buffer incoming/outgoing

Il principio generale di funzionamento del sistema dovrebbe essere ormai facilmente ipotizzabile. Per sciogliere ogni dubbio, vediamo di presentare un rapido resoconto.

Immaginiamo che il sistema si trovi in una sorta di stato `idle`, ovvero ogni task viene mandato in esecuzione periodicamente dallo scheduler di FreeRTOS, ma nessuno dei task ha messaggi in coda, pertanto nessuno ha *pending processes*.

Sistema idle

Alla ricezione di un messaggio sulla radio, il driver del transceiver, allertato mediante interrupt, segnala il fatto al CommMng mediante un messaggio sulla coda Interrupt Q. L'esecuzione riprende normalmente dopo la ISR; appena raggiunto il turno del task CommMng, quest'ultimo, controllando i messaggi in coda, gestirà il messaggio in arrivo. Esso dunque procederà con la copia del messaggio dal buffer del transceiver verso il buffer nel TIM. Se il messaggio sarà *well-formed*, segnalerà che è presente un nuovo messaggio 1451 da interpretare nel buffer e accenderà il semaforo in modo da impedire la sovrascrittura del messaggio in ingresso.

Ricezione messaggio

Tale segnalazione comporterà la richiesta di interpretazione comando, diretta al task CmdInt. Durante il turno di esecuzione del CmdInt verrà processato il messaggio nel buffer e in base alla classe comando e all'identificativo comando verrà inoltrato un ulteriore messaggio verso il task CmdEx o il task TEDSMng. Quello sollecitato fra i due, non appena entrato in esecuzione, inizierà la vera e propria fase di processing. Durante questa operazione il task che esegue il comando pone il `lock` sul buffer in uscita in modo da allestire il pacchetto di risposta senza che altri task possano sovrascriverlo.

Interpretazione comando

Esecuzione comando

Completato il processing, il task che ha eseguito il comando invierà al CommMng richiesta di invio del messaggio in uscita e eventualmente segnalerà al CmdInt la necessità di alterare uno o più automi del nodo per segnalare variazioni di stato o codici di errore. Sbloccherà poi il `lock` sul buffer incoming. Il CommMng, durante il suo turno, chiamerà il driver dell' MRF24J40 e disporrà la trasmissione

Invio reply

del pacchetto presente nel buffer di uscita. Completata la trasmissione, il task CommMng eliminerà l'ultimo `lock` sul buffer outgoing, rendendolo nuovamente disponibile agli altri task.

Sistema nuovamente idle

Ora il sistema è nuovamente in grado di gestire un'altra richiesta.

La casistica di funzionamento è assai più complessa in realtà, ma la precedente trattazione è un ottimo punto di partenza per descrivere il principio di funzionamento del software che verrà illustrato ancor più nel dettaglio nel seguito.

Ora finalmente andiamo alla descrizione del software.

Capitolo 6

Implementazione del 1451.0

In questo capitolo si spiega nel dettaglio come è scritto il codice dello standard 1451.0 sul nodo TIM. Viene offerta al lettore una carrellata dei file e viene poi spiegato il ruolo di ogni funzione/macro/ opzione di compilazione e linking

Prima di iniziare la lettura di questo capitolo può essere di grande aiuto la lettura [Mic07a] e [Mic05c] per avere una minima famigliarità coi concetti che seguono. Per quanto riguarda FreeRTOS, si consigliano vivamente [Bar09a] e [Bar09b].

6.1 Operazioni preliminari

Download FreeRTOS

Per iniziare l'implementazione del progetto si è partiti con lo scaricare dal sito `http://www.freertos.org/` FreeRTOS V7.0.0.

Scaricato il pacchetto compresso del sistema operativo si procede all'estrazione dello stesso: quello che ci si trova davanti ora è un gran numero di cartelle e file. Tale gran varietà è dovuta al fatto che il pacchetto scaricato contiene tutte le applicazioni demo supportate da FreeRTOS. Per applicazione demo intendiamo un'applicazione costruita per testare FreeRTOS in un particolare *environment*: una combinazione di microcontrollore, scheda di sviluppo, compilatore ed eventualmente IDE. La prima cosa da fare è stato pertanto ripulire l'albero di FreeRTOS in modo da isolare la sola demo che era di nostro interesse, quella nella directory `PIC18_MPLAB`.

PIC18 official demo

Come già accennato in precedenza, fra le demo applications di FreeRTOS non figura direttamente la PicdemZ e tantomeno il PIC18F4620, pertanto il primo problema che si è risolto è stato adattare la demo del PIC18 per la scheda.

Tale risultato è stato raggiunto cominciando con lo studiare dettagliatamente l'applicazione demo ufficiale per il PIC18 di FreeRTOS, la relativa *board*,

PIC18 demo porting

il microcontrollore target: `http://www.freertos.org/index.html?http://www.freertos.org/a00097.html`.

Nel seguito vengono riportate le caratteristiche essenziali della demo:

architettura target PIC18

ambiente di sviluppo e compilatore MPLAB IDE e Microchip C18

development board FED development/evaluation board for 40 pin devices

microcontroller PIC18F452

Linker script, FreeRTOSconfig.h, configuration bits

Mediante lo studio approfondito di [Mic08b] e in particolar modo dei capitoli riguardanti l'organizzazione della memoria del uC, si è prodotto un nuovo script linker (si veda 6.2.1) adatto al più potente 18F4620. In seguito si sono corretti i parametri non compatibili nel file di configurazione principale di FreeRTOS (file `FreeRTOSconfig.h`). Si sono poi adattate le direttive per i *configuration bits*, (file `conf_bits.h`) in modo da supportare correttamente la velocità di clock del quarzo presente sulla PicdemZ. Il *porting* realizzato è liberamente disponibile online. (Si veda [Bru11]) per il download.

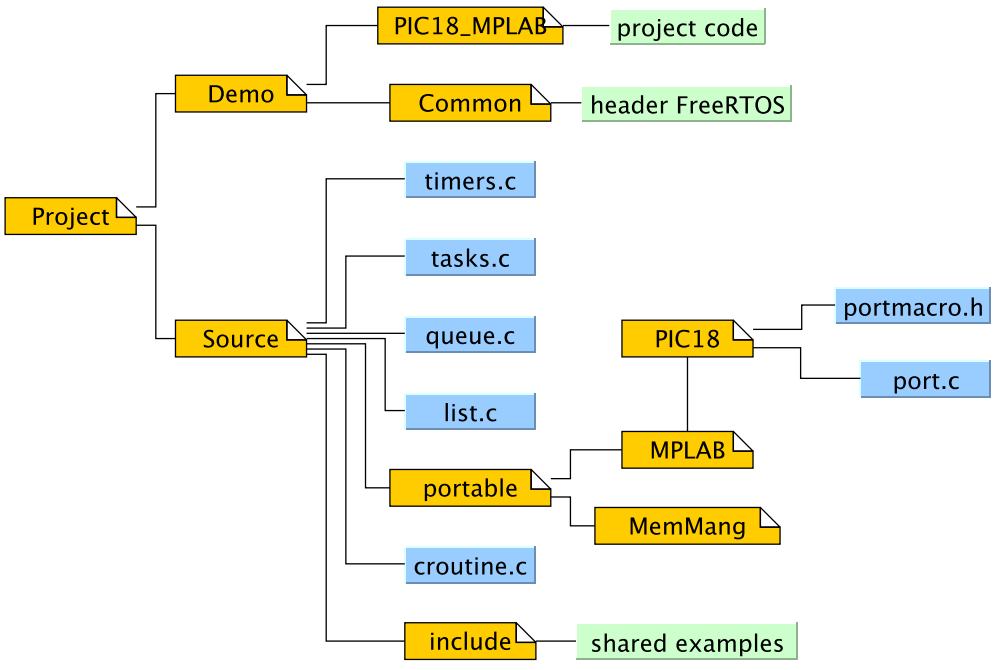

Figura 6.1: Il File system del progetto; in azzurro i file principali di FreeRTOS

Disponiamo da qui innanzi di un *workspace MPLAB* equipaggiato del codice necessario per inserire correttamente sul nostro futuro nodo TIM, il sistema operativo real-time FreeRTOS.

6.2 Filesystem del progetto

Presentiamo dunque la struttura del filesystem del progetto. Come si osserva in figura 6.1 il codice è organizzato in due directory principali: `Demo` e `Source`. Que-

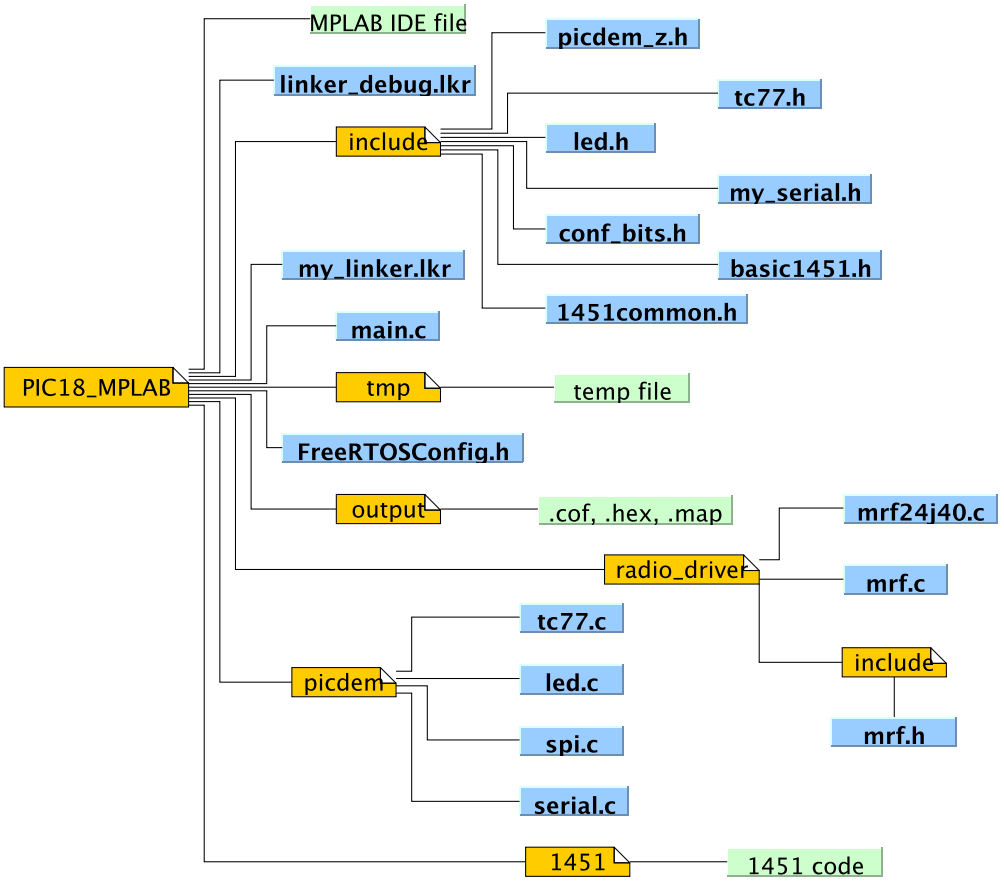

Figura 6.2: Il File system dell'applicazione; sono riportati solo i file più significativi

st'ultima contiene tutti i file sorgente del sistema operativo FreeRTOS, nonchè due ulteriori directory:

- `include`

- `portable`.

La prima contiene gli header file di FreeRTOS, la seconda invece contiene altre due directory `MemMang` e `MPLAB`, rispettivamente contenenti l'implementazione dello *heap* per la gestione della memoria da parte dell' OS e il codice di FreeRTOS *architecture-depending*. La directory `Source` contiene codice che non *va assolutamente toccato, ritoccato, modificato, rimosso*, pena il mancato funzionamento del RTOS.

`Demo` invece, contiene due directory, `Common` e `PIC18_MPLAB`. `Common` contiene esempi di utilizzo delle API condivisi fra le varie demo di FreeRTOS e i relativi header file. Il contenuto di `PIC18_MPLAB` è visibile nella figura 6.2: tale directory contiene tutto il codice sorgente. Facciamo ora una rapida overview di tutti gli elementi a profondità uno nella figura 6.2.

Workspace MPLAB

La nuvola gialla in 6.2 raggruppa i file necessari per il funzionamento dell'ambiente di sviluppo, il più importante di essi è il file workspace di MPLAB `RTOSWorkSpace.mcw`.

MPLAB Workspace

Usando la funzionalità *Open workspace..* all'interno dell'IDE è possibile richiamare l'intero progetto software con le relative impostazioni:

- file sorgente .c da compilare per ottenere l'eseguibile da scrivere nel uC
- header file del progetto e di FreeRTOS
- *path* relativi degli header locali e globali
- *path* dei file della libreria
- impostazione di compilazione per il PIC18F4620
- impostazioni per MPLAB ICD 2
- linker script per versione di *debug* e di *release*
- impostazioni per l'ottimizzazione in fase di compilazione

Directory output

Le directory `tmp` e `output` sono directory ausialiarie e contengono rispettivamente i file oggetto *post-compiling*, pronti per il linking e il risultato finale del processo di compilazione *post-linking*. Ogni qualvolta viene richiesta la compilazione dei sorgenti, i file oggetto coinvolti vengono rigenerati e vengono poi ri-linkati per ottenere un nuovo file .hex: formato adatto al *flashing* del microcontrollore. Si veda la figura 1-3 di [Mic05c] per comprendere il meccanismo di funzionamento dell'operazione di compilazione e le relazioni fra i file.

PIC18_MPLAB

Notiamo inoltre che sono presenti quattro file molto importanti e pertanto illustrati nel seguito:

main.c c source file contenente la funzione `main()`

FreeRTOSConfig.h header file di configurazione di FreeRTOS

my_linker.lkr linker script per *release mode*

linker_debug.lkr linker script con supporto alla modalità *in-circuit-debug* dell'ICD2

Ultime, ma non meno importanti, le directory contenenti il source code classificato in base al ruolo funzionale nell'architettura del software:

1451 source code del 1451

picdem source code per l'hardware della scheda

radio_driver source code per la daughter-card MRF24J40

include header file globali per tutti i source file

Si conclude così la rapida panoramica del filesystem del software. Passiamo ora a spiegare con maggior dettaglio qual'è il ruolo dei file più importanti.

6.2.1 File principali

Nel seguito riportiamo le porzioni salienti dei file principali e ne illustriamo le righe essenziali, in modo da consentire a chiunque voglia utilizzare e magari migliorare questo lavoro di muoversi agilmente nel codice.

main()

```
1  /* Scheduler include files. */
2  #include <p18f4620.h>
3    ...
4  /* App include files. */
5  #include "conf_bits.h"
6    ...
7  void main(void){
8   vPortInitialiseBlocks();
9    xSerialPortInitMinimal(mainBAUD_RATE, \
10     mainCOMMS_QUEUE_LENGTH);
11   ...
12  vLedInitialize();
13  radioInit();
14  xTaskCreate(vIntMngTask, "T0", 150, NULL, 1, NULL);
15   ...
16  vTaskStartScheduler();
17   xSerialPutChar( NULL, 'Q', mainNO_BLOCK );
18 }
```

Listing 6.1: `main.c` : porzioni salienti del `main()`

In 6.1 sono riportate le porzioni salienti del codice del `main()` del progetto. Essendo tale funzione l'entry point di qualsiasi programma C è importante capire brevemente come tutto il software viene avviato.

Le seguenti funzioni hanno un ruolo importante:

vPortInitialiseBlocks() inizializza lo heap di FreeRTOS per la gestione della memoria del micro

xSerialPortInitMinimal(..) inizializza il driver seriale

vLedInitialize() inizializza il uC per pilotare i led

radioInit() inizializza la comunicazione SPI fra microcontrollore e transceiver

xTaskCreate(..) e vTaskStartScheduler() sono funzioni delle API di FreeRTOS molto importanti

xTaskCreate(..)

`xTaskCreate(vIntMngTask, T0, 150, NULL, 1, NULL);`, funzione propria delle API di FreeRTOS, ci permette di creare un task all'interno del RTOS. Cerchiamo di capire il senso dei parametri passati alla funzione.

`vIntMngTask` è un puntatore a funzione contenente l'implementazione del task che intendiamo creare

`T0` è un'etichetta (stringa) per rappresentare il task in caso si desiderassero le funzionalità di logging di FreeRTOS.

`150` è la porzione di memoria riservata in byte (150 in questo caso) per il corretto funzionamento dello stesso

`1` è la priorità assegnata al task, in una scala che va da 0 a `configMAX_PRIORITIES`, parametro di configurazione presente nel file `FreeRTOSConfig.h`. Da notare che al task idle di FreeRTOS è sempre assegnata priorità 0, pertanto la priorità minima assegnabile di fatto è 1.

Si consiglia ovviamente di vedere [Bar09b] per una spiegazione più esaustiva.

Creazione dei task

La cosa importante ai fini della nostra trattazione è che nel `main()`, facciamo 5 chiamate a questa funzione: ciascuna creerà i task che costituiscono l'architettura software del nodo TIM. I task creati saranno dunque:

- `vIntMngTask`
- `vCommMngTask`

- `vCmdIntTask`
- `vCmdExTask`
- `vTEDSMngTask`

I nomi dei task dovrebbero essere piuttosto famigliari al lettore (si veda 5.3.3). Ciascun task inizia con la lettera `v` seguendo le convenzione di FreeRTOS, infatti le funzioni che implementano il task sono di tipo `void` e per chiarezza sono suffissati con la stringa `Task`. Vedremo nel seguito dove si trova l'implementazione dei task.

vTaskStartScheduler()

Si occupa invece di avviare lo scheduler di FreeRTOS. Non appena eseguita questa funzione la gestione del software in esecuzione sul uC passa dal flusso *naturale* del codice, a quello stabilito dalle impostazioni dello scheduler. Da adesso in poi i nostri task sono attivi e funzionanti sul nodo TIM.

Configurazione di FreeRTOS

FreeRTOS customization

Un altro file importante è `FreeRTOSconfig.h` di cui riportiamo solo le righe fondamentali in 6.2. Tale header file è utilizzato per configurare FreeRTOS.

```
1 ...
2 #define configUSE_PREEMPTION 0
3 #define configTICK_RATE_HZ (( portTickType)100)
4 #define configCPU_CLOCK_HZ ((unsigned long )16000000)
5 #define configTOTAL_HEAP_SIZE ((size_t)2860)
6 ...
```

Listing 6.2: `FreeRTOSConfig.h` : Configurazione di FreeRTOS

È sostanzialmente composto di righe nella forma:
`# define configOPTION_NAME value_of_option`
il cui significato dovrebbere essere ovvio.

Le opzioni che abbiamo sfruttato per la PicdemZ sono:

`USE_PREEMPTION` utilizziamo lo scheduler in modo cooperativo

`TICK_RATE_HZ` $100Hz$ frequenza del tick di FreeRTOS

`CPU_CLOCK_HZ` frequenza del quarzo e PLL pari a $16Mhz$

`TOTAL_HEAP_SIZE` memoria allocata dallo heap di FreeRTOS

Linker script

Linker script

Un altro file molto importante per il progetto è `my_linker.lkr`. Esso contiene le direttive di linking utilizzate da mplink per produrre il file hex finale. Le direttive possibili sono molte e assai complesse: per capirne le potenzialità si veda [Mic09a].

Confrontiamo lo script di linking di default del C18 per il PIC18F4620:

```
1 ...
2 ACCESSBANK NAME=accessram  START=0x0  END=0x7F
3 DATABANK   NAME=BIG_BLOCK  START=0x80 END=0xF7F
4 ACCESSBANK NAME=accesssfr  START=0xF80  END=0xFFF PROTECTED
5 ...
```

Listing 6.3: `my_linker.lkr` : release linker script del progetto

```
1 ...
2 ACCESSBANK NAME=accessram  START=0x0                END=0x7F
3 DATABANK   NAME=gpr0       START=0x80               END=0xFF
4 DATABANK   NAME=gpr1       START=0x100              END=0x1FF
5 ...
6 DATABANK   NAME=gpr14      START=0xE00              END=0xEFF
7 DATABANK   NAME=gpr15      START=0xF00              END=0xF7F
8 ACCESSBANK NAME=accesssfr  START=0xF80              END=0xFFF
  PROTECTED
9 ...
```

Listing 6.4: `18f4620_g.lkr` : C18 default linker script per PIC18F4620

Come si può vedere in 6.3, i singoli banchi di ram del uC (`gprX` in 6.4 dove `X` è l'indice del banco) sono mappati in un solo macro-blocco `BIG_BLOCK` in modo da assegnare tutta la memoria disponibile sul micro allo heap e sfruttare poi quest'ultimo per le allocazioni.

Nel codice sorgente è presente il file `linker_debug.lkr` che è sostanzialmente basato su `my_linker.lkr` ma supporta altresì le funzioni avanzate di *in-circuit-debug* del programmer ICD2. Da notare che gli script sono mutuamente esclusivi: il primo è adatto al flashing finale, il secondo invece è pensato per sviluppo/debugging.

Header file globali

PIC18 configuration bits

Nella directory `include` sono contenuti header file comuni a tutto il progetto. Uno dei più importanti di questi (pena l'assenza di segni di vita da parte del microcontrollore) è il file `conf_bits.h`. Esso contiene delle direttive particolari la cui finalità è l'impostazione dei bit di configurazione del microcontrollore. Si veda la documentazione allegata al compilatore C18 per tutte le opzioni possibili sul PIC18F4620.

6.2.2 I Task del 1451

L'implementazione dei task del 1451 si trova all'interno della directory `1451`. In quest'ultima si trova la directory `include` che contiene gli header file dei task. Come visibile in figura 6.3 trovano collocazione quì anche altri file .c importanti:

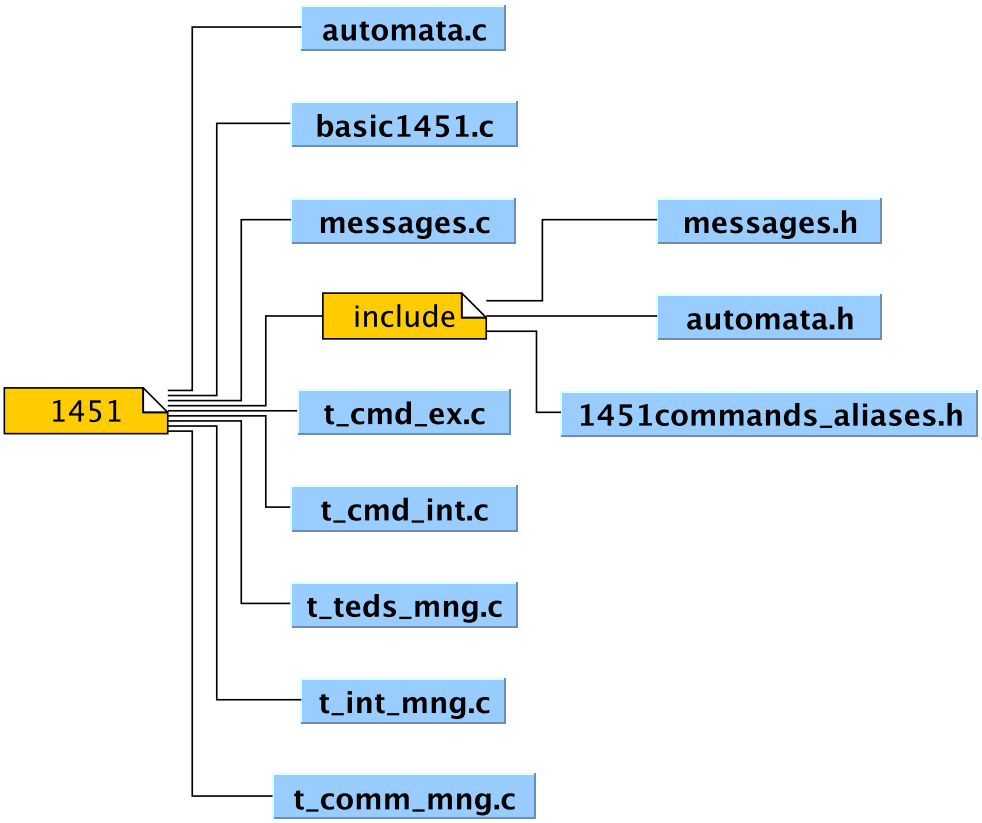

Figura 6.3: I file che costituiscono l'implementazione del 1451

`automata.c` automa deterministico generico, compatibile con la definizione di [HMU07]

`messages.c` messaggistica interna e buffer ram per la radio

`basic1451.c` pool di funzioni per la gestione dei comandi del 1451 (si veda 6.2.2)

`t_gp_task.c` task ausiliario utilizzato con funzionalità di testing del codice del TIM

I task del 1451

Passiamo ora ad una rapida presentazione di tutti i task del 1451. Si ricordi che il ciclo di lavoro standard di uno qualunque dei task è un'estensione dell'automa in figura 5.1, ovvero:

1. controlla se ci sono messaggi in coda
2. se ce ne sono analizza il messaggio altrimenti termina
3. se il messaggio è lecito processa l'azione corrispondente al codice del messaggio
4. termina esecuzione

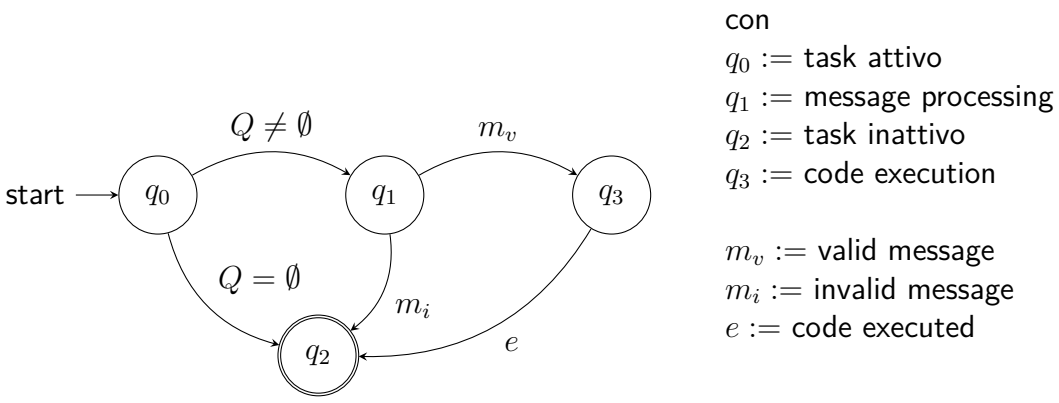

Figura 6.4: DFA basilare per i task del 1451

Communication Manager

Il Communication Manager è implementato all'interno del file `basic1451.c`. Esso non trova collocazione come gli altri task in un file separato poichè funge semplicemente da *wrapper* per il driver della radio che verrà esaminato nel seguito. Il principio di funzionamento è esattamente quello del task in figura 5.1.

Internal Manager

L' Internal Manager trova implementazione nel file `t_int_mng.c`. La sua esecuzione comincia con la creazione *una-tantum* di tutte le code che compongo il sistema. Fatto ciò come tutti gli altri task si mette in modalità di *listening* di eventuali messaggi in coda e li processa.

Esso si occupa di inoltrare ai vari task i messaggi e gestisce le macro-variazioni di stato del sistema. Come già anticipato nel capitolo sul design funge da nodo di routing fra i task in modo da simulare un grafo completo di comunicazione. Processa inoltre i comandi di variazione di stato degli automi del nodo TIM.

Command Interpreter

Il Command Interpreter si occupa di processare i comandi presenti nel buffer del sistema. Dopo averne accertata la bontà sintattica e la validità, inoltra un messaggio opportuno al Command Executer piuttosto che al TEDS Manager. SI occupa inoltre di segnalare i difetti nei comandi negli opportuni registri di stato del TIM. L'implementazione si trova nel file `t_cmd_int.c`. I comandi eseguiti dal Command Interpreter sono

Comandi CmdInt

Class 1

- run_self_test_cmd()
- write_serv_req_mask_cmd()
- read_serv_req_mask_cmd()
- read_status_event_reg_cmd()
- read_status_condition_reg_cmd()
- clear_status_event_register_cmd()
- write_status_event_state_cmd()
- read_status_event_state_cmd()

Class 5

- wake_up_cmd()

Class 6

- read_tim_ver_cmd()
- tim_sleep_cmd()
- store_op_setup_cmd()
- recall_op_setup_cmd()
- read_ieee_1451_ver_cmd()

Class 7

- reset_cmd()

I comandi qui riportati hanno una nomenclatura che fa chiaro riferimento al nome ad alto livello fornito nello standard allo specifico comando.

Command Executer

Il Command Executer si occupa di mettere in esecuzione tutti i comandi relativi alla gestione dei Transducer Channel. È implementato nel file `t_cmd_ex.c` e i comandi gestiti sono quelli riportati:

Comandi CmdEx

Class 2

- set_tch_data_rep_count_cmd()
- set_tch_p_trigger_count_cmd()
- address_group_def_cmd()
- sampling_mode_cmd()
- data_transmission_mode_cmd()
- buffered_state_cmd()
- end_of_data_set_op_cmd()
- actuator_halt_mode_cmd()
- edge_to_report_cmd()
- calibrate_tch_cmd()
- zero_tch_cmd()
- write_corrections_cmd()
- read_corrections_cmd()
- write_tch_initiate_trig_state_cmd()
- write_tch_initiate_trig_conf_cmd()

Class 3

- read_tch_data_seg_cmd()
- write_tch_data_seg_cmd()

- trigger_command_cmd()
- abort_trigger_cmd()

Class 4

- tch_operate_cmd()
- tch_idle_cmd()
- write_tch_trig_state_cmd()
- read_tch_trig_state_cmd()
- read_tch_data_rep_count_cmd()
- read_tch_p_trig_count_cmd()
- read_add_group_cmd()
- read_samp_mode_cmd()
- read_data_tx_mode_cmd()
- read_buff_state_cmd()
- read_eod_set_op_cmd()
- read_actuator_halt_mode_cmd()
- read_edge_to_report_cmd()
- read_tch_init_trig_state_cmd()
- read_tch_init_trig_conf_cmd()
- device_clear_cmd()

Vista la finalità del task l'elenco è ovviamente più nutrito.

TEDS Manager

Il TEDS Manager infine gestisce i comandi: Comandi TEDSMng

Class 1

- query_teds_cmd()
- read_teds_cmd()
- write_teds_cmd()
- update_teds_cmd()

Esso è implementato nel file `t_teds_mng.c`. Questo sottoinsieme di comandi del 1451.0 è relativo, come da design, alla gestione dei teds sul nodo TIM. È scorporato dal Command Executer poichè usa librerie hardware-dependent e nell'ottica di sviluppare una libreria, tutte le chiamate che sfruttano hardware specifico vanno incapsulate in modo da poter essere rimpiazzate in caso di variazione del nodo target.

General Purpose Auxiliary Task

Test Task

Un task di cui non si è fatta menzione finora nella trattazione, trova implementazione nel file `t_gp_task.c`. Esso è un task ausiliario, istanziato solo in caso di necessità di debugging/testing. È stato sfruttato estensivamente durante lo sviluppo come generatore di messaggi interni. Essendo infatti tutte le azioni intraprese dal sistema *triggerate* dalla ricezione di un messaggio in un dato task, il GPTask è stato utilizzato come generatore di messaggi per verificare il corretto comportamento dei task del 1451.0.

I comandi 1451.0

Ciascuno dei task che esegue un qualsiasi comando del 1451, invoca una delle macro presenti nel file `1451commands_aliases.h` nella directory `include` del 1451. Ogni macro corrisponde ad uno specifico comando dello standard. Ciascuna macro è poi associata ad una specifica funzione c che ne contiene l'implementazione.

Alias comandi

L'idea di fondo del meccanismo è la seguente: per non alterare la struttura dei task qualora si debba fornire una migliore implementazione di un dato comando del 1451 è sufficiente variare l'associazione fra la macro e la relativa funzione c. Il task continuerà ad invocare la stessa macro ma la macro punterà ad una funzione c differente.

```
1 ...
2 //CommandInterpreter
3 //Class 1
4 #define run_self_test_cmd() CmdIntSelfTest()
5 #define write_serv_req_mask_cmd() CmdIntWriteServReqMask()
6 #define read_serv_req_mask_cmd() CmdIntReadServReqMask()
7 ...
```

Listing 6.5: `1451commands_aliases.h` : la seconda stringa è il nome della macro, la terza la funzione c; i comandi sono raggruppati per classi

È sufficiente pertanto variare la terza stringa in ciascuna delle righe del file 6.5 per far eseguire al sistema un comando piuttosto che un altro. Il lettore smaliziato intuisce certamente qual'è la finalità di questa scelta: l'implementazione del set di

comandi sarà una libera scelta dello sviluppatore a seconda delle proprie necessità o a seconda dell'hardware della scheda. Nonostante questo l'architettura del sistema rimane invariata. Ecco una delle altre caratteristiche che la rendono una libreria FreeRTOS e non una applicazione specifica per il PicdemZ.

Lo scheletro dei comandi default è implementato nel file `basic1451.c`.

6.3 I driver ausiliari

Come anticipato nella precedente sezione, alcuni task utilizzano primitive che ovviamente dipendono dall'hardware della scheda. Tali primitive riguardano il *driving* del sensore di temperatura TC77, il transceiver radio, la eeprom per la memorizzazione dei teds. Oltre a questi tre elementi principali sono state sviluppate altre due librerie. Una per gestire i led della scheda in modo da fornire feedback visivo durante l'esecuzione e un'altra per sfruttare la porta seriale presente sul PicdemZ. Connettendo la scheda ad una macchina dotata di un software per console seriale è possibile monitorare l'esecuzione di ciascuno dei task nel sistema.

Tutti questi componenti sono presentati in dettaglio nel prossimo capitolo.

Capitolo 7

Driver ausiliari

In questo capitolo prendiamo in esame la porzione del codice che si occupa del vero e proprio interfacciamento con l'hardware della PicdemZ. Si trovano nel seguito presentate le librerie per la gestione dei led della scheda, dell'interfaccia USART, del sensore di temperatura e del radio transceiver.

Prima di affrontare la trattazione, si consiglia di leggere la guida introduttiva alla scheda, fornita dalla Microchip®: [Mic08c]. Per maggiori informazioni, si vedano gli specifici *datasheet* e le *application note* cui si fa riferimento nel testo.

Macro wrapping

Altra importante precisazione, utile allo sviluppatore che dovesse utilizzare/-migliorare quanto sviluppato è il fatto che tutto il codice che impartisce al microcontrollore direttive pertinenti all'hardware fa uso di un header file di appoggio, ovvero il file `picdem_z.h`. In tale file sono ridefinite con un nome alternativo tutte le macro della libreria standard del compilatore, relative al PIC18F4620. Tale fatto consente di riadattare facilmente il codice qualora si vogliano usare le librerie su di una scheda o su di un microcontrollore leggermente diverso, oltre a rendere nettamente più leggibile il codice.

7.1 LED driver

Nel mondo della programmazione è prassi iniziare l'esperienza con un nuovo linguaggio mediante il classico programma *Hello World*. Nel mondo dei microcontrollori, il programma *Hello World* diventa il *blinking* di un led, ovvero istruire il uC in modo da far lampeggiare un led ad esso connesso.

Si consiglia di leggere attentamente gli esempi di *led-blinking* contenuti in [Mic05c] per capire gli elementi fondamentali di questo processo.

Led blinking

Per onorare la tradizione, il primo task che si è implementato per testare il corretto funzionamento di FreeRTOS aveva il compito di far lampeggiare un led.

Per ottenere questo risultato sono state realizzate tre semplicissime funzioni contenute del file `led.c`. Ne riportiamo uno spaccato in 7.1.

```
1  // Initialize pin of micro for driving leds
2  void vLedInitialize(void){
3    ...
4  }
5
6  // Set uxLED led (possible values are 0..5)
7  // to value xValue (possible values are 0,1)
8  void vLedSetLevel(unsigned portBASE_TYPE uxLED,
9  portBASE_TYPE xValue){
10   ...
11 }
12
13 // Toggle uxLED led (possible values are 0..5)
14 void vLedToggle(unsigned portBASE_TYPE uxLED){
15   ...
16 }
```

Listing 7.1: `led.c` : semplice libreria per la gestione dei led della PicdemZ

Come si può vedere le funzioni implementate hanno un ruolo piuttosto ovvio:

`void vLedInitialize(void)`
: inizializza i registri TRIS dei pin coinvolti ion modo da abilitare la scrittura su di essi

`void vLedSetLevel(char uxLED, char xValue)`
: sfruttando i parametri `uxLED` e `xValue` è possibile settare il valore di un determinato led. Nel seguito riportiamo una lista riportante gli `uxLED` passabili, il relativo pin pilotato e l'alias ad alto livello, definito in `picdem_z.h`:

0 LATDbits.LATD4 (Led0)

1 LATDbits.LATD5 (Led1)

2 LATDbits.LATD6 (Led2)

3 LATDbits.LATD7 (Led3)

4 LATAbits.LATA1 (Led1G)

5 LATAbits.LATA0 (Led0G)

gli `xValue` ammessi sono ovviamente:

0 led spento

1 led acceso

void vLedToggle(char uxLED)
come facilmentente intuibile dal nome della funzione, essa ha lo scopo di commutare di stato il led uxLED, ovvero:

- se uxLED è spento → uxLED viene acceso
- se uxLED è acceso → uxLED viene spento

Gli uxLED ammissibili sono gli stessi della funzione precedente.

Nel file led.c sono inoltre implementati quattro semplici task di prova, ognuno di essi si occupa del *blinking* di un led. Per muovere i primi passi con FreeRTOS è possibile usare la primitva xTaskCreate(...) (già discussa nel capitolo precedente) per istanziare i suddetti task e iniziare a comprenderne il funzionamento e la struttura.

7.2 Serial driver

Per far funzionare correttamente la comunicazione fra un pc e la PicdemZ mediante la porta seriale, è stato necessario scrivere un semplice driver. Esso è stato realizzato analizzando la struttura del driver seriale di FreeRTOS della demo del PIC18 e mediante attenta lettura del capitolo sulla USART di [Mic08b]. Da notare che questo driver pur essendo abbastanza *conciso* è assai più complesso del driver che pilota i led, poichè sfrutta alcuni SFR (*special function register*) del microcontrollore per la configurazione del modulo EUSART.

I file che costituiscono l'implementazione sono tre:

serial.c contiene l'implementazione delle funzioni che realizzano il *driving* della USART e che verrano illustrate nel seguito

serial.h contiene i prototipi delle funzioni precedenti

my_serial.h contiene delle macro per rendere più agevole l'utilizzo del codice.

Invio e ricezione di un carattere

Il driver dal punto di vista strutturale è realizzato in modo molto semplice: una porzione del codice serve ad inizializzare i registri di configurazione del uC in modo da sfruttare la periferica USART. La comunicazione da e verso la porta è implementata mediante due code: una per i caratteri in ingresso e una per i caratteri in uscita. Il codice infatti incorpora due funzioni: una per leggere un carattere e una per scriverlo; riportiamo le funzioni implementate a tale scopo:

xSerialPutChar(NULL, unsigned char toSend, NULL)
Invia immediatamente il carattere toSend alla porta seriale

xSerialPutChar(NULL, unsigned char* toRecv, NULL)
: Riceve immediatamente il prossimo carattere sulla seriale e lo salva nella variabile `toRecv` passata per riferimento.

In un qualunque punto del codice è possibile invocare le due precenti funzioni per inviare o ricevere caratteri. Molto utile dal punto di vista dello sviluppo è stato poter inviare alla seriale stringhe per il debugging del codice.

7.3 SPI driver

Invio di un byte su SPI

Dopo la trattazione sull'hardware della PicdemZ dovrebbe essere chiaro il meccanismo di funzionamento del bus SPI. La funzione che realizza la trasmissione di un byte è riportata nel listato 7.2.

```
1 unsigned char spi_byte(unsigned char send){
2   unsigned char recv;
3   MRF_CS_Active;
4   SSPBUF=send;
5   while(!SSPSTATbits.BF);
6   recv=SSPBUF;
7   MRF_CS_Inactive;
8   return recv;
9 }
```

Listing 7.2: `mrf.c` : Codice per l'invio di un byte sul bus SPI

Il codice esegue le seguenti operazioni:

- attiva la linea di trasmissione CS (nel listato specificamente quella del transceiver mrf24j40)
- invia al buffer della SPI `SSPBUF` l' *unsigned char send* passato alla funzione
- attende il completamento della trasmissione
- legge dal buffer SPI il byte inviato dall'altro dispositivo connesso al bus e lo salva nella variabile `unsigned char recv`
- disabilita la linea CS
- restituisce alla funzione chiamante il byte ricevuto

Per testare il funzionamento della funzione precedente e come si vedrà nel seguito, per testare la comunicazione fra il microcontrollore e il transceiver radio, si è utilizzato estensivamente l'oscilloscopio Agilent mixed-signal MSO 6032A. Quest'ultimo è stato connesso al PIC18 mediante una sonda digitale da 16 bit,

sfruttando però solamanente quattro linee. Le connessioni fra i puntali della sonda dell'oscilloscopio e i pin del PIC18F4620 sono:

- D8 - RC3, linea SCK
- D9 - RD1, linea CS
- D10 - RC4, linea SDI
- D11 - RC5, linea SDO

L'oscilloscopio è stato poi configurato di modo da riconoscere le quattro linee in figura come bus SPI. Tale fatto ha consentito di usare la modalità di trigger SPI supportata nativamente dallo strumento.

Riportiamo nelle figura 7.1 uno *screenshot* realizzato tramite l'oscilloscopio che dovrebbe chiarire in via definitiva il funzionamento del bus.

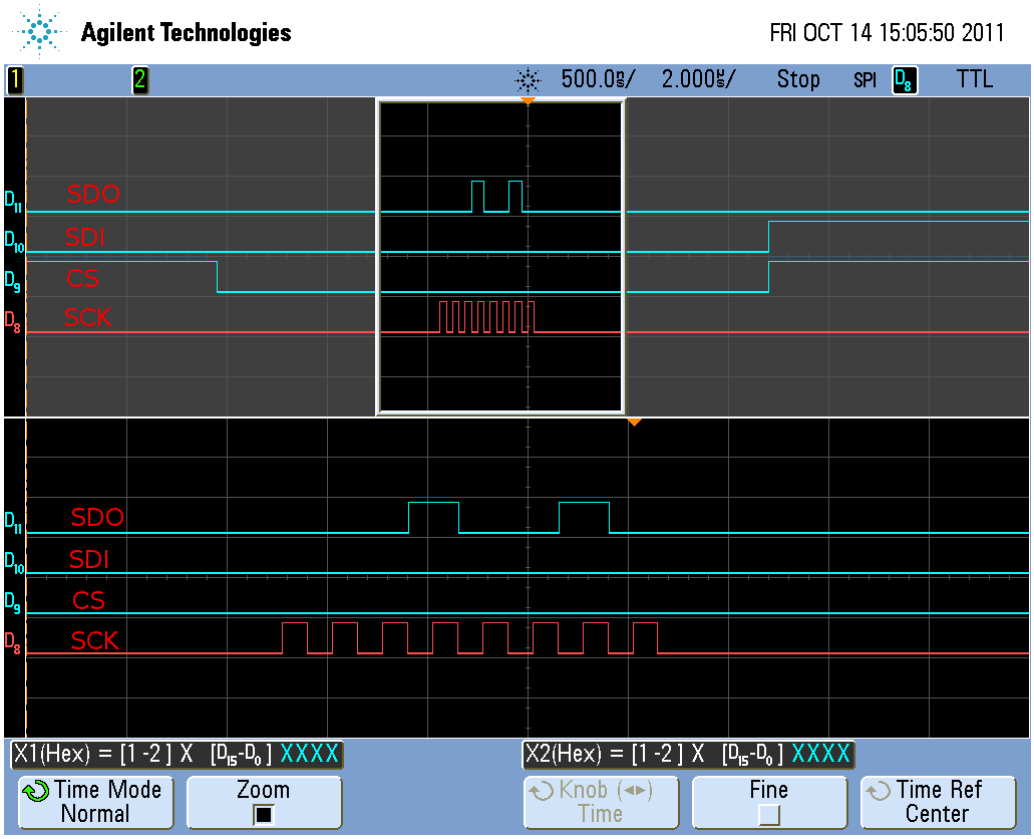

Figura 7.1: Invio di un byte sulla SPI (ingrandimento della trasmissione)

In figura è possibile vedere la variazione di fronte d'onda (*falling-edge*) della linea CS che segnala l'inizio di una trasmissione. Lungo la linea SCK è presente il clock trasmesso dal microcontrollore al dispositivo slave. Contemporaneamente al clock sulla linea SCK è possibile vedere sulla linea SDO la trasmissione del byte `0x12`. Sulla linea SDI è ricevuto il byte `0x00`. Terminata la trasmissione/ricezione di un byte, la linea CS varia nuovamente di stato. La funzione eseguita dal codice era pertanto `spi_byte(0x12)`.

7.4 TC77 driver

Il sensore di temperatura TC77 è interfacciato al uC mediante bus SPI ma in modo non standard. Il TC77 è connesso al microcontrollore mediante una linea CS, una linea SCK e una sola linea dati in cui sono collassate le linee SDI e SDO. La sovrapposizione di SDI e SDO (che chiameremo d'ora innanzi SDI/O) rende inutilizzabile l'hardware dedicato del microcontrollore.

Tale fatto ha richiesto la scrittura di un driver *bit-banged*. Tale dicitura viene usata nell'ambito della programmazione dei microcontrollori per definire una porzione di codice in cui, per effettuare la comunicazione con un qualche tipo dispositivo, si sfruttino i pin del uC in modalità GPIO, ovvero *general purpose input output*.

TC77 bit-banged driver

Specificamente, il clock sulla linea SCK è realizzato alzando e abbassando il valore TTL di un certo pin. Discorso analogo per la linea SDI/O: in una prima porzione del codice, essa è usata come input, ovvero viene letto via software il valore di tensione del pin; nel seguito essa viene usata come output, cioè viene impostato mediante software il valore TTL del pin. La lettura e la scrittura del valore TTL della linea SDI/O è ovviamente legato alla variazione del fronte d'onda della linea SCK.

Il driver del TC77 è contenuto nel file `tc77.c` ed è costituito dalla funzione:

int getTC77Sample()
: ad ogni invocazione essa restituisce in un intero da 16 bit il valore acquisito dal sensore (rappresentato con una parola da 13 bit in complemento a due).

Per ulteriori dettagli si veda il datasheet del sensore [Mic02] e l'application note [BL04].

7.5 MRF driver

Descriviamo ora il funzionamento della comunicazione e il meccanismo di controllo del transceiver radio.

Quest'ultimo, dal punto di vista del uC è semplicemente una memoria esterna sulla quale è possibile leggere e scrivere dei valori in alcune celle di memoria. La scrittura di opportuni valori, in determinate locazioni di memoria farà sì che l'hardware del transceiver riceva od invii i pacchetti 802.15.4 in aria.

Memoria MRF24J40

L'MRF24J40 è dotato di due memorie distine: una di controllo e una di archiviazione dei pacchetti. La prima di esse detta *short address memory space* misura 64 byte ed è indirizzabile nel range `0x00 - 0x3F`, con indirizzi da 6 bit. La seconda invece, detta *long address memory space*, molto più capiente, è indirizzabile fra `0x000 - 0x38F` con indirizzi da 10 bit.

La short memory contiene i registri che servono per controllare il funzionamento del transceiver. La long memory invece contiene sostanzialmente due buffer: uno destinato a contenere il pacchetto ricevuto denominato `RXFIFO` e uno per il pacchetto in trasmissione detto `TXFIFO`. Per maggiore chiarezza si riportano gli indirizzi dei buffer nella long memory:

TXFIFO `0x000 - 0x07F` di 128 byte

RXFIFO `0x300 - 0x38F` di 144 byte

Senza dilungarsi sui dettagli di funzionamento della ricezione e dell'invio di un pacchetto, poichè ampliamente descritti nel datasheet del transceiver, descriviamo gli elementi fondamentali operativi del MRF24J40:

- se è presente un pacchetto ricevuto, esso si trova nel buffer `RXFIFO`; da qui può essere trasferito al microcontrollore per elaborarne il contenuto (ed estrarre da esso il pacchetto di comando 1451.0)
- se si desidera inviare un pacchetto, esso va trasferito nel buffer `TXFIFO` e poi va dato ordine al transceiver, mediante scrittura in un registro short, di effettuare l'invio dei dati nel buffer
- il transceiver invia un interrupt hardware al microcontrollore sul pin INT0 alla ricezione di un nuovo pacchetto o al completamento della trasmissione di un pacchetto in uscita

Lettura e scrittura sul transceiver

Come si avrà avuto modo di capire da quanto detto poco sopra, per pilotare correttamente il transceiver e per trasferire i pacchetti 802.15.5 da e verso il microcontrollore è necessario leggere e scrivere dei valori nella long memory e nella short memory. Le operazioni di comunicazione sono effettuate mediante bus SPI come già ampliamente riferito.

Tali operazioni sono riportate brevemente nel seguito.

Lettura di un indirizzo short

Per effettuare la lettura di un registro di tipo short è necessario:

- abbassare la linea CS per segnalare l'inizio di una trasmissione
- caricare nel buffer SPI un byte che abbia come bit più significativo uno `0` e come meno significativo un altro `0`; nei 6 bit centrali fra i due zeri è presente l'indirizzo del registro
- caricare nel buffer SPI un byte qualsiasi
- leggere nel buffer SPI il valore del registro desiderato
- alzare la linea CS per segnalare la fine della trasmissione

Il codice che realizza questa operazione è contenuto nella funzione
`unsigned char PHYGetShortRAMAddr(unsigned char address)`
nel file `mrf24j40.c` nella directory `radio_driver`. Il parametro della funzione è l'indirizzo che si desidera leggere e il valore di ritorno il relativo contenuto.

In figura 7.2 è presente lo screenshot eseguito con l'oscilloscopio di tale operazione . Come si noterà in figura, sulla linea SDI transita il valore del registro richiesto, in concomitanza con la seconda sequenza di fronti d'onda della linea SCK: la trasmissione del secondo byte.

Scrittura di un registro short

Similmente alla lettura, è necessario

- abbassare la linea CS per segnalare l'inizio di una trasmissione
- caricare nel buffer SPI un byte che abbia come bit più significativo uno `0` e come meno significativo un `1`; nei 6 bit centrali fra lo zero e l'uno è presente l'indirizzo del registro
- caricare nel buffer SPI il valore che si intende dare al registro
- alzare la linea CS per segnalare la fine della trasmissione

Il codice è contenuto nella funzione
`void PHYSetShortRAMAddr(unsigned char address, unsigned char value)`
nel file precedentemente menzionato. I parametri sono il registro che si intende scrivere e il valore che gli si vuole attribuire.

Nella figura 7.3 è visibile il risultato di una scrittura di un registro corto. La linea SDI rimane muta.

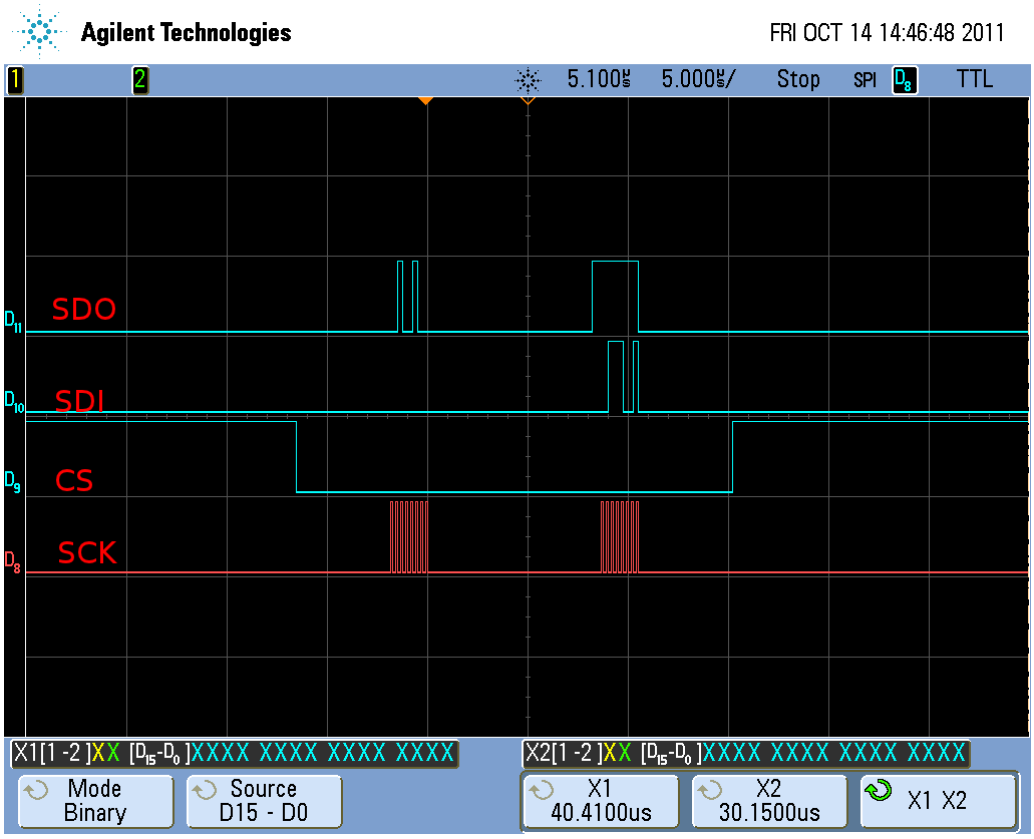

Figura 7.2: Lettura di un registro short via SPI

Lettura di un registro long

Concettualmente le operazioni di lettura e scrittura dei registri long non sono diverse da quelle dei registri short. È necessaria solo qualche accortezza in più vista la taglia (10 bit) dei registri in esame. Dal punto di vista operativo, si deve:

- abbassare la linea CS per segnalare l'inizio di una trasmissione
- caricare nel buffer SPI un byte che abbia come bit più significativo un 1 i 7 bit più significativi dell'indirizzo del registro
- caricare nel buffer SPI un byte che abbia nei tre bit più significativi, i tre bit meno significativi dell'indirizzo del registro, seguiti poi da uno 0. Gli ultimi 4 bit possono essere qualsiasi valore
- caricare nel buffer SPI un byte qualsiasi

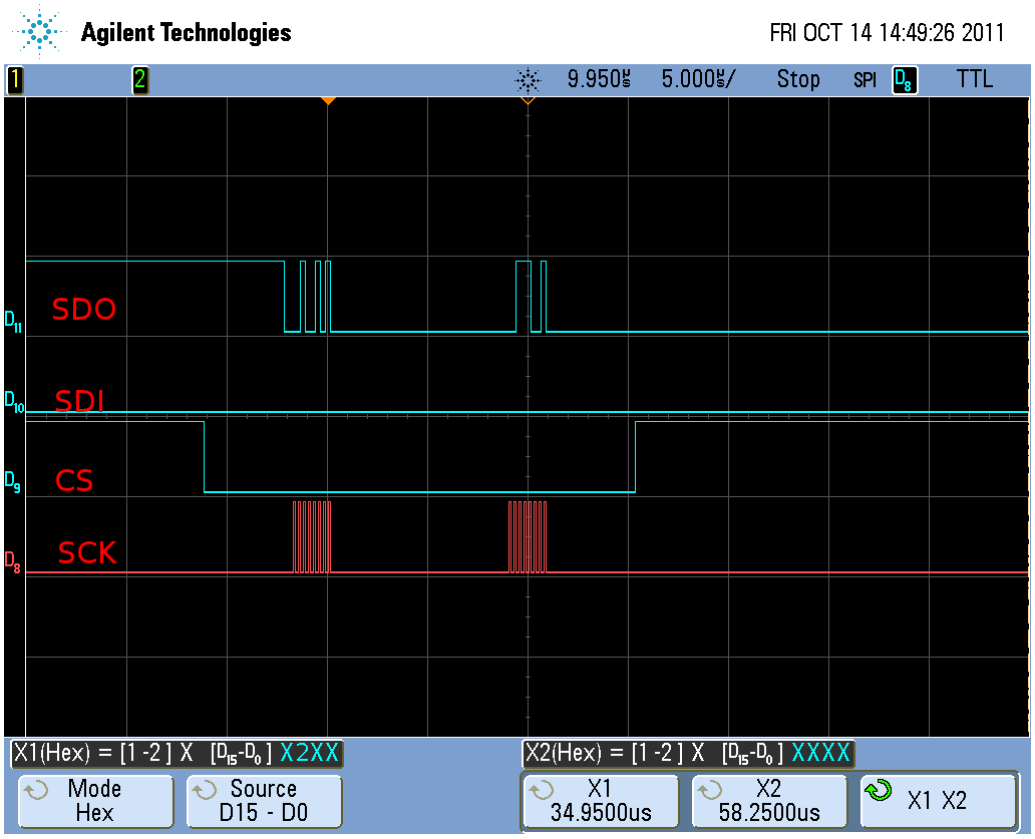

Figura 7.3: Scrittura di un registro short via SPI

- leggere nel buffer SPI il valore del registro desiderato
- alzare la linea CS per segnalare la fine della trasmissione

Il codice è contenuto nella funzione
`unsigned char PHYGetLongRAMAddr(int address)`.
Il significato dei parametri è lo stesso che per la versione short. Nella figura 7.5 è visibile il risultato della trasmissione mediante oscilloscopio. Come si vede in figura, la linea SDI contiene il valore del registro solo in concomitanza con l'invio sulla linea SDO del terzo byte.

Scrittura di un registro long

Similmente alla lettura, si procede ad:

- abbassare la linea CS per segnalare l'inizio di una trasmissione

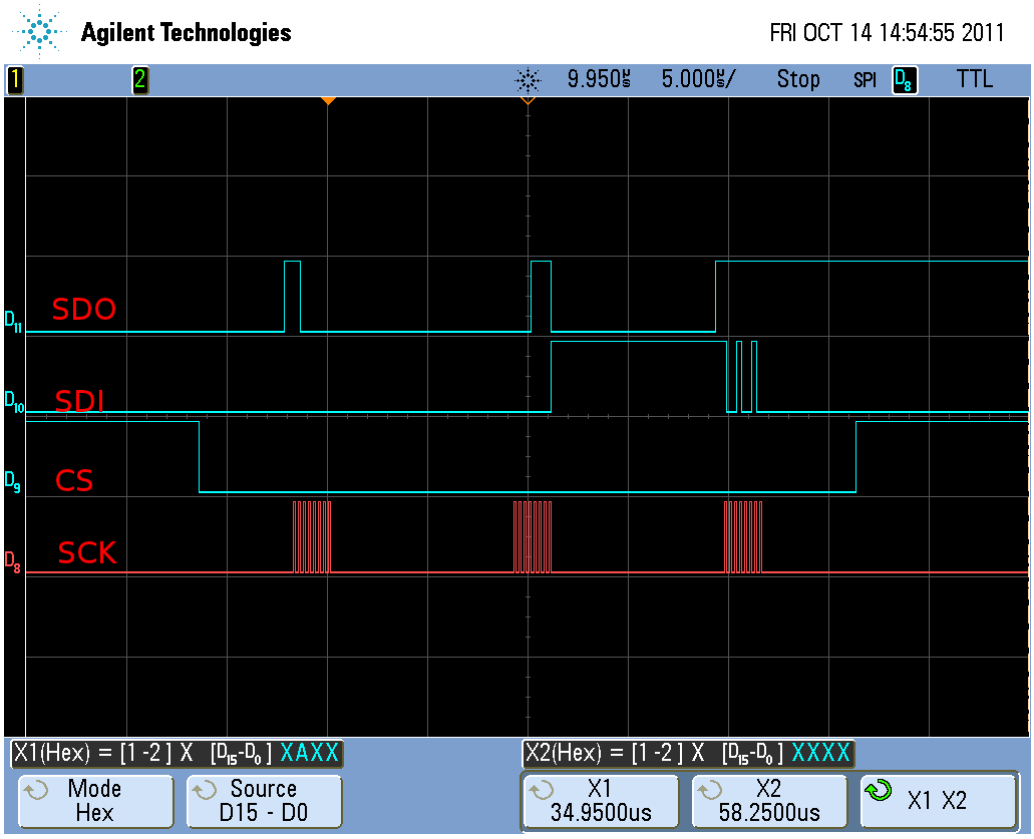

Figura 7.4: Lettura di un registro long via SPI

- caricare nel buffer SPI un byte che abbia come bit più significativo un 1 i 7 bit più significativi dell'indirizzo del registro
- caricare nel buffer SPI un byte che abbia nei tre bit più significativi, i tre bit meno significativi dell'indirizzo del registro, seguiti poi da uno 1. Gli ultimi 4 bit possono essere qualsiasi valore
- caricare nel buffer SPI il valore che si intende dare al registro
- alzare la linea CS per segnalare la fine della trasmissione

Il codice che implementa la precedente procedura è contenuto in:
`void PHYSetLongRAMAddr(unsigned int address, unsigned char value)`.

Visibile in figura 7.5 il risultato della trasmissione sull'oscilloscopio. Come nel caso di una scrittura di un registro short, la linea SDI rimane immobile.

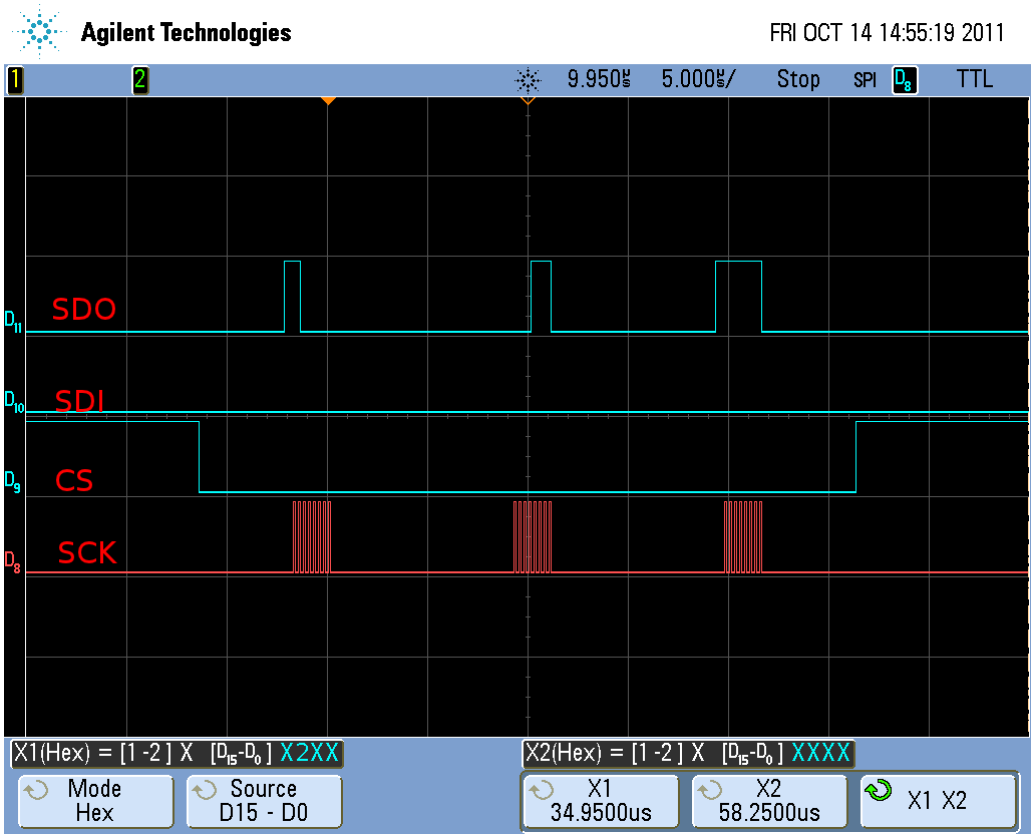

Figura 7.5: Scrittura di un registro long via SPI

Utilizzando opportunamente le funzioni presentate è possibile ricevere ed inviare pacchetti 802.15.4; vediamo più nel dettaglio come.

Invio e ricezione di un pacchetto

Invio e ricezione di un pacchetto

Come si ricorderà dal capitolo che tratta dell'implementazione del 1451, sono previsti all'interno del codice due buffer: uno destinato a contenere i messaggi in uscita e uno per i messaggi in entrata. I messaggi contengono i comandi 1451.0 e le relative risposte. Illustriamo dunque come il codice del 1451.0 si interfaccia al driver del MRF24J40.

Le funzioni che realizzano l'invio e la ricezione sono:

- `SendPacket()` copia il pacchetto da inviare dal buffer dei messaggi in uscita del uC al buffer `TXFIFO` del transceiver; dispone poi l'invio

- `ReceivePacket()` copia il pacchetto ricevuto dal buffer `RXFIFO` al buffer dei messaggi in arrivo del uC

Tali primitive sono contenute nel file `mrf24j40.c`. Le due funzioni precedenti contengono semplicemente invocazioni opportune delle quattro funzioni precedentemente illustrate.

Ultima precisazione sull'invocazione delle due funzioni precedenti. La funzione `SendPacket()` deve essere esplicitimente invocata all'interno del codice qualora si intenda inviare il pacchetto contenuto nel buffer in uscita.

La funzione `ReceivePacket()` è invocata dalla interrupt service routine che si occupa di gestire l'interrupt inviato dal transceiver al microcontrollore.

Per ulteriori dettagli si veda il datasheet [Mic10].

Capitolo 8

Risultati sperimentali

In questo capitolo viene presentato l'environment di test che ha consentito di effettuare un semplice collaudo delle funzionalità del nodo TIM.

Per collaudare il nodo TIM contenente il codice fin quì realizzato, è stato realizzato del codice di prova che è stato inserito in un altra scheda PicdemZ dotata dello stesso hardware fin quì presentato.

Environment di test

Il codice realizzato propone una semplice interfaccia seriale interattiva che ci permette di inviare fino a 5 diversi messaggi 1451.0 al nodo TIM.

Il nodo TIM, come previsto, risponde correttamente alle richieste ricevute; queste ultime sono state analizzate mediante la scheda Zena e mediante l'output fornito in seriale dalla scheda stessa.

8.1 Environment di test

Più nel dettaglio, l'environment è costituito da:

PicdemZ - TIM
: contenente il codice presentato in questo lavoro; essa è connessa via seriale ad un notebook mediante l'adattatore USBtoSerial e l'output viene visualizzato all'interno del software console Termite

PicdemZ - Tester
: contenente FreeRTOS con un semplice task che si occupa di stampare un menu in console seriale, accettare le scelte dell'utente e inviare al nodo TIM il comando 1451.0 richiesto. Il nodo Tester è connesso allo stesso notebook in modo identico

Zena Analyzer Boar
: impostata sul canale 20 del protocollo 802.15.4, per l'analisi del traffico di rete

Da notarsi inoltre che entrambe le schede sono connesse via seriale nella modalità 9600 baud, 8N1.

Configurazione layer MAC

Dal punto di vista del layer MAC del protocollo 802.15.4, i nodi sono stati così configurati:

PicdemZ - TIM

PAN ID 0x0005
MAC ID 0x0001

PicdemZ - Tester

PAN ID 0x0005
MAC ID 0x0002

Presentiamo ora l'interfaccia di testing visibile all'utente finale.

8.2 Interfaccia utente

Interfaccia seriale

Sul terminale seriale del nodo tester è visibile il seguente menu:

```
X
RI
NCAP
a-<11>
s-<21>
d-<31>
f-<41>
g-<51>
?
```

Il significato di quanto presentato è molto semplice. La `X` indica il fatto che la scheda si è avviata correttamente; `RI` indica che la il transceiver radio è inazializzato ed è pronto a trasmettere/ricevere.

Seguono poi le scelte disponibili all'utente: Il primo carattere indica il carattere da digitare da tastiera e fra parentesi angolate il pacchetto che verrà inviato

al nodo TIM. La prima cifra rappresenta la classe di comando 1451.0 mentre la seconda è l'indentificativo della funzione.

Sostanzialmente, digitando il carattere a in console seriale, seguito dal tasto `Enter`, si ordinerà al nodo Tester di inviare al nodo TIM il messaggio 1451.0 corrispondente alla classe comando 1 (ovvero *Commands common to the TIM and Transducer Channel*, tabella 15 di [14507a]) e specificamente il comando 1 (ovvero *Query TEDS*, tabella 16 di [14507a]).

TIM idle serial output

Prima della ricezione di messaggi 1451.0, sul nodo TIM saranno invece visualizzate continuamente sequenze nella forma:

```
...
-T0[n]
-T1[n]
-T2[n]
-T3[n]
-T4[n]
....
-T0[n]
-T1[n]
-T2[n]
-T3[n]
-T4[n]
...
```

L'output della scheda TIM, che ad una prima occhiata può sembrare un po' criptico in realtà è abbastanza semplice: ciascuno dei task che compongono l'implementazione del 1451 è dotato di un identificativo numerico, inserito all'interno dell'header file `1451common.h`. I task sono dotati dei seguenti codici identificativi:

0 Task IntMng

1 Task CommMng

2 Task CmdInt

3 Task CmdEx

4 Task TEDSMng

Ora il log della seriale appare molto più chiaro: `-T1[]` significa che è andato in esecuzione il Task CommMng e che non aveva messaggi in coda da processare (`n` all'interno delle parentesi quadre).

Se non viene inviato un messaggio 1451.0 l'output visualizzato nei terminali seriali non sarà mai diverso da quanto presentato.

8.3 Esecuzione di un test

Se all'interno del terminale seriale del nodo Tester, viene premuto un carattere fra quelli previsti, ovvero `a, s, d, f, g` seguito dal tasto `Enter`, il menu della seriale visualizza:

```
Ex:f
PSE:f

NCAP
a-<11>
s-<21>
d-<31>
f-<41>
g-<51>
?
```

Nell'esempio il tasto premuto era `f`. L'output restituito significa:

Ex:f esecuzione della voce di menu `f`

PSE:f pacchetto inviato ed esecuzione di `f` completata

Dopodichè viene riproposto all'utente nuovamente il menu in modo da poter inviare un altro pacchetto al nodo TIM. Abbiamo ora inviato il messaggio 1451.0 al nodo TIM con classe comando 4 e identificativo comando 1, ovvero *Transducer Channel Operate* (tabella 34 di [14507a]).

Output TIM in fase di processing

L'output fornito dal nodo TIM, non appena ricevuto il pacchetto è più interessante:

```
...
-T3[n]
-T4[n]
-T0[n]
-T1[r<1|1|0>RX::52::161::PPS]
-T2[n]
-T3[n]
-T4[n]
-T0[r<1|0|0>md]
-T1[r<1|1|1>TX]
-T2[r<0|2|0>CC4CF1->]
-T3[r<0|3|19>TCOSR]
-T4[n]
```

```
-T0[n]
-T1[r<1|1|1>TX]
-T2[n]
-T3[n]
-T4[n]
...
```

Il task 1, ovvero il CommMng segnala di aver ricevuto un pacchetto 802.15.4, di 52 byte e la cui potenza di trasmissione era 161, ovvero circa -56 dBm. Per le tabelle di equivalenza fra potenza segnalata dal transceiver e potenza elettrica equivalente si veda il datasheet del transceiver [Mic10]. Il tutto è leggibile nella riga:
`-T1[r<1|1|0>RX::52::161::PPS]`.
Gli ultimi 3 caratteri segnalano il fatto che il CommMng ha anche inviato un pacchetto al nodo Tester per segnalare la reale comprensione del pacchetto ricevuto. Tale pacchetto non è previsto dallo standard ma è utilizzato in questa semplice applicazione di prova per fornire più informazioni possibili all'utente.

Elaborazione di un messaggio 1451

A questo punto il pacchetto è dispobile al protocollo 1451 sul nodo TIM che si mette in azione per comprendere il pacchetto ricevuto, eseguirlo e inviare la risposta al nodo Tester.
`T0`, ovvero il task IntMng, avvisato dal CommMng segnala che è presente un messaggio e che deve essere inoltrato all'interprete comandi per la decodifica: `md` significa infatti *message dispatched*.

Ricompare poi il task CommMng che segnala di aver inviato correttamente il pacchetto menzionato sopra: segnala il fatto mediante la dicitura `TX`, ovvero *transmitted*.

Visibile poi l'output del task CmdInt, cioè `T2`. Quest'ultimo sollecitato dal task `T0` effettua la decodifica del pacchetto ricevuto e produce l'output `CC4CF1->`. La classe comando (`CC`) compresa dal CmdInt è quella corretta: 4; il codice della funzione (`CF`) è corretto anch'esso: 1.

Non essendo sua competenza eseguire quello specifico comando, esso lo inoltra ulteriormente e segnala il fatto mediante una sorta di freccia `->`.

Avvisato dal task CmdInt, il task `T3` ovvero il task CmdEx riceve notifica di dover eseguire un comando e produce l'ouput:
`TC0`, ovvero un acronimo che sta per *transducer channel operate*. Dopo aver seguito il comando invia un ulteriore pacchetto per segnalare l'effettiva esecuzione del comando: `SR`, cioè *sent reply*. La tabella degli acronimi di output per ciascuno dei comandi dello standard è definita nel file `basic1451.c`.

Nuovamente il CommMng segnala di aver inviato il pacchetto mediante la stringa `TX`.

Traffico radio

Ecco dunque un esempio completo di esecuzione di un comando 1451.0 sul

nodo TIM. Per completare il test è stato acquisito il traffico 802.15.4 mediante

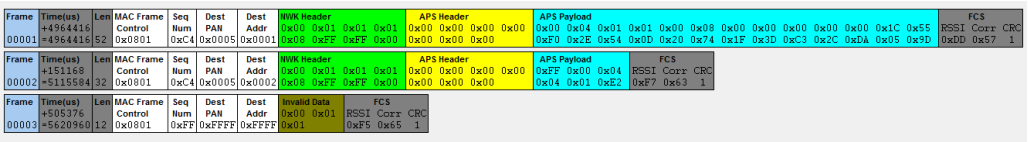

Figura 8.1: Acquisizione mediante ZENA del traffico dell'esempio di test presentato

la scheda Zena. In figura 8.1 è visibile l'acquisizione dei pacchetti menzionati nell'esecuzione:

- il pacchetto dal nodo Tester al nodo TIM
- il pacchetto di conferma dal nodo TIM al nodo Tester
- il pacchetto conclusivo dal nodo TIM al nodo Tester

Il test su tutti i comandi ha dato esito positivo. Questo primo collaudo fa pensare che la struttura complessiva del codice sia solida e performante.

Conclusioni e sviluppi futuri

I risultati ottenuti per via sperimentale provano l'effettivo funzionamento dell'architettura software del sistema realizzato.

Al momento odierno è stata quindi realizzata una implementazione, seppure parziale (non supportando tutti i comandi) dello standard 1451.0 lato TIM su architettura PIC18.

Il nodo è equipaggiato del software in grado di dargli l'abilità di ricevere comandi 1451.0 provenienti dalla rete wireless, e di replicare a questi ultimi in modo coerente. È in grado di amministrare l'automa di funzionamento del nodo, e quello dello stato del Transducer Channel. È capace di acquisire campioni dal sensore di temperatura in maniera essenziale ma corretta. Ha la possibilità di archiviare al suo interno i quattro TEDS obbligatori previsti dallo standard ed è in grado di trasmetterli su richiesta di un potenziale nodo NCAP.

Tale software è inoltre strutturato in modo da essere una libreria per FreeRTOS: tale fatto la rende utilizzabile su tutte le altre architetture supportate dal RTOS, *mutatis mutandis*.

Ciò nonostante è opportuno precisare come questo lavoro di tesi sia una prima implementazione di uno standard la cui complessità è fuori discussione e pertanto, a tal proposito, molte sono le strade sulle quali poter continuare quanto realizzato.

Va ricordato che obbiettivo più ampio del progetto in cui questo lavoro si inserisce è la realizzazione di un vero e proprio network 1451. Tale fatto comporta obbligatoriamente la realizzazione di un nodo NCAP, magari sfruttando le schede Explorer 16, ben più dotate della PicdemZ. Tale scelta comporta l'adattamento del codice realizzato verso una architettura leggermente diversa, ovvero la Microchip® PIC24.

Oltre a ciò bisogna ricordare che lo scenario descritto dallo standard 1451.0 prevede anche la possibilità di monitorare e interrogare i nodi TIM mediante API HTTP presenti sul nodo NCAP. Tale fatto origina un ulteriore interessante lavoro di intefacciamento del nodo NCAP mediante rete ethernet 802.3.

È possibile inoltre aumentare i servizi di rete offerti dal nodo TIM integrando una porzione più ampia dello stack Microchip® Zigbee o cimentarsi in soluzioni

differenti basate sul nascente e molto promettente protocollo 6LoWPAN.

Doveroso precisare come sembri assai interessante l'interfacciamento hardware del nodo TIM con un sensore più sofististicato del TC77 in modo da sperimentare le ampie possibilità fornite dallo standard 1451.0

Bibliografia

[14507a] IEEE Standard for a Smart Transducer Interface for Sensors and Actuators - Common Functions, Communication Protocols, and Transducer Electronic Data Sheet (TEDS) Formats. *IEEE Std 1451.0-2007*, pages 1 –335, 21 2007.

[14507b] IEEE Standard for a Smart Transducer Interface for Sensors and Actuators Wireless Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats. *IEEE Std 1451.5-2007*, pages C1 –236, 5 2007.

[Bar09a] R. Barry. *FreeRTOS reference manual: API functions and configuration options*. Real Time Engineers Ltd., 2009.

[Bar09b] R. Barry. *Using the FreeRTOS real time kernel: a practical guide*. Real Time Engineers Ltd., 2009.

[Ber08] Filippo Bergamin. Procedure di interfacciamento di sensori wireless secondo gli standard ieee 1451.5 e zigbee. Master's thesis, Università degli Studi di Padova, 2008.

[BL04] S. Bible and J. Lepkowski. *AN913 - Interfacing the TC77 Thermal Sensor to a PICmicro® Microcontroller*. Microchip Technology Inc., 2004. DS00913A.

[Bri88] Brian W. Kernighan; Dennis M. Ritchie. *The C Programming Language (2nd ed.)*. Englewood Cliffs, NJ: Prentice Hall, 1988.

[Bru11] Brundo, S. FreeRTOS PIC18F4620 porting, 2011. `https://github.com/salvix/pic18f4620_free_rtos`.

[CFM11] G. Clemente, F. Filira, and M. Moro. *Sistemi operativi*. Progetto Libreria, 2011.

[FOR06] D. Flowers, K. Otten, and N. Rajbharti. *AN965 - Microchip Stack for the ZigBee™ Protocol*. Microchip Technology Inc., 2006. DS00965B.

[HMU07] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to automata theory, languages, and computation*. Pearson/Addison Wesley, 2007.

[Hua05] H.W. Huang. *PIC microcontroller: an introduction to software and hardware interfacing*. Thomson/Delmar Learning, 2005.

[Ibr08] D. Ibrahim. *Advanced PIC microcontroller projects in C: from USB to RTOS with the PIC18F series*. Electronics & Electrical. Newnes/Elsevier, 2008.

[Liu00] J.W.S. Liu. *Real-Time systems*. Prentice Hall, 2000.

[Mic02] Microchip Technology Inc. *TC77 - Thermal Sensor with SPI™ Interface*, 2002. DS20092A.

[Mic05a] Microchip Technology Inc. *MPLAB®C18 C COMPILER LIBRARIES*, 2005. DS51297F.

[Mic05b] Microchip Technology Inc. *MPLAB®C18 C COMPILER USER'S GUIDE*, 2005. DS51288J.

[Mic05c] Microchip Technology Inc. *MPLAB®C18 C COMPILER GETTING STARTED*, 2005. DS51295F.

[Mic06a] Microchip Technology Inc. *MPLAB®IDE User's Guide*, 2006. DS51519B.

[Mic06b] Microchip Technology Inc. *ZENA™ Wireless Network Analyzer User's Guide*, 2006. DS51606A.

[Mic07a] Microchip Technology Inc. *MPLAB®ICD 2 In-Circuit Debugger User's Guide*, 2007. DS51331C.

[Mic07b] Microchip Technology Inc. *PIC18Cxxx Configuration Settings*, 2007. PIC18F4620.

[Mic08a] Microchip Technology Inc. *MRF24J40MA Data Sheet - 2.4 GHz IEEE Std. 802.15.4™ RF Transceiver Module*, 2008. DS70329B.

[Mic08b] Microchip Technology Inc. *PIC18F2525/2620/4525/4620 Data Sheet - 28/40/44-Pin Enhanced Flash Microcontrollers with 10-Bit A/D and nanoWatt Technology*, 2008. DS39626E.

[Mic08c] Microchip Technology Inc. *PICDEM™Z Demonstration Kit User's Guide*, 2008. DS51524C.

[Mic09a] Microchip Technology Inc. *MPASM™ Assembler, MPLINK™ Object Linker MPLIB™ Object Librarian User's Guide*, 2009. DS33014K.

[Mic09b] Microchip Technology Inc. *MPLAB® IDE User's Guide with MPLAB Editor and MPLAB SIM Simulator* , 2009. DS51519C.

[Mic09c] Microchip Technology Inc. *PIC18F Peripheral Library Help Document*, 2009.

[Mic10] Microchip Technology Inc. *MRF24J40 Data Sheet - IEEE 802.15.4™ 2.4 GHz RF Transceiver*, 2010. DS39776C.

[MM09] S. Myneni and T. Manolescu. *AN1192 - MRF24J40 Radio Utility Driver Program*. Microchip Technology Inc., 2009. DS01192B.

[TS07] A.S. Tanenbaum and M. Steen. *Distributed systems: principles and paradigms*. Pearson Prentice Hall, 2007.