UNIVERSITÁ DEGLI STUDI DI PADOVA

FACOLTÀ DI INGEGNERIA ELETTRONICA ED INFORMATICA

CORSO DI LAUREA IN INGEGNERIA ELETTRONICA

REALIZZAZIONE DI UN GATEWAY TCP/IP - ZIGBEE PER RETI DI SENSORI WIRELESS (STANDARD IEEE 1451)

LAUREANDO:

Pierpaolo Russo

RELATORE:

Chiar.mo Prof. Claudio Narduzzi

Padova, 26 Aprile 2010

Anno Accademico 2009/2010

UNIVERSITÁ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

TESI DI LAUREA

REALIZZAZIONE DI UN GATEWAY TCP/IP - ZIGBEE PER RETI DI SENSORI WIRELESS (STANDARD IEEE 1451)

LAUREANDO: Pierpaolo Russo

RELATORE: Chiar.mo Prof. Claudio Narduzzi

CORSO DI LAUREA IN INGEGNERIA ELETTRONICA

Padova, 26 Aprile 2010

Anno Accademico 2009/2010

"Brick walls are there for a reason.

They let us prove how badly we want things."

Dr. Randolph Frederick "Randy" Pausch

(October 23, 1960 - July 25, 2008)

...ad Eleonora

Sommario

Scopo di questa tesi é dimostrare la realizzabilitá di un gateway TCP/IP - ZigBee su microcontrollore Microchip, utilizzando gli strumenti software gratuiti rilasciati dallo stesso costruttore.

Sebbene infatti siano presenti sul mercato diverse soluzioni commerciali che permettono l'interfacciamento tra reti cablate di personal computer e reti di sensori wireless basate su protocollo di comunicazione ZigBee, non si é trovata alcuna documentazione riguardante un dispositivo a basso costo, altamente flessibile e completamente personalizzabile, come potrebbe essere quello basato su hardware e software Microchip.

Si é proceduto per fasi, dapprima verificando l'effettivo funzionamento della scheda di sviluppo come NCAP in una comunicazione ZigBee con un sensore wireless, poi verificando le potenzialitá di un web server integrato ed infine unendo le due fasi, con l'auspicio che chi proseguirá il lavoro possa arrivare a controllare la comunicazione in una rete di sensori mediante interfaccia web.

Indice

Elenco delle figure

Elenco dei Listati

Capitolo 1

Introduzione agli standard 1451

La realizzazione di una rete di trasduttori secondo uno standard unico dá molteplici vantaggi, tra i quali:

- La possibilitá di definire una rete di trasduttori non vincolata a case produttrici.
- Avere un modello comune per la gestione dei dati provenienti dai trasduttori, facilitando cosí le operazioni di controllo, configurazione e calibrazione dei trasduttori stessi.
- L'installazione, l'aggiornamento, la sostituzione o la rimozione dei trasduttori nella rete in modalitá plug & play.
- La possibilitá di definire per ciascun trasduttore un data sheet elettronico contenente tutte le informazioni necessarie per gestire il trasduttore stesso.
- L'accesso ai trasduttori sia in modalitá wired che wireless, attraverso una vasta scelta di mezzi fisici.

Questa considerazioni hanno dato vita alla famiglia di standard IEEE 1451, un insieme di documenti elaborati negli ultimi dieci anni dall'Institute of Electric and Electronic Engineers, che prevede la definizione di un'interfaccia standard per le reti di sensori intelligenti. Questa famiglia di standard definisce un'architettura di base della rete, che consente di modificarne la configurazione in modalitá plug and play e definisce le modalitá d'accesso

ad una rete di trasduttori differenti per mezzo di comandi e procedure standardizzati. Prevede inoltre l'aggiunta di data sheets elettronici, i TEDS (Transducer Elestronic Data Sheet), contenenti le informazioni relative al trasduttore stesso, costruttore, range di misura, accuratezza, dati di calibrazione e molto altro. La direttiva proposta considera un modello di dispositivo programmabile ed indipendente dalla rete a cui é collegato (1451.1), dotato di un'interfaccia digitale (1451.2) e di un protocollo di comunicazione che consente di accedere al trasduttore via microprocessore. Una volta definite le caratteristiche principali delle reti 1451, l'IEEE ha definito in maniera piú dettagliata le caratteristiche dei singoli nodi, classificandoli in due distinte categorie:

TIM (Transducer interface module): dispositivi a cui spetta la gestione del sensore/trasduttore/attuatore, il condizionamento dell'informazione rilevata, la conversione di tali informazioni e la trasmissione delle stesse. Sono considerati dei nodi terminali di rete e si appoggiano ai nodi della seconda categoria: gli NCAP.

NCAP (Network Capable Application Processor): dispositivi ai quali spetta la gestione della rete e l'inoltro di messaggi che circolano in essa. Tali dispositivi operano da Gateway tra gli utenti della rete ed i TIM, nel caso fosse previsto l'accesso da esterno. Questa famiglia non definisce l'interfaccia fisica tra l'NCAP e la rete, ma i tipi di interfaccia che possono esistere tra l'NCAP e i TIM.

In questo capitolo verranno presentati ed analizzati i protocolli dello standard IEEE 1451 che sono stati studiati per la realizzazione del progetto. Per comprendere l'organizzazione dei protocolli della famiglia IEEE 1451 é utile osservare la figura 1.1. A partire da sinistra, ovvero dall'NCAP, si incontra prima l'applicazione lato utente, la quale si interfaccia al livello 1451.0 tramite una Application API (AAPI). Il livello 1451.0 si interfaccia a sua volta con il sottostante 1451.5 tramite una Communication API (CAPI). Il 1451.5 invece é collegato ad un livello fisico da esso supportato (nel caso di questo progetto ZigBee). Proseguendo sempre verso destra, in maniera speculare, si incontrano il livello fisico (ZigBee), il livello 1451.5 e il livello 1451.0 del WTIM, interfacciati tra di loro ancora tramite CAPI e AAPI rispettivamente. Infine, il livello 1451.0 del WTIM, tramite una

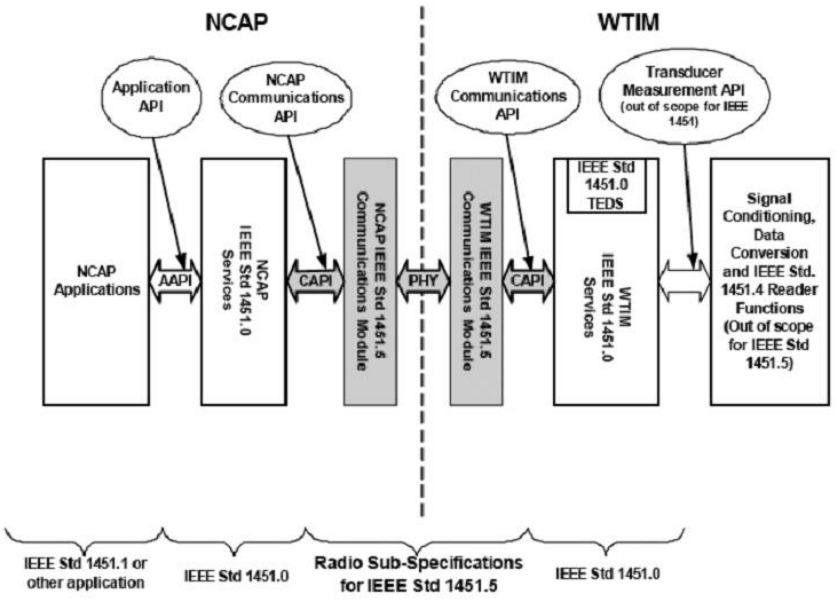

Figura 1.1: Modello di riferimento per le comunicazioni NCAP-WTIM

Transducer Measurement API, é connesso al trasduttore (il quale puó implementare il protocollo 1451.4, della stessa famiglia).

1.1 Lo standard 1451.5

Lo standard IEEE 1451.5 (Si veda [2]) introduce il concetto di Wireless Interface Transducer Module (WTIM) connesso ad un Network Capable Application Processor (NCAP) Service Module attraverso un canale radio approvato tra IEEE 802.11, IEEE 802.15.4, IEEE Bluetooth e ZigBee. Secondo le direttive 1451.5, un WTIM é un dispositivo che comprende una Dot 5 Approved Radio (Dot5AR), il condizionamento del segnale, la conversione analogica/digitale e uno o piú trasduttori (sensori/attuatori). Poiché i WTIM possono avere interfacce Dot5AR diverse tra loro, l'NCAP dovrá avere almeno una Dot5AR per ogni tipo presente nei WTIM a cui deve essere associato. Questo standard si focalizza sull'interfaccia di comunicazione tra WTIM ed NCAP attraverso i protocolli Dot5AR, stabilendo di fatto i metodi e i formati di dati necessari a governare i trasduttori, per le operazioni di rete e per i Transducer Electronic Data Sheets (TEDS). Lo standard in questione é basato unicamente su interfacce wireless, ma non specifica le caratteristiche fisiche e tecniche né

dei trasduttori né del sistema wireless. La nascita di questo protocollo é dovuta alla volontá di uniformare le specifiche ed accomunare piú tecnologie sotto un unico standard aperto, riducendo al minimo il rischio di incompatibilitá. Le specifiche dell'IEEE 1451.5 riguardanti ZigBee indicano i requisiti che una rete di tale tipo deve avere affinché possa fungere da rete di trasporto per un sistema compatibile con IEEE 1451.

1.1.1 Convergence Layer

Lo standard IEEE 1451.5 definisce il concetto di strato di convergenza tra l'entitá superiore (IEEE 1451.0) e la rete di trasporto(in questo caso ZigBee). Esso ha infatti il compito di tradurre i comandi provenienti dal livello superiore in comandi specifici comprensibili al livello inferiore e viceversa. Nella documentazione del protocollo 1451.5 sono elencati solamente i metodi della Communication API tra 1451.5 e 1451.0 (appartenenti alla MCI del 1451.0). Per ciascuno di essi, assieme ad una descrizione sintetica che ne spiega l'utilizzo in ambito 1451.5, vengono elencati nome, tipi di dato e parametri; questi metodi costituiscono il confine superiore del convergence layer.

1.1.2 Regole generali

- Un NCAP puó instradare dati sia verso una rete esterna, sia verso un trasduttore collegato ad un WTIM.

- Un NCAP puó avere piú WTIM associati.

- Un NCAP puó avere piú interfacce radio (anche diverse).

- L'interfaccia ZigBee é gestita dal protocollo IEEE 1451.5 sia per gli NCAP che per i WTIM.

- Un WTIM puó associarsi ad un solo NCAP.

- Un WTIM puó interfacciarsi a piú trasduttori.

- É ammessa la comunicazione tra due WTIM.

1.1.3 Diagramma di stato dell' NCAP

Lo stato iniziale di un NCAP dopo l'accensione o dopo il reset 'e UNREGISTERED. Dopo un processo di registrazione del modulo 1451.5, l'NCAP passa allo stato DOT5REGISTERED per l'entitá DOT5AR del protocollo 1451.5 e vi rimane finché non vi si associa alcun WTIM. L'NCAP ha il compito di mantenere, separatamente, lo stato di ognuna delle sue entitá 1451.5, le quali a loro volta, devono mantenere separatamente gli stati di tutti i loro WTIM associati. Quando un'entitá 1451.5 registra uno o piú WTIM, l'NCAP passa allo stato TIM-REGISTERED per tale modulo. Prima di iniziare uno scambio dati con un WTIM, l'NCAP deve invocare un comando di tipo open, con il quale passa appunto allo stato OPEN per tale WTIM e vi rimane fino al primo comando di tipo close, attraverso il quale ritorna allo stato TIMREGISTERED. Il vantaggio di mantenere separati gli stati per ogni entitá 1451.5 e per ogni WTIM consiste nel fatto che esso puó iniziare in qualsiasi momento la ricerca di altri dispositivi e l'attesa di nuove connessioni.

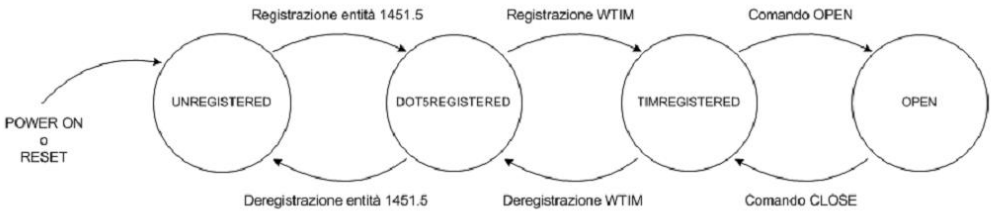

Figura 1.2: Diagramma di stato dell'NCAP

1.1.4 Diagramma di stato del WTIM

Lo stato iniziale di un WTIM dopo l'accensione o dopo il reset é UNREGISTERED. Dopo il processo di registrazione presso un NCAP, il WTIM passa allo stato REGISTERED ma non puó comunicare fintanto che non invoca un comando di tipo open, con il quale passa appunto allo stato OPEN e vi rimante fino al primo comando di tipo close, attraverso il quale ritorna allo stato REGISTERED.

Figura 1.3: Diagramma di stato del WTIM

1.2 Lo standard 1451.0

Lo standard IEEE 1451.0 (Si veda a [1]) si posiziona, nello stack protocollare, tra l'applicazione ed il protocollo di comunicazione scelto in base all'interfaccia fisica utilizzata, fungendo quindi a sua volta da convergence layer tra l'applicazione e la rete. Ovviamente anche questo protocollo riconosce i due dispositivi previsti dalla famiglia IEEE 1451 (NCAP e TIM) e si occupa di descriverne tutte le caratteristiche comuni, nonché le operazioni e le funzionalitá che caratterizzano i dispositivi di una rete di sensori intelligenti, indipendentemente dal tipo di interfaccia fisica prescelta.

Conformitá

Per ottenere la dicitura plug-and-play, un dispositivo di questo tipo deve rispondere a determinate caratteristiche richieste a livello applicazione. Esse sono tutte specificate nella documentazione dello standard, ma in particolare si ricorda che sia gli NCAP che i TIM funzionanti secondo lo standard IEEE 1451.0 devono supportare un protocollo di comunicazione e un mezzo fisico definiti nella stessa famiglia di standard IEEE 1451.

Rete

Lo standard 1451.0 non richiede alcun tipo specifico di rete fisica, pertanto la scelta é lasciata all'utente, ma necessita che l'NCAP sia dotato del software e dell'hardware appropriato per la gestione della tipologia di rete prescelta.

Indirizzamento

Esistono due tipi di indirizzamento definiti da questo standard. Il primo é di tipo fisico, del quale si occupa il physical layer, tramite il parametro destId (cosí riconosciuto dal Module Communication Interface). Il secondo parametro per l'indirizzamento é il TransducerChannelNumber.

API

Lo standard definisce una Application Program Interface (API) per tutte le applicazioni che provvedono alla comunicazione tra la rete e il layer IEEE 1451 ed alla comunicazione tra il protocollo IEEE 1451.0 e i sottostanti layer fisici di comunicazione, di solito denominati in questo standard come layer IEEE 1451.X. Lo standard 1451.0 definisce quindi due tipi di API: la prima é la Transducer Service Interface, API del solo NCAP, utilizzata dalle applicazioni di misura e controllo lato utente, per accedere al layer IEEE 1451.0. Questa API contiene i metodi per leggere e scrivere i TEDS, per gestire i TransducerChannels, per inviare comandi di configurazione e controllo ai TIM. L'altra API, la Module Communication Interface, si colloca tra lo standard 1451.0 e un altro membro della famiglia 1451. Essa é 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. Per mantenere una neutralitá di linguaggio, le funzioni e i parametri delle API sono descritte con il linguaggio Interface Definition Language (IDL).

Capitolo 2

Standard ZigBee

2.1 Introduzione

ZigBee é il nome di una specifica per un insieme di protocolli di comunicazione ad alto livello che utilizzano piccole antenne a bassa potenza, basato sullo standard IEEE 802.15.4 per Wireless Personal Area Networks (WPAN). ZigBee opera nelle frequenze radio dedicate a scopi industriali, scientifici e medici (ISM). Questa tecnologia si propone di essere piú semplice e piú economica di altre WPAN come, ad esempio, Bluetooth. Il protocollo ZigBee é stato ideato per rendere facile la sua implementazione su microprocessori a basso costo che offrono solamente le caratteristiche minime ed essenziali. Il progetto dei radiotrasmettitori é stato ottimizzato per avere un basso costo unitario con produzioni su larga scala. Essi hanno poca circuiteria analogica e ne utilizzano di digitale ovunque sia possibile. Per esempio, si valuta che un nodo ZigBee del tipo piú complesso richieda solamente il 10% del codice necessario per un tipico nodo Bluetooth o Wi-Fi, mentre il piú semplice dovrebbe richiederne intorno al 2%. Anche se i radiotrasmettitori sono economici, il ZigBee Qualification Process comporta una validazione completa delle richieste del livello fisico. Questa minuziosa analisi del livello fisico ha molti vantaggi poiché garantisce che tutti i trasmettitori derivanti da uno stesso set di semiconduttori abbiano le stesse caratteristiche RF. D'altra parte un livello fisico non certificato che presenta malfunzionamenti potrebbe influenzare negativamente le capacitá di una rete ZigBee. In questo caso, infatti, i vincoli ingegneristici sono necessariamente stretti, in termini di banda e di

consumo di energia.

2.2 Tipi di dispositivo

Le specifiche IEEE 802.15.4 definiscono due tipi di dispositivi:

Full Function Device (FFD): Offre piena funzionalitá, é solitamente alimentato da rete e rimane acceso anche quando inattivo.

Reduced Function Device (RFD): Offre funzionalitá limitata, é solitamente alimentato da batterie e rimane spento quando inattivo.

Le specifiche ZigBee definiscono invece tre tipi di dispositivi:

ZigBee Coordinator (ZC): É il dispositivo piú completo tra quelli disponibili, costituisce la radice di una rete ZigBee e puó operare da ponte tra piú reti. Ci puó essere un solo Coordinator in ogni rete. Esso é inoltre in grado di memorizzare informazioni riguardanti la propria rete e puó agire da deposito per le chiavi di sicurezza.

ZigBee Router (ZR): Questo dispositivo agisce come router intermedio passando i dati da e verso altri dispositivi.

ZigBee End Device (ZED): Include solo le funzionalitá minime per dialogare con un nodo Coordinator o Router, non puó trasmettere dati provenienti da altri dispositivi; é il nodo che richiede il minor quantitativo di memoria e quindi risulta spesso piú economico rispetto ai ZR o ai ZC.

Basandosi sulle definizioni proprie delle specifiche IEEE802.15.4, il protocollo ZigBee classifica cosí i suoi tre tipi di dispositivi:

Coordinator: Di tipo FFD, unico all'interno della rete; forma la rete e la gestisce, alloca e memorizza gli indirizzi di tutti i nodi collegati.

Router: Di tipo FFD, opzionale, estende la rete dando la possibilitá di aumentare il numero di nodi e di trasferire dati tra di essi; puó esercitare funzioni di controllo e monitoraggio.

End Device: Di tipo FFD o RFD, é tipicamente il dispositivo sensore o attuatore ed esercita funzioni di controllo e/o monitoraggio.

In questo specifico progetto poiché l'NCAP si occuperá della creazione, gestione e manutenzione della rete, sará costituito da un nodo di tipo ZigBee Coordinator (ZC), mentre il WTIM sará costituito da un nodo ZigBee End Device (ZED), limitato alla sola trasmissione dei dati provenienti dalla lettura del proprio sensore.

2.3 Tipi di rete

Una rete wireless ZigBee puó assumere diverse configurazioni. Comunque, in ogni rete, ci devono essere almeno due componenti: un nodo Coordinator ed un nodo End Device. Il Router invece é presente solo in alcuni tipi di rete. Le tre tipologie di reti realizzabili sono: rete a stella (star network), rete ad albero (cluster tree network) e rete a maglia (mesh network). Per questo progetto si é presa in considerazione unicamente la configurazione a stella, formata da un coordinatore e da uno o piú nodi terminali i quali non possono comunicare tra di loro, ma solo con il coordinatore.

2.4 Indirizzamento

Ogni nodo ZigBee ha, a livello di rete, due indirizzi: un MAC address a 64 bit (dei quali 24 bit identificano il produttore) ed un network address di 16 bit. Per stabilire una connessione con una nuova rete, il nodo utilizza l'indirizzo MAC, dopodich é, una volta connesso, verrá identificato nella rete attraverso il suo network address e tramite esso potrá comunicare con gli altri dispositivi. Per la messaggistica di tipo unicast viene utilizzato il MAC address specifico del nodo a cui ci si riferisce, mentre per il broadcast (impiegato nelle operazioni di gestione della rete quali creazione, connessione ecc.) si utilizza il MAC address generico 0xFFFF.

2.5 Sincronizzazione

Un beacon frame é un pacchetto contenente tutte le informazioni sulla rete e che viene trasmesso dal coordinatore della stessa. Zigbee supporta due tipologie di reti, beacon e non beacon.

- In una rete beacon enabled gli ZigBee Router e i Coordinator trasmettono periodicamente dei beacon frame per confermare la loro presenza agli altri nodi. La caratteristica importante di queste reti é che tra un beacon e l'altro i nodi possono entrare in risparmio energetico. Questa tipologia di rete é preferibile nei casi in cui il risparmio di energia é importante, tipicamente quando sono previsti dei nodi alimentati da una batteria. Questa tipologia di rete é peró di piú complessa gestione, richiedendo meccanismi di timing preciso, piú difficili e dispendiosi da realizzare.

- In una rete non-beacon enabled tutti i nodi sono costantemente attivi e perció molto piú semplici da gestire. Tuttavia ne consegue un superiore consumo energetico.

Poiché negli scopi del progetto non rientra il risparmio energetico, la tipologia di rete adottata é quella non-beacon. In una rete di tipo non-beacon un dispositivo che vuole inviare un messaggio deve semplicemente attendere che il canale sia libero, dopodiché puó iniziare la trasmissione. Se il dispositivo di destinazione é di tipo FFD il suo ricetrasmettitore sará sempre acceso e il messaggio verrá ricevuto immediatamente. Diversamente un RFD potrá avere il ricetrasmettitore spento (in modalitá risparmio energetico) al momento della ricezione, per cui, una volta acceso il ricetrasmettitore, dovrá appoggiarsi al suo FFD associato (genitore) richiedendo l'inoltro dei messaggi non ricevuti, il quale li avrá temporaneamente memorizzati in un buffer. Tale caratteristica permette all'RFD di risparmiare energia ma, al contempo,richiede all'FFD una sufficiente quantitá di memoria libera. Se dopo un certo tempo, chiamato macTransactionPersistenceTime, l'RFD non ha richiesto al suo FFD genitore l'inoltro dei messaggi ad esso riservati, essi verranno irreversibilmente scartati.

2.6 Profili

Un profilo ZigBee é una semplice descrizione dei componenti logici e delle loro interfacce. I dati da scambiare, per esempio le letture o le tarature dei sensori, sono chiamati attributi e ad ognuno di essi é associato un identificatore univoco. Gli attributi sono raggruppati in cluster ai quali, a loro volta, é associato un altro identificatore univoco. Di conseguenza le interfacce sono specificate a livello di cluster. I profili inoltre possono specificare quali cluster sono obbligatori. Ogni blocco funzionale che supporta uno o piú cluster viene definito endpoint, pertanto dispositivi diversi possono comunicare tra loro facendo ricorso agli stessi endpoint (se implementati).

2.7 Messaggistica

Il protocollo ZigBee definisce due formati per i frame utilizzati per lo scambio di messaggi: il formato Key-Value-Pair (KVP) e il formato Messagge (MSG). Entrambi i formati sono associati ad un Cluster ID, ma il KVP rispetta una struttura definita, a differenza dell'MSG che non é vincolato da alcuna struttura. Il profilo in uso a livello applicazione specifica il formato dei messaggi supportato, ma non é permesso ai cluster di utilizzare entrambi i formati. La gestione della sicurezza é basata su chiavi "link" e chiavi "network". La sicurezza nella comunicazione di tipo unicast tra i due device a livello APL (livello applicazione) si basa sulla gestione condivisa di una chiave "linker" a 128 bit, mentre per quella di tipo broadcast la chiave "network" é condivisa tra tutte le periferiche collegate alla rete. Ovviamente sia la sorgente che la destinazione devono essere a conoscenza del formato in uso nel profilo specificato. Il device puó acquisire la link key tramite key-transport, key-establishment o pre-installation mentre per la network key si usa o la key-transport o la pre-installation. L'architettura include un sistema di sicurezza che riguarda due livelli del protocollo, il layer APS (Application Support Sublayer) e quello NWK (Network layer) che sono responsabili del trasporto dei rispettivi frame. Inoltre il sub-layer APS fornisce i servizi per stabilire e mantenere le relazioni di sicurezza dei device. La ZDO ZigBee Device Object gestisce le politiche e le configurazioni di sicurezza. Nella realizzazione

del progetto si é omessa l'implementazione di questa sezione riguardante la sicurezza per motivi di tempo e poiché la trasmissione prevede solo l'NCAP e il WTIM.

2.7.1 Binding

I dispositivi ZigBee possono comunicare tra di loro se conoscono i rispettivi indirizzi di rete (messaggistica diretta). La scoperta e il mantenimento di questi indirizzi comporta un notevole di- spendio di risorse e un alto overhead. Il protocollo ZigBee, per ovviare a tale problema, offre una caratteristica chiamata binding: il coordinatore memorizza una tabella di corrispondenze degli endpoint connessi alla sua rete. Una volta creati tutti i bind necessari, i dispositivi possono comunicare tra di loro attraverso il coordinatore, che detiene le corrispondenze. Tale tipo di messaggistica prende il nome di messaggistica indiretta ed é utile nelle reti con piú end device. Nello sviluppo di questo progetto il binding é ininfluente e, oltretutto, richiederebbe una gestione efficiente dell'allocazione dinamica di memoria; pertanto non verrá implementato.

2.7.2 Routing

In generale, i protocolli ZigBee minimizzano il tempo di attivitá del radiotrasmettitore, cosí da ridurre il consumo di energia. Nelle reti beacon enabled i nodi consumano energia solo nel periodo in cui c'é il beacon. Il routing in una rete ZigBee é gestito autonomamente dallo stack e non necessita interventi dal livello applicazione.

2.7.3 Operazioni di rete

La rete ZigBee puó essere creata solo da un coordinatore. All'accensione, esso cerca altri coordinatori tra i suoi canali a disposizione. Il tempo che un dispositivo impiega per la scansione delle reti disponibili e per la determinazione dell'energia di ogni canale é definito dal parametro ScanDuration. Per la banda 2.4 GHz, il tempo di scansione (in secondi) é determinato dalla seguente equazione: $T_{Scansione} = 0.01536 \cdot (2 \cdot ScanDuration + 1)$. In base all'energia del canale e al numero di reti presenti in tali canali, il coordinatore stabilisce la propria rete e le attribuisce un PAN ID a 16 bit univoco. Da questo momento in poi router

ed end device possono unirsi alla rete. In caso di conflitto causato da PAN ID uguali, uno dei due coordinatori avvia una procedura di risoluzione del conflitto[1] I dispositivi ZigBee salvano nella memoria non volatile le informazioni relative ai nodi genitori e figli in una tabella chiamata tabella dei vicini. All'accensione, un nodo puó determinare se faceva precedentemente parte di una rete e, in tal caso, avvia una procedura per ricongiungersi ad essa come nodo orfano. I nodi che ricevono tale richiesta verificano se l'orfano era proprio figlio e, in caso affermativo, comunicano la loro posizione all'interno della rete; altrimenti, se tale procedura fallisce oppure il nodo orfano non ha nessun genitore memorizzato nella propria tabella dei vicini, tenterá di connettersi alla rete come nuovo nodo, generando una lista di potenziali genitori e determinando la posizione migliore (in termini di distanza dal coordinatore). Una volta nella rete, un dispositivo puó abbandonarla sia su richiesta del proprio genitore sia autonomamente.

2.8 Caratteristiche tecniche

I dispositivi ZigBee devono rispettare le norme dello standard IEEE 802.15.4-2003 per Low-Rate Wireless Personal Area Network (WPAN). Esso specifica il protocollo di livello fisico (PHY) e il sottolivello Data Link del Medium Access Control (MAC).

2.8.1 PHY

Il protocollo ZigBee opera nella banda ISM non licenziata 2.4 GHz, 915 MHz e 868 MHz. Nella banda 2.4 GHz ci sono 16 canali ZigBee, da 3 MHz ciascuno. I trasmettitori radio usano una codifica DSSS. Si usa una modulazione BPSK nelle bande 868 e 915 MHz e una QPSK con offset (O-QPSK) che trasmette 2 bit per simbolo nella banda 2.4 GHz. Il data rate over-the-air é di 250 Kb/s per canale nella banda 2.4 GHz, 40 Kb/s per canale nella banda 915 MHz e 20 Kb/s nella banda 868 MHz. Il range di funzionamento é compreso tra 10 e 75 metri, dipendentemente dall'ambiente circostante. La massima potenza trasmessa é in genere 0 dBm (1 mW).

[1]Non supportata dallo stack Microchip utilizzato in questo progetto. Cap. 3.

2.8.2 MAC

La modalitá base di accesso al canale, specificata da IEEE 802.15.4-2003, é il Carrier Sense Multiple Access/Collision Avoidance (CSMA/CA). Questo significa che i nodi controllano se il canale é libero prima di trasmettere. Vi sono alcune eccezioni all'uso del CSMA: i segnali di beacon, inviati secondo uno schema prefissato, i messaggi di acknowledge e le trasmissioni di dispositivi in reti beacon-oriented che hanno necessitá di bassa latenza ed usano Guaranteed Time Slots (GTS) che per definizione non fa uso di CSMA. La lunghezza massima dei pacchetti MAC definiti dall'IEEE 802.15.4 é di 127 byte, compreso un campo CRC a 16 bit per il controllo dell'integritá del frame; inoltre lo standard IEEE 802.15.4 prevede l'uso (opzionale) di un meccanismo di acknowledge tramite l'impostazione di un flag di ACK all'interno dei frame inviati. Un messaggio ZigBee puó essere quindi formato al piú da 127 byte, cosí suddivisi:

Medium Access Control (MAC) header: Contiene i campi di controllo del frame a livello MAC, il Beacon Sequence Number (BSN) e le informazioni sull'instradamento del messaggio.

Network layer (NWK) header: Contiene, tra le altre informazioni, l'indirizzo della sorgente e della destinazione.

Application Support Sub-Layer (APS) header: Contiene informazioni riguardo al profilo, al cluster e all'endpoint di destinazione.

APS payload: Dati utili, la cui gestione spetta al livello applicazione.

Capitolo 3

Microchip ZigBee Stack

Lo stack ZigBee fornito da Microchip, che verrá utilizzato per lo sviluppo di questo progetto, é basato sulla versione 2.6 del protocollo ZigBee ed offre diverse caratteristiche, tra cui:

- supporto certificato della versione 2.6 del protocollo ZigBee.
- supporto della banda di frequenze a 2.4 GHz.
- supporto di tutti i tipi di dispositivi ZigBee (Coordinator, Router ed End-Device).
- design modulare e nomenclatura corrispondenti a quelli usati nel protocollo ZigBee e nelle specifiche IEEE 802.15.4.
- portabilitá su tutte le famiglie di microcontrollori PIC18 , PIC24.
- supporto per l'indirizzamento multi casting.
- supporto per meccanismi di rejoin degli end device.

Esistono peró alcune limitazioni, tra cui:

- supporto per sole reti non-beacon.
- indirizzo di rete non riassegnabile ai nodi che lasciano la rete.
- risoluzione dei conflitti PAN ID non supportata.
- frammentazione non supportata.

3.1 Architettura e Livelli

Il Microchip ZigBee Stack é stato progettato rispettando il protocollo ZigBee[1] e le specifiche IEEE 802.15.4, mantenendo cioé la nomenclatura il piú coerente possibile ed organizzando i vari layer in file sorgenti separati ottenendo una libreria modulare indipendente dall'applicazione. Detto stack é scritto in linguaggio "C" ed é orientato ai microcontrollori Microchip della serie PIC18 e PIC24, pertanto puó sfruttare la memoria flash interna del processore per il salvataggio dei parametri quali indirizzo MAC, tabella dei vicini e tabella di binding. Tuttavia é progettato in modo da essere facilmente portabile in altri sistemi basati su microcontrollori PIC e in modo da supportare diversi transceiver con minimi cambiamenti ai livelli piú alti dell'applicazione senza intervenire nei layer piú bassi. Lo stack si basa sul modello ISO/OSI dove ogni livello opera indipendentemente dagli altri fornendo ai livelli immediatamente adiacenti dei servizi su richiesta e scambiando dati attraverso i Service Access Point (SAP) predefiniti. Come mostrato in Figura 3.1, lo stack ZigBee messo a punto da Microchip é ridotto a quattro strati, rispetto ai tradizionali sette previsti dal modello ISO/OSI. Con un approccio di tipo bottom-up si incontrano i seguenti macro layer:

Physical layer (PHY): definito nello standard IEEE 802.15.4, si occupa della gestione del transceiver wireless.

Medium access control layer (MAC): definito nello standard IEEE 802.15.4, si occupa dell'accesso al mezzo.

Network layer (NWK): si occupa del routing (se previsto), della gestione e della sicurezza della rete.

Application layer (APL): fornisce il supporto a quelli che nel modello ISO/OSI sono i layer transport, session, presentation ed application. Questo livello é perció molto complesso e costituisce il cuore dello stack poiché, tramite il proprio sottolivello Application Support Sublayer (APS), scambia messaggi con i livelli sottostanti. All'interno del livello APL esiste inoltre un multiplexer che smista i messaggi inoltrandoli

[1] Si veda [4] e [13].

ai destinatari appropriati. Questi sono indirizzati attraverso strutture denominate endpoint, che possono essere gestiti dal framework dell'applicazione (AFG) oppure endpoint dal protocollo ZigBee (ZDO).

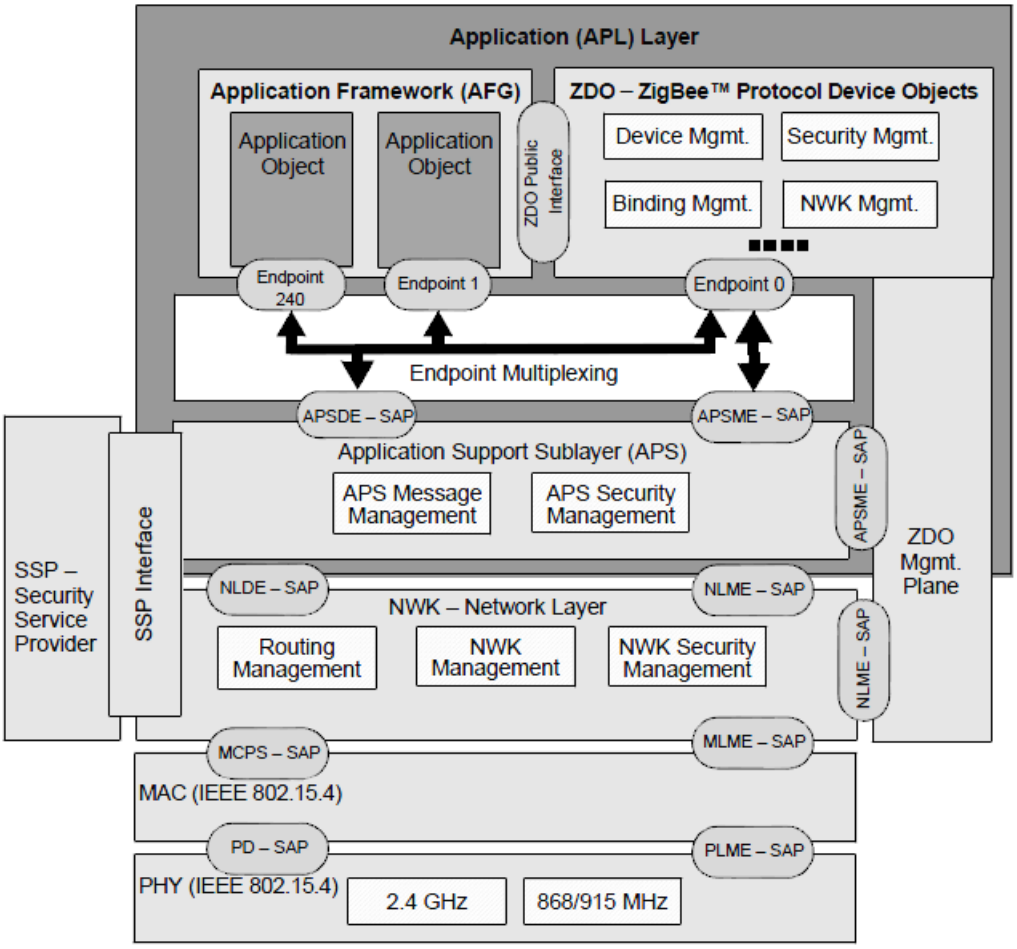

Figura 3.1: Architettura dello stack ZigBee

3.2 Utilizzo

Lo stack Microchip é implementato tramite una macchina a stati con datapath (FSMD) le cui transizioni sono determinate dal tipo di primitiva elaborata ad ogni ciclo della macchina a stati e dal valore dei campi che la compongono.

3.2.1 Primitive

Il funzionamento dello stack di protocolli si basa sull'accesso e la modifica di una particolare struttura dati denominata primitiva; tale struttura raggruppa un insieme di campi che sono accessibili dall'esterno e da ZigBee. Per ogni layer dello stack é definito un insieme di primitive, con funzioni e parametri specifici dello strato a cui appartengono ma il programmatore necessita di conoscere solo il set associato al layer APS in quanto gli altri livelli sono trasparenti all'applicazione e quindi anche alla sua implementazione.
La struttura della macchina a stati ZigBee é estremamente complessa ma allo stesso tempo facile da pilotare in quanto ZigBee risolve automaticamente le transizioni a tutti gli stati; é pertanto sufficiente impostare i parametri della primitiva del layer APS e passarla allo stack. ZigBee la processerá al successivo ciclo macchina e realizzerá le transizioni e le relative azioni associate autonomamente, in funzione della particolare richiesta inoltrata con la primitiva (e.g. invia dato). In Figura 3.2 sono elencate le principali primitive ZigBee.

3.2.2 Funzionamento

Innanzitutto il codice sorgente dell'applicazione deve includere il file header "zAPL.h" per avere accesso alle funzioni dello stack ZigBee. Ogni dispositivo deve poi necessariamente memorizzare una variabile di tipo ZIGBEE PRIMITIVE (qui identificata con il nome "currentPrimitive") per tenere traccia in ogni momento della primitiva in esecuzione da parte dello stack. I dispositivi router ed end device inoltre devono memorizzare le variabili "currentNetworkDescriptor" e "NetworkDescriptor", di tipo NETWORK DESCRIPTOR, utilizzate per le operazioni di rete; una volta memorizzate queste variabili, l'applicazione deve configurare i registri e i pin del microprocessore per attivare l'interfaccia con il transceiver. Ora lo stack puó essere inizializzato con la chiamata alla funzione ZigBeeInit() e possono essere attivati gli interrupt con le istruzioni RCONbits.IPEN=1 e INTCONbits.GIEH=1. Una volta completate le procedure appena descritte, lo stack puó funzionare attraverso la gestione delle primitive definite dai protocolli ZigBee e IEEE 802.15.4. Dopo la memorizzazione del nome della primitiva da eseguire nella variabile "currentPrimitive", la

Primitiva invocata	Risposta	Descrizione
APSDE_DATA_request	APSDE_DATA_confirm	Usata per spedire messaggi ad altri dispositivi
APSME_BIND_request	APSME_BIND_confirm	Usata per creare un binding (se supportato)
APSME_UNBIND_-request	APSME_UNBIND_-confirm	Usata per rimuovere un binding (se supportato)
NLME_NETWORK_-DISCOVERY_request	NLME_NETWORK_-DISCOVERY_confirm	Usata per ricercare reti disponibili. Non usata dai coordinatori.
NLME_NETWORK_-FORMATION_request	NLME_NETWORK_-FORMATION_confirm	Usata per creare una rete sul canale specificato. Solo per coordinatori.
NLME_PERMIT_-JOINING_request	NLME_PERMIT_-JOINING_confirm	Usata per permettere ad altri nodi di connettersi alla rete come propri figli. Solo per coordinatori e router.
NLME_START_-ROUTER_request	NLME_START_-ROUTER_confirm	Usata per attivare le funzionalità di routing. Solo per router.
NLME_JOIN_request	NLME_JOIN_confirm	Usata per connettersi o riconnettersi ad una rete. Non usata dai coordinatori.
NLME_DIRECT_JOIN_-request	NLME_DIRECT_JOIN_-confirm	Usata per aggiungere un dispositivo come figlio. Solo per coordinatori e router.
NLME_LEAVE_request	NLME_LEAVE_confirm	Usata per disconnettersi dalla rete.
NLME_SYNC_request	NLME_SYNC_confirm	Usata per richiedere al nodo genitori i messaggi in attesa di spedizione. Solo per dispositivi di tipo RFD.

Figura 3.2: Principali primitive ZigBee

chiamata alla routine ZigBeeTasks() innesca la procedura di funzionamento dello stack per la primitiva in questione. Tale funzione infatti é un handler (gestore) delle primitive, ovvero si occupa di coordinare i vari layer passando loro le primitive da gestire, anche in maniera ricorsiva. Ogni volta che termina il ciclo di gestione di una primitiva, la variabile "currentPrimitive" deve essere aggiornata con la successiva primitiva da gestire oppure il sistema puó essere lasciato in attesa tramite l'assegnazione della primitiva speciale NO PRIMITIVE. Poiché puó essere gestita soltanto una primitiva per volta, esiste una struttura dati (descritta nel file "ZigBeeTasks.h") adatta al mantenimento dei parametri relativi alla primitiva in corso di gestione. Lo stack dá inoltre la possibilitá di visualizzare via seriale l'output dell'applicazione[2] attraverso una console di tipo terminale. Per fare ció la porta seriale deve essere configurata per supportare un baud rate di 19200 b/s, 8 bit di dati, 1 bit di stop, senza controllo di paritá né di flusso (19200,N,8,1).

[2]Si veda il capitolo 3.5

3.2.3 Funzioni

Le principali funzioni impiegate per gestire il funzionamento dello stack sono:

APLDisable: questa funzione disabilita il ricetrasmettitore. Tipicamente usata dai dispositivi di tipo RFD per risparmiare energia nella modalitá sleep mode.

- Sintassi: BOOL APLDisable(void).
- Input: nessuno.
- Output: TRUE se il ricetrasmettitore é stato disabilitato, FALSE se é stato impossibile disabilitarlo.

APLDiscard: questa funzione scarta il messaggio corrente. Deve essere chiamata al termine del processamento di ogni messaggio ricevuto. Il fallimento di questa funzione impedisce il processo e la ricezione di ulteriori messaggi.

- Sintassi: void APLDiscard(void).
- Input: nessuno.
- Output: nessuno.

APLEnable: questa funzione abilita il ricetrasmettitore.

- Sintassi: void APLEnable(void).
- Input: nessuno.
- Output: nessuno.

APLGet: questa funzione viene utilizzata dall'applicazione per ricevere un byte del messaggio processato da parte dei livelli sottostanti. Se chiamata dopo che tutto il messaggio é stato ricevuto, restituisce il valore 0x00. Il puntatore al byte corrente si aggiorna in modo automatico ad ogni chiamata.

- Sintassi: BYTE APLGet(void).
- Input: nessuno.

- Output: il byte corrente del messaggio processato.

ZigBeeBlockTx: questa funzione blocca il buffer di trasmissione (TxBuffer). Per avere la conferma che il buffer sia effettivamente bloccato, la successiva chiamata alla funzione ZigBeeReady() deve restituire il valore FALSE.

- Sintassi: void ZigBeeBlockTx(void).
- Input: nessuno.
- Output: nessuno.

ZigBeeInit: questa funzione inizializza lo stack ZigBee e deve essere invocata prima di qualsiasi altra funzione e dopo la configurazione dell'hardware.

- Sintassi: void ZigBeeInit(void).
- Input: nessuno.
- Output: nessuno.

ZigBeeReady: questa funzione indica se lo stack é pronto all'invio di un messaggio.

- Sintassi: BOOL ZigBeeReady(void).
- Input: nessuno.
- Output: TRUE se é possibile caricare un nuovo messaggio nel buffer d'uscita, FALSE se il buffer é ancora occupato con il messaggio precedente.

ZigBeeTasks: questa funzione coordina le operazioni dello stack. Il riferimento alla primitiva da eseguire deve essere passato nella variabile *primitive (se non ci sono primitive da eseguire riferirsi alla primitiva NO PRIMITIVE). La funzione continuerá fintanto che non ci saranno primitive del livello applicazione da eseguire.

- Sintassi: BOOL ZigBeeTasks(ZIGBEE PRIMITIVE *primitive).
- Input: primitive 1, puntatore al valore della prossima primitiva da eseguire.
- Output: TRUE se lo stack ha ancora dei task da eseguire in background, FALSE se non ne ha.

3.2.4 Creazione e connessione ad una rete

Il nodo coordinatore, per creare una rete, esegue la primitiva NLME_NETWORK_ FORMATION_request. Se il dispositivo non é un coordinatore e non é connesso a nessuna rete, esso deve raggiungerne una; nel caso fosse precedentemente connesso ad una rete, deve tentare di riconnettersi alla stessa utilizzando la primitiva NLME_JOIN_request con il parametro "RejoinNetwork" settato a TRUE. Nel caso questa procedura fallisca oppure nel caso il dispositivo non facesse precedentemente parte di nessuna rete , deve tentare di connettersi come nuovo nodo; per far ció deve innanzitutto scoprire le reti disponibili attivando la primitiva NLME_NETWORK_DISCOVERY_request. Dopodiché l'applicazione sceglie la rete a cui collegarsi e puó farlo tramite la primitiva NLME_JOIN_request, questa volta con il parametro "RejoinNetwork" settato a FALSE.

3.2.5 Invio di un messaggio

Lo stack ZigBee implementato da Microchip permette di inviare un solo messaggio per volta. Perché ció sia possibile é necessario innanzitutto verificare che la funzione ZigBeeReady() restituisca il valore TRUE, ad indicare che lo stack é pronto. La funzione ZigBeeBlockTx() blocca le trasmissioni (di conseguenza ulteriori chiamate a ZigBeeReady() restituiscono il valore FALSE) ed é quindi possibile caricare il payload del messaggio da inviare nel vettore "TxBuffer" (indicizzato dalla variabile "TxData"), nonché i parametri d'invio nelle apposite locazioni di memoria. Una volta terminato il caricamento, la variabile "TxData" deve puntare al primo elemento libero del buffer, cosicché "TxData" indica anche la lunghezza dei dati contenuti in esso. Fatto questo, é necessario settare "currentPrimitive" con la primitiva APSDE_DATA_request, caricarne i relativi parametri e chiamare la funzione ZigBeeTasks(). Per inviare un messaggio si devono conoscere l'indirizzo e l'endpoint della destinazione, da salvare rispettivamente nelle variabili "destinationAddress" e "destinationEndpoint". Poiché ogni frame é identificato da un Transaction ID univoco, esso puó essere reperito chiamando la funzione APLGetTransID(). Lo stato di invio del messaggio é restituito dalla primitiva APSDE_DATA_confirm. Nel caso

l'invio fallisse, lo stack ritenta l'invio automaticamente fino ad un massimo di tentativi specificato dalla variabile "apscMaxFrameRetries".

3.2.6 Ricezione di un messaggio

Lo stack notifica all'applicazione la ricezione di un nuovo messaggio attraverso la primitiva APSDE_DATA_indication. Assieme a questa primitiva lo stack fornisce tutte le informazioni e i parametri relativi al messaggio memorizzato nel buffer. La funzione APLGet() ad ogni chiamata estrae sequenzialmente un byte dal buffer. Il parametro "DestEndpoint", come dice il nome stesso, indica l'endpoint di destinazione. Se il messaggio é pervenuto all'endpoint corretto puó essere processato, altrimenti deve essere scartato. Dopo essere stato processato, il messaggio deve essere comunque scartato tramite la funzione APLDiscard() in modo da consentire anche ai messaggi successivi di essere processati. La violazione di questa regola porta al blocco della ricezione dei messaggi dopo il primo messaggio processato ma non scartato.

3.3 Implementazione dello Stack su NCAP e WTIM

Per la realizzazione di questo progetto si sono utilizzati, come punto di partenza, due firmware di prova forniti da Microchip (uno per PIC18 (WTIM) e uno per PIC24 (NCAP)) che sono stati modificati per rispondere alle specifiche richieste. Tale approccio é stato l'unico possibile in quanto Microchip non fornisce documentazione sufficiente per poter scrivere un firmware da zero. Studiando quindi i codici di questi demo applicativi si é capito come modificarli per creare due driver ottimizzati. Dal punto di vista funzionale e relativo a ZigBee, i due dispositivi sono identici pertanto identica é anche la struttura dei firmware e la gestione dello stack. Dopo aver ottimizzato i driver a livello ZigBee, si é passati all'implementazione dello standard IEEE1451 e alla sua inclusione negli applicativi. Il cuore del firmware é riportato nell'estratto 3.1, descritto all'interno della funzione main() nei file "Coordinator.c" e "RFD.c":

```
1  while (1) {
2      /* Clear the watch dog timer */
3      CLRWDT();
4
5      /* Process the current ZigBee Primitive */
6      ZigBeeTasks( &currentPrimitive );
7
8      /* Determine the next ZigBee Primitive */
9      ProcessZigBeePrimitives();
10
11     /* do any non ZigBee related tasks */
12     ProcessNONZigBeeTasks();
13 }
```

Listing 3.1: Ciclo while del main, Coordinator.c

CLRWDT(): é la funzione di reset del watchdog timer, necessaria per impedire il blocco sistematico del programma con la generazione di un'interruzione non mascherabile.

ZigBeeTasks(&currentPrimitive): é la funzione che, per ogni iterazione del ciclo macchina, controlla che non ci siano processi pendenti nei layer inferiori. Questo controllo viene effettuato controllando il valore della primitiva contenuta nella variabile currentPrimitive passata in ingresso. Se essa é uguale a NO_PRIMITIVE, ossia la macchina é in stato idle, ZigBeeTasks() interroga i layer piú bassi con funzioni specifiche per di ogni livello, partendo dal piú basso (layer PHY) e risalendo in alto fino a ZDO. Tali funzioni restituiscono NO_PRIMITIVE se non ci sono tasks in backgound nel layer specifico, altrimenti ritornano una primitiva che verrá risolta successivamente. Se il risultato finale di queste verifiche é ancora NO_PRIMITIVE allora il processo di verifica é finito e la funzione ZigBeetask() termina; in caso contrario, tramite una struttura switch-case viene gestita la primitiva specifica. Questo meccanismo viene iterato finché non sono risolti tuti i processi pendenti.

ProcessZigBeePrimitives(): se il sistema non é in idle, ossia currentPrimitive é diversa da NO_PRIMITIVE, il controllo passa alla funzione ProcessZigBeePrimitive(void); che, tramite un costrutto switch-case, gestisce le azioni che devono essere svolte per ogni tipo di primitiva. Questa funzione si occupa inoltre di impostare la primitiva di risposta (ossia lo stato successivo) che dovrá essere processata alla prossima iterazione. Un altro scopo importante di questa funzione é la gestione della console video che viene effettuata al termine del processo della primitiva chiamando la fun-

zione specifica ProcessMenu(). Tale metodo é molto importante in quanto é tramite la console che l'operatore puó modificare il flusso del programma, ossia scegliendo una particolare opzione dal menu video si fa eseguire delle istruzioni particolari al dispositivo. L'uso della console video per accedere al dispositivo é una soluzione di sviluppo temporanea in quanto l'uso dello standard IEEE1451.0 permetterá l'accesso all'NCAP (ossia il Coordinator ZigBee) da remoto tramite connessione ethernet. Per lo scopo di questo progetto, l'output video e l'ingresso da tastiera fungono essenzialmente da debugger.

ProcessNONZigBeeTasks(): nello stack dimostrativo Michrochip questa funzione é vuota, ossia il programmatore dovrebbe inserire qui dentro tutte le istruzioni da eseguire che non riguardano specificatamente ZigBee. Questo impedisce di danneggiare il flusso della macchina a stati ZigBee che é molto delicata. In linea di principio, tutte le chiamate ai metodi IEEE1451 sarebbero dovute risiedere all'interno di questa funzione, ma secondo la filosofia IEEE1451 questo standard dovrebbe introdursi nel firmware nel modo meno invasivo possibile, essendo appunto un accessorio universale del dispositivo che lo utilizza. Si é deciso pertanto di mantenere questa linea di sviluppo e lo standard é stato descritto su listati separati. La chiamata ai metodi IEEE1451 avviene quindi all'interno della funzione ProcessMenu() e non in ProcessNONZigBeeTasks().

3.4 Implementazione standard IEEE1451 e convergence layer

3.4.1 Transducer Services API

IEEE1451Dot0TimDiscovery.c

Questa libreria realizza l'interfaccia tra NCAP e remoto e realizza i metodi per la ricerca di moduli 1451.X registrati con l'NCAP.

```
#include "IEEE1451dot0TimDiscovery.h"
#include "Coordinator .h"
extern UInt16Array timIdDot5 [10];
externUInt8Array moduli1451 [6];
externUInt16Array channelList [][10][10];

UInt16 reportCommModule( UInt8Array *moduleIds){
      moduleIds=moduli1451;
      return 0x0;// NOERROR
}

UInt16 reportTims(UInt8 *moduleId , UInt16Array *timIds){

      if (*moduleId ==5) timIds=timIdDot5;
      return 0x0;// NOERROR
}

UInt16 reportChannels(UInt16 *timId , UInt16Array *channelIds){

      channelIds = &channelList [DOT_FIVE][*timId ][0];
      return 0x0; // NO ERROR
}
```

Listing 3.2: Transducer Services API, IEEE1451Dot0TimDiscovery.c

timIdDot5 e moduli1451 sono delle liste dichiarate nel file principale dell'NCAP "Coordinator.c" e servono per indicizzare rispettivamente i TIM e i moduli registrati con l'NCAP.

channelList memorizza i canali definiti per ogni TIM registrato per ogni modulo IEEE1451 registrato.

reportCommModule fornisce la lista di tutti i moduli 1451 registrati con l'NCAP.

reportTims fornisce, per il modulo specifico selezionato, la lista di tutti i TIM registrati con quel modulo specifico. La lista é unica in quanto l'unico modulo previsto per questo progetto é l'IEEE1451.5.

reportChannels fornisce la lista di tutti i canali disponibili per lo specifico TIM selezionato.

Ciascuna funzione restituisce un codice di errore che dovrá essere considerato in futuro quando si dovrá gestire un database dinamico in cui memorizzare gli identificativi dei dispositivi presenti. In questo caso, trattandosi di assegnazione statica e utilizzando inoltre un unico dispositivo, risulta inutile la gestione di errore in quanto ció che fanno le sopracitate funzioni é semplicemente passare ad una variabile l'indirizzo di liste inserite precedentemente dal programmatore nel firmware.

3.4.2 Transducer Services API

IEEE1451dot0TransducerAccess.c

Questa interfaccia fornisce i metodi per accedere al Transducer Channel e poter quindi leggere o scrivere dei dati nel trasduttore.

```
#include "IEEE1451dot0TransducerAccess.h"

UInt16 open(UInt16 *timId, UInt16 *channelId, UInt16 *transCommId){

    *transCommId=*channelId; // Creo un identificativo per la comunicazione

        params.APSDE_DATA_request.DstAddrMode = APS_ADDRESS_16_BIT;
            params.APSDE_DATA_request.DstAddress.ShortAddr.v[1] = 0x79; //indirizzo destinazione
        params.APSDE_DATA_request.DstAddress.ShortAddr.v[0] = 0x6f;

        params.APSDE_DATA_request.SrcEndpoint = 1;
           params.APSDE_DATA_request.DstEndpoint = 240;
           params.APSDE_DATA_request.ProfileId.Val = MY_PROFILE_ID;

              //params.APSDE_DATA_request.asduLength; TxData
              params.APSDE_DATA_request.RadiusCounter = DEFAULT_RADIUS;
             params.APSDE_DATA_request.DiscoverRoute = TRUE;
             params.APSDE_DATA_request.TxOptions.bits.acknowledged = 1;

           params.APSDE_DATA_request.DiscoverRoute = ROUTE_DISCOVERY_SUPPRESS;

         return 0x0;
}

//UInt16 close(UInt16 transCommId)

UInt16 readData( UInt16 *transCommId, TimeDuration timeout, /*UInt8 SamplingMode,*/ OctetArray *result){
    UInt8 trHigh=((*transCommId)&0xFF00)>>8;
    UInt8 trLow=((*transCommId)&0x00FF);
    result->val[0]=trHigh; //trCmIdMostSgnByte
    result->val[1]=trLow; //trCmIdLeastSgnByte
    result->val[2]=XdcrOperate;   //cmdFunct 3
    result->val[3]=ReadDataSet;    //cmdClass 1

    result->val[4]=0X00; //lenghtMostSgnByte
    result->val[5]=0X01;  //lenghtLeastSgnByte 1 BYTE perch c' solo un otteto dipendente
    result->val[6]=0;   //dataOffset
    result->len=7;
    return 0x0;
}

UInt16 writeData( UInt16 transCommId, TimeDuration timeout, /*UInt8 SamplingMode,*/ OctetArray *value){

    return 0x0;
}
```

Listing 3.3: Transducer Services API, IEEE1451dot0TransducerAccess.c

open: questa funzione apre un canale di comunicazione tra NCAP e TIM ossia imposta i parametri di connessione della primitiva APSDE_DATA_request specificando l'indirizzo del destinatario e la configurazione della trasmissione. Questo metodo deve restituire anche un identificativo univoco per la trasmissione e in questa re-

alizzazione tale identificativo é pari all'identificativo del canale TEMPERATURE. Trattandosi di una comunicazione P2P con un dispositivo a canale singolo non ci sono infatti problemi di conflitti.

close: allo stato attuale non sono gestibili aperture parallele di canali di comunicazione, inoltre ZigBee non richiede che ci siano particolari procedure per terminare una connessione pertanto tale funzione é stata dichiarata ma mai utilizzata.

readData: é la funzione che realizza la formattazione del payload secondo la struttura del Command Message.

3.4.3 Module Communications API - P2PComm

É l'interfaccia che collega il modulo 1451.5 allo strato ZigBee. Rappresenta il nucleo dello strato di convergenza tra i due protocolli in quanto, tramite le funzioni qui implementate, si va a trasferire il payload IEEE1451 al sistema fisico di trasmissione che é ZigBee.

write: la funzione write trasmette il payload dal modulo IEEE1451.5 a ZigBee che lo invia al TIM specificato nell'indirizzo (impostato dalla funzione open()). Questo avviene per mezzo di un ciclo for che carica il buffer di trasmissione TxBuffer byte a byte con il Command Message. Successivamente viene impostata la variabile currentPrimitive con la primitiva APSDE_DATA_request, la cui funzione é preparare e avviare una comunicazione radio. Alla successiva iterazione del ciclo macchina, ZigBee processerá tale primitiva e trasmetterá il contenuto del buffer di trasmissione al WTIM. Il WTIM risponderá con un Reply Message contenente il dato di temperatura letto dal sensore, e la funzione read() successivamente trasmetterá il dato al modulo 1451.5. La caratteristica di tale funzione é che l'elaborazione della primitiva APSDE_DATA_request non avviene nel ciclo macchina principale, bensí all'interno della funzione stessa. Ció é possibile grazie all'utilizzo ricorsivo del ciclo macchina all'interno di read(). Questa soluzione ha un grosso vantaggio: permette di interfacciare gli standard ZigBee e IEEE1451 senza dover mettere mano al programma principale dal momento che tutte le operazioni di lettura e scrittura vengono real-

```
#include "ZigBeeTasks.h"
#include "zAPS.h"
#include "zNVM.h"
#include "Console.h"
#include "IEEE1451dot5ProcessNextState.h"

extern void ProcessZigBeePrimitives(void);
extern UInt16 successo;
extern OctetArray PAYLOAD;
UInt16 ERROR_CODE = 0;
extern ZIGBEE_PRIMITIVE currentPrimitive;

UInt16 read ( TimeDuration timeout ,  UInt32 *len , OctetArray *payload, _Boolean *last)
{
      TxBuffer[TxData++] = 1; //N BYTES RICHIESTI
      ZigBeeBlockTx();
      params.APSDE_DATA_request.DstAddrMode = APS_ADDRESS_16_BIT;
     params.APSDE_DATA_request.DstAddress.ShortAddr.v[1] = 0x79; //SHORT ADDRESS
      params.APSDE_DATA_request.DstAddress.ShortAddr.v[0] = 0x6f; //SHORT ADDRESS
      params.APSDE_DATA_request.RadiusCounter = DEFAULT_RADIUS;
      params.APSDE_DATA_request.DiscoverRoute = ROUTE_DISCOVERY_SUPPRESS;
     #ifdef I_SUPPORT_SECURITY
      params.APSDE_DATA_request.TxOptions.Val = 1;
     #else
      params.APSDE_DATA_request.TxOptions.Val = 0;
     #endif
      params.APSDE_DATA_request.TxOptions.bits.acknowledged = 1;
      params.APSDE_DATA_request.SrcEndpoint = 1;
     params.APSDE_DATA_request.DstEndpoint = 240;
     params.APSDE_DATA_request.ProfileId.Val = 0x7f01;
     params.APSDE_DATA_request.ClusterId.Val = BUFFER_TEST_REQUEST_CLUSTER;
     currentPrimitive = APSDE_DATA_request;

      while(timeout.nsec>0 && currentPrimitive==NO_PRIMITIVE)
      {
           /* Clear the watch dog timer */
         CLRWDT();

           /*MACCHINA A STATI 1451.5*/
         IEEEdot5ProcessNextState();

         /* Process the current ZigBee Primitive */
         ZigBeeTasks( &currentPrimitive );

         /* Determine the next ZigBee Primitive */
         ProcessZigBeePrimitives();

           timeout.nsec--;
      }
      //MACCHINA A STATI 1451.5 per aggiornare lo stato che altrimenti resterebbe open
     IEEEdot5ProcessNextState();
      if(successo != 0x00)
           return ERROR_CODE = 5;  //CORRUPT COMUNICATION NETWORK FALIURE CODE
      else if(timeout.nsec==0)
           return ERROR_CODE = 3;  //OPERATION TIMEOUT CODE
      else
           {
                _Boolean ctrLast= 1;
               last= &ctrLast;
           }
     return ERROR_CODE = 0;
}

UInt16 write ( TimeDuration timeout , OctetArray *payload , _Boolean last)
           {
               int temp = 1; //1 byte to send

               // send buffer test to device with short address 0002
                TxBuffer[TxData++] = temp + 2; // Length
                TxBuffer[TxData++] = 0x00; // octet sequence
                TxBuffer[TxData++] = 0x00;
```

Listing 3.4: Module Communications API, IEEE1451dot0P2PComm.c

```
72              int temp2 = 10;
73              int i ;
74              // Load the transmit buffer with the data to send
75            for(i=0 ; i < temp2; i++)
76              {
77                  TxBuffer[TxData++] = payload.val[i]; //trasmette il payload byte a byte
78               }
79
80         params.APSDE_DATA_request.DstAddrMode = APS_ADDRESS_16_BIT;
81              params.APSDE_DATA_request.DstAddress.ShortAddr.v[1] = 0x79; //indirizzo destinazione
82         params.APSDE_DATA_request.DstAddress.ShortAddr.v[0] = 0x6f;
83
84         params.APSDE_DATA_request.SrcEndpoint = 1;
85            params.APSDE_DATA_request.DstEndpoint = 240;
86            params.APSDE_DATA_request.ProfileId.Val = MY_PROFILE_ID;
87
88               //params.APSDE_DATA_request.asduLength; TxData
89               params.APSDE_DATA_request.RadiusCounter = DEFAULT_RADIUS;
90              params.APSDE_DATA_request.DiscoverRoute = TRUE;
91              params.APSDE_DATA_request.TxOptions.bits.acknowledged = 1;
92
93
94            params.APSDE_DATA_request.DiscoverRoute = ROUTE_DISCOVERY_SUPPRESS;
95
96              #ifdef I_SUPPORT_SECURITY
97                   params.APSDE_DATA_request.TxOptions.Val = 1;
98            #else
99                params.APSDE_DATA_request.TxOptions.Val = 0;
100           #endif
101              params.APSDE_DATA_request.TxOptions.bits.acknowledged = 1;
102        params.APSDE_DATA_request.ClusterId.Val = TRANSMIT_COUNTED_PACKETS_CLUSTER;
103
104         ZigBeeBlockTx();
105             currentPrimitive = APSDE_DATA_request;
106
107                 while(timeout.nsec>0 && currentPrimitive!=NO_PRIMITIVE)
108                 {
109
110                 CLRWDT();
111
112                      /*MACCHINA A STATI 1451.5*/
113                 IEEEdot5ProcessNextState();
114
115                      ConsolePutROMString( (ROM char *)currentPrimitive );
116
117                      ZigBeeTasks(&currentPrimitive);
118
119                 ProcessZigBeePrimitives();
120
121                      timeout.nsec--;
122                 }
123       //MACCHINA A STATI 1451.5 per aggiornare lo stato che altrimenti resterebbe open
124      IEEEdot5ProcessNextState();
125
126         if(!IEEEPrimitive)
127              return ERROR_CODE = 5;  //CORRUPT COMUNICATION NETWORK FALIURE CODE
128         else if(timeout.nsec==0)
129              return ERROR_CODE = 3;  //OPERATION TIMEOUT CODE
130         else
131         {
132              payload= &PAYLOAD;
133
134              last= 1;
135
136
137              //len E last NON CI SERVONO VISTO CHE NON GESTIAMO CHIAMATE MULTIPLE E IL
                      PAYLOAD  FISSO.
138         }
139    return ERROR_CODE = 0;
140
141     };
```

Listing 3.5: Module Communications API - segue da 3.4, IEEE1451dot0P2PComm.c

izzate su un'unitá di traduzione diversa. Nell'ipotesi quindi che Microchip richieda ad esempio di implementare lo standard IEEE1451 su un suo Coordinator ZigBee, sarebbe sufficiente includere questi listati in fase di compilazione aggiungendo poche righe di codice al firmware originale.

read: qui avviene il procedimento inverso rispetto a write(); questo metodo viene invocato in ricezione per trasferire il dato di temperatura da ZigBee al modulo IEEE1451.5. Il dato viene prelevato dal buffer di ricezione, convertito in un formato alfanumerico e viene incapsulato nel payload IEEE1451.

3.5 Test

Per testare il funzionamento del dispositivo, l'EXPLORER_16 con antenna MRF24J40 viene collegato (spento) ad una porta seriale (RS232) di un computer su cui sia disponibile un programma di comunicazione bidirezionale di basso livello, nel caso specifico, un pc con Microsoft Windows Xp PRO ed il programma Microsoft HyperTerminal. Una volta configurata e aperta la comunicazione seriale (19200,N,8,1) é possibile alimentare la board. Come nodo si utilizza una scheda di sviluppo Microchip PICDEM_Z corredata della stessa antenna MRF24J40. All'avvio, dopo i processi preliminari di inizializzazione dell'hardware, l'NCAP crea una rete (Figura 3.3) e manda a video un menu tramite il quale l'utente puó inizializzare la rete per utilizzare le funzionalitá dello standard IEEE 1451 (Figura 3.4). Da questo momento in poi il WTIM puó collegarsi autonomamente

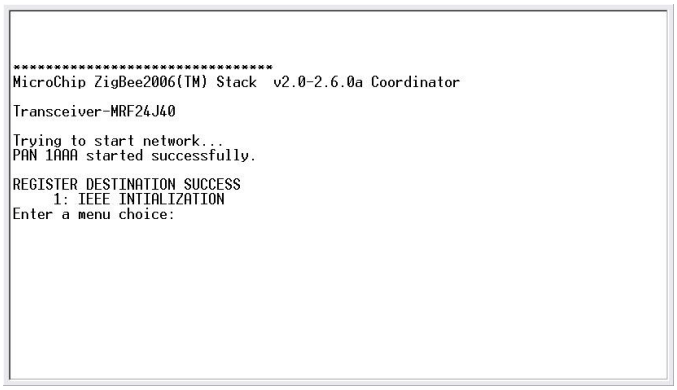

Figura 3.3: Accensione dell'NCAP

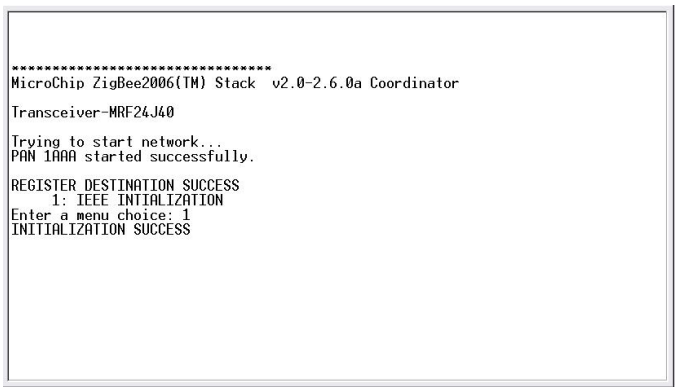

Figura 3.4: Inizializzazione

alla rete. L'NCAP lo riconosce e lo registra inserendolo nella lista dei TIM Dot5. Dopo aver dato notifica dell'avvenuta registrazione, l'NCAP visualizza un menú con le opzioni di comunicazioni specifiche per quel TIM (Figura 3.5). A questo punto, l'NCAP fornisce

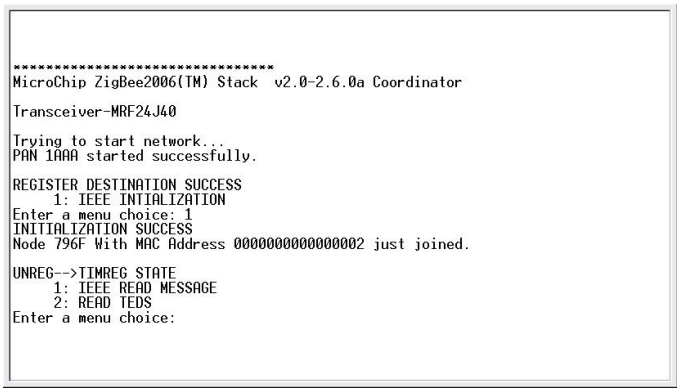

Figura 3.5: Registrazione del WTIM

le liste di moduli, TIM e canali presenti da cui l'utente puó selezionare il dispositivo desiderato (Figura 3.6). In questa fase lutente, scegliendo prima il modulo, poi il TIM e poi il canale non fa altro che chiamare le funzioni "reportCommModule", "reportTims" e "reportChannels" impostandone i parametri di ingresso in funzione delle scelte fatte. Lo scambio di messaggi necessario per la ricezione del dato di misura inizia qui:

- NCAP esegue "open()" che ritorna il transCommId e imposta i parametri di connessione modificando i campi della primitiva "APSDE_DATA_request"

- la funzione "readData()" compone il Command Message e lo trasferisce al payload IEEE1451.

- NCAP chiama la funzione "write()" che carica il buffer di trasmissione di ZigBee con il payload IEEE1451 ed effettua la trasmissione al WTIM passando al ciclo macchina interno la primitiva "APSDE_DATA_request".

- Il WTIM riceve il messaggio e genera la primitiva "APSDE_DATA_indication" che contiene le informazioni sul mittente del messaggio. La funzione "ProcessZigBeePrimitive" del WTIM analizza i campi di questa primitiva e ne estrae il clusterID. Il WTIM scompatta quindi il messaggio ricevuto in base alla descrizione fornita dal cluster e lo decodifica. Quando il messaggio é una richiesta del dato di misura (in questo caso, una temperatura), effettua una lettura del trasduttore e compone il Reply Message caricando tutti i rispettivi campi nel suo buffer di trasmissione. A questo punto il WTIM imposta la sua primitiva "APSDE_DATA_request" e la esegue, inviando quindi il messaggio di risposta all'NCAP.

- NCAP riceve il messaggio e genera la primitiva "APSDE_DATA_indication" che contiene le informazioni sul mittente del messaggio. La funzione "ProcessZigBeePrimitive" dell'NCAP analizza i campi di questa primitiva e ne estrae il clusterID. NCAP scompatta quindi il messaggio ricevuto in base alla descrizione fornita dal cluster e recupera il dato di temperatura che viene formattato e trasferito allo strato IEEE1451.5 e IEEE1451.0.

- NCAP stampa su video il valore del dato di temperatura e la comunicazione termina.

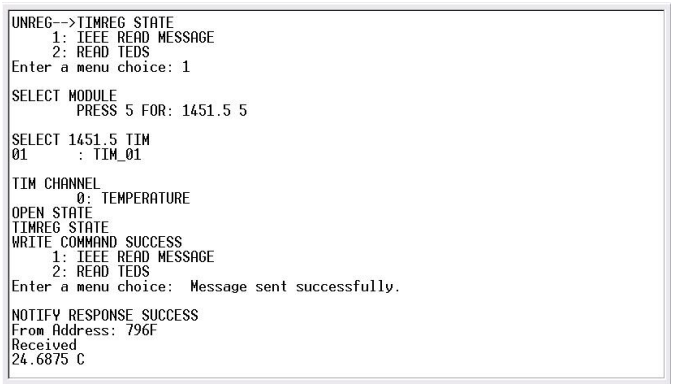

Figura 3.6: Sequenza di comandi per la lettura del dato

Capitolo 4

Standard TCP/IP

4.1 Introduzione

Il Transmission Control Protocol/Internet Protocol non é un vero e proprio standard, ma una suite di protocolli progettati per gestire il trasferimento di dati su grandi network composti da sotto reti connesse mediante router. Seppur mai ufficialmente normato, TCP/IP é ormai uno standard de facto. TCP/IP é infatti il protocollo utilizzato per Internet, che é costituita dall'unione di migliaia di reti distribuite quali universitá, biblioteche, agenzie governative, compagnie e privati. Le radici del TCP/IP possono essere ricondotte al dipartimento della difesa Americano, nello specifico all'Advanced Research Projects Agency (DARPA), tra il 1960 e i primi anni del 1970.

4.2 Tipi di dispositivo

Sebbene non esista uno standard che definisca puntualmente dispositivi e concetti, le seguenti definizioni si sono imposte nel tempo:

Nodo: Qualunque dispositivo, anche router o host, sul quale sia implementato l'IP.

Router: Nodo in grado di inoltrare pacchetti IP non specificatamente destinati ad esso.

Host: Nodo NON in grado di inoltrare pacchetti (NON router). Un host é tipicamente una sorgente o una destinazione di traffico IP. Scarta autonomamente i pacchetti che non siano ad esso destinati.

Protocollo di alto livello: Un qualunque protocollo al di sopra dell'IP che utilizzi l'IP come transport. ICMP, TCP, UDP, FTP, DNS rientrano in questa categoria.

segmento LAN: Porzione di subnet.

Subnet: Una o piú LAN connesse ad uno stesso router, condividono lo stesso prefisso di IP address.

Network: Due o piú subnet connesse mediante un router.

Neighbor: Un nodo connesso alla stessa subnet di un altro nodo.

Interfaccia: Rappresentazione di un collegamento (fisico o logico) tra nodo e subnet.

Indirizzo IP: Identificatore che puó essere utilizzato come sorgente o destinazione di pacchetti IP, assegnato dall'Internet layer ad una interfaccia.

Pacchetto: Unitá dato del protocollo, definita nell'Internet layer, comprende header e payload.

4.3 Modello a strati

Il protocollo TCP/IP viene mappato in un modello concettuale a quattro livelli, noto come modello DARPA. Ad ogni strato del modello DARPA corrispondono uno o piú livelli del modello OSI. (Figura 4.1).

Network Interface Layer: Detto anche Network Access layer, invia e riceve i pacchetti sulla rete. TCP/IP é stato progettato per essere indipendente dal metodo di accesso alla rete, dal formato del frame e dal mezzo. Per questo motivo, TCP/IP puó essere utilizzato per comunicaresu reti che utilizzano tecnologie differenti, LAN e WAN, come ethernet e 802.11. Teoricamente, l'essere indipendente da una specifica

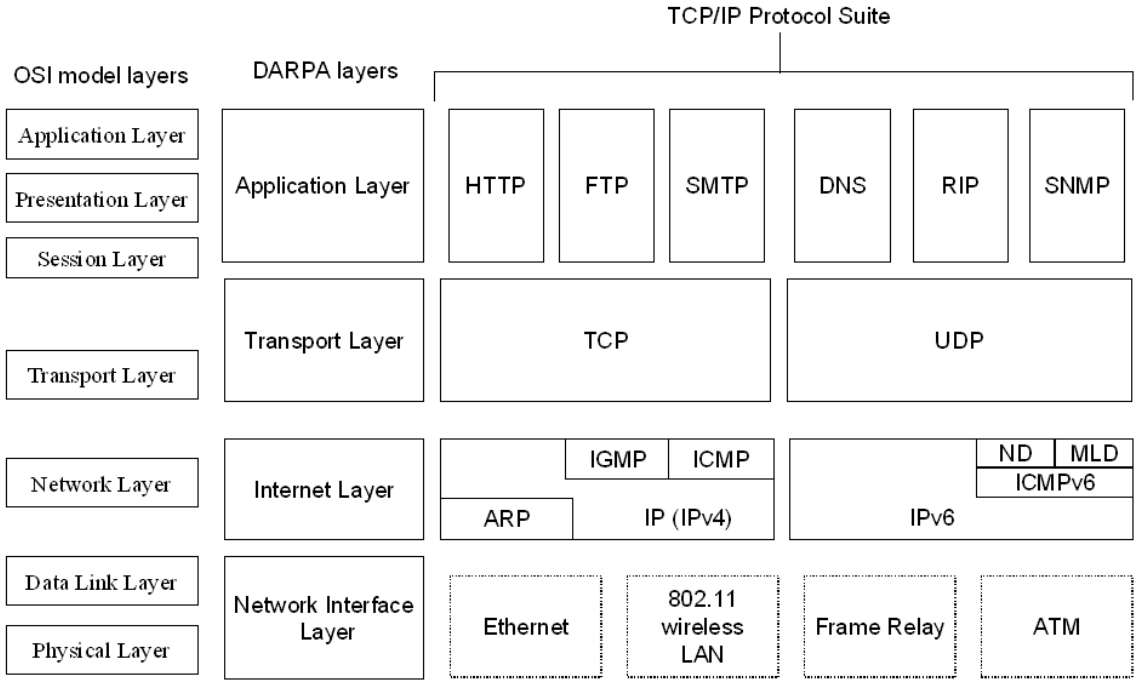

Figura 4.1: Suite TCP/IP, confronto con modelli OSI e DARPA

tecnologia di rete, permette di adattare TCP/IP a qualunque nuova tecnologia. Il Network Interface layer del modello DARPA comprende il Data Link e il Physical layer del modello OSI.

Internet Layer: Questo livello si occupa di impacchettare i dati, indirizzarli e di tutte le funzioni relative al routing. É analogo al Network layer del modello OSI. I protocolli fondamentali di questo livello sono:

- Address Resolution Protocol (ARP), converte indirizzi dell'Internet layer in indirizzi dei livello Network interface, come indirizzi hardware.
- Internet Protocol (IP), indirizza, instrada, frammenta e riassembla i pacchetti di dati.
- Internet Control Message Protocol (ICMP), riporta gli errori ed ogni altra informazione utile a diagnosticare il mancato recapito dei pacchetti di dati.
- Internet Group Management Protocol (IGMP), gestisce i gruppi di IP per multicast.

Transport Layer: Detto anche Host-to-Host Transport layer, fornisce all'Application layer servizi di comunicazione. I protocolli fondamentali di questo livello sono il TCP e l'UDP. TCP fornisce servizi atti a garantire una comunicazione affidabile di tipo uno-a-uno tra dispositivi collegati. Stabilisce le connessioni, verifica la ricezione dei pacchetti inviati e recupera i pacchetti persi in trasmissione. Diversamente, l'UDP fornisce servizi per la comunicazione uno-a-uno o uno-a-tanti di piccoli dati, tipicamente inglobati in un solo pacchetto IP. L'affidabilitá di questo sistema di comunicazione non é garantita e deve essere gestita da livelli superiori o dall'applicazione stessa.

Application Layer: Questo livello consente all'applicazione di accedere ai servizi degli altri livelli e definisce il protocollo usato per lo scambio di dati. L'Application layer contine molti protocolli e ne vengono sempre sviluppati di nuovi. I piú noti ed utilizati sono:

- Hypertext Transfer Protocol (HTTP), trasferisce i file necessari alla visualizzazione di pagine web.
- File Transfer Protocol (FTP), per il trasferimento file.
- The Domain Name System (DNS), risolve gli host name in indirizzi IP.
- The Routing Information Protocol (RIP), protocollo utilizzato dai dispositivi di tipo router per scambiare informazioni sull'instradamento dei dati in una rete IP.
- Simple Network Management Protocol (SNMP), raccoglie e scambia informazioni sulla gestione della rete tra dispositivi di tipo router o piú complessi (bridge, server).

Capitolo 5

Microchip TCP/IP Stack

Lo Stack TCP/IP Microchip é una suite di programmi e codici sorgente che fornisce servizi per applicazioni basate sul protocollo TCP/IP (HTTP Server, Mail Client, etc.). Lo Stack TCP/IP Microchip é implementato in una struttura modulare che rispecchia la struttura a strati dello standard descritto al capitolo 4, in cui tutti i servizi, definiti ad un alto livello di astrazione, possono essere abilitati o disabilitati a piacimento. I punti di forza di questa struttura sono proprio la flessibilitá, la facilitá di configurazione e il fatto che al programmatore non é richiesta nessuna conoscenza specifica di TCP/IP. Dal punto di vista dell'hardware, lo stack si propone come una soluzione valida per tutti quegli impieghi in cui le risorse siano limitate o - come nel caso in oggetto - impiegate da altre applicazioni. É infatti una soluzione particolarmente efficiente sotto l'aspetto dello spazio occupato e poco esosa in termini di capacitá di calcolo, grazie all'adozione di un multitasking a cicli cooperativi, che rende di fatto superflua l'adozione di un sistema operativo dedicato.

5.1 Architettura e Livelli

Buona parte delle implementazioni del protocollo TCP/IP rispecchiano l'architettura del modello di riferimento di figura 5.1. I software basati su questo modello sono divisi in diversi livelli sovrapposti, da cui il nome Stack, pila. É importante notare come in questo modello, ogni livello acceda unicamente a servizi del livello sottostante. Nel caso specifi-

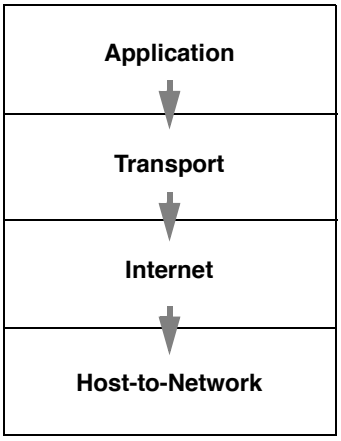

Figura 5.1: Modello di riferimento per TCP/IP

co, molti livelli del modello non si attivano unicamente quando un servizio viene richiesto da un livello superiore ma anche in circostanze quali scadenze di temporizzazioni o arrivi di nuovi pacchetti di dati. Implementare un modello di questo tipo richiederebbe la realizzazione da parte del programmatore di un vero e proprio sistema operativo multi tasking, con conseguente impiego di notevoli risorse hardware, o la fusione tra Stack e main application, con una enorme perdita di flessibilitá. Per rendere fruibile lo Stack anche su microcontrollori a 8 bit con poche centinaia di byte di memoria RAM, Microchip ha preferito realizzare un modello lievemente diverso di Stack, implementando ogni singolo layer in un file sorgente separato, mantenendo le API raggruppate e accessibili al programmatore per mezzo dei file di header. Come si vede in figura 5.2, diversamente

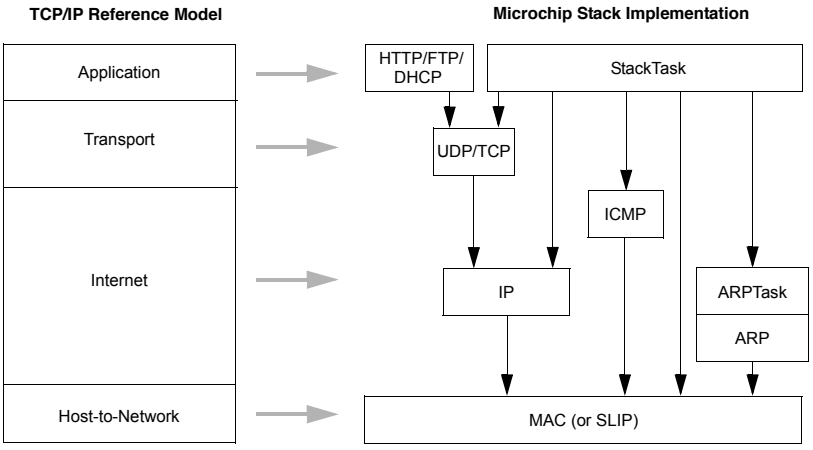

Figura 5.2: Confronto tra modello di riferimento TCP/IP e modello Microchip

dal modello di riferimento, molti dei livelli dello Stack Microchip accedono direttamente

```
1   while(1)
2   {
3       // Blink LED0 (right most one) every second.
4       if(TickGet() - t >= TICK_SECOND/2ul)
5       {
6           t = TickGet();
7           LED0_IO ^= 1;
8       }
9
10      // This task performs normal stack task including checking
11      // for incoming packet, type of packet and calling
12      // appropriate stack entity to process it.
13      StackTask();
14
15      // This tasks invokes each of the core stack application tasks
16      StackApplications();
17
18        // Process application specific tasks here.
19        // For this demo app, this will include the Generic TCP
20        // client and servers, and the SNMP, Ping, and SNMP Trap
21        // demos. Following that, we will process any IO from
22        // the inputs on the board itself.
23        // Any custom modules or processing you need to do should
24        // go here.
25        #if defined(STACK_USE_GENERIC_TCP_CLIENT_EXAMPLE)
26        GenericTCPClient();
27        #endif
28
29        #if defined(STACK_USE_GENERIC_TCP_SERVER_EXAMPLE)
30        GenericTCPServer();
31        #endif
32
33        #if defined(STACK_USE_ICMP_CLIENT)
34        PingDemo();
35        #endif
36
37        #if defined(STACK_USE_SNMP_SERVER) && !defined(SNMP_TRAP_DISABLED)
38        SNMPTrapDemo();
39        if(gSendTrapFlag)
40            SNMPSendTrap();
41        #endif
42
43        ProcessIO();
44
45      // If the local IP address has changed (ex: due to DHCP lease change)
46      // write the new IP address to the LCD display
47        if(dwLastIP != AppConfig.MyIPAddr.Val)
48        {
49            dwLastIP = AppConfig.MyIPAddr.Val;
50
51            DisplayIPValue(AppConfig.MyIPAddr);
52
53        }
54  }
```

Listing 5.1: Esempio di Multitasking Cooperativo, MainDemo.c

ad uno o piú livelli che non sono direttamente sottostanti. Un'altra grossa differenza é rappresentata dall'aggiunta di due nuovi moduli, "StackTask" e "ARPTask".
Per permettere l'esecuzione di operazioni cadenzate in maniera asincrona, senza ricorrere ad un complesso multitasking eppure mantenendo una certa separazione tra Stack e main application, Microchip ricorre alla tecnica chiamata "multitasking cooperativo".
In un sistema multitasking cooperativo sono presenti diversi task, che peró non vengono eseguiti contemporaneamente, ciascuno esegue le proprie operazioni per poi rendere il controllo al task successivo. Questo comporta che l'applicazione principale debba essere divisa in diversi task, o che sia organizzata come una macchina a stati finiti. Il codice 5.1, estratto dal MainDemo fornito con lo Stack, é una chiara illustrazione di come un cooperative multitasking possa essere implementato mediante una macchina a stati. Ad ogni ciclo, infatti, StackTask verifica la presenza di pacchetti in entrata, instradandoli verso il componente dello stack designato a processarli. L'istruzione successiva, StackApplications, chiama tutti i moduli applicazione configurati, di modo che possano processare detti pacchetti. A seguire possono essere introdotti tutti i task specifici del programma, avendo cura che non impieghino per troppo tempo il microcontrollore. Come ultima istruzione, infine, vengono processati tutti gli input/output della scheda.

5.2 Configurazione dello Stack e dei servizi

Per facilitare il processo di configurazione, lo stack utilizza le direttive "#define" del codice "C". Per abilitare, disabilitare o impostare un particolare parametro, il programmatore deve agire quasi unicamente su queste definizioni, raggruppate per lo piú nel file "StackTsk.h". Procedura analoga vale per quanto riguarda la configurazione dei servizi che si intendono utilizzare a livello dell'applicazione. Per abilitare o disabilitare un determinato servizio, al programmatore é richiesto di includere o escludere (commentando) la direttiva "#define" associata che si trova nel file "TCPIPConfig.h". Nella Tabella 1 dell'Application Note 833 rilasciata da Microchip [14] si trova una dettagliata lista dei "#define" disponibili, dei relativi valori impostabili, dei file a cui fanno riferimento ed una breve ma esaustiva descrizione della loro funzione. Negli estratti di codice 5.2 e 5.3 invece,

é possibile vedere la configurazione correntemente adottata in questo progetto.

I servizi inclusi sono quelli basilari per la realizzazione di un server HTTP, ovvero:

```
//============================================================
// Application Options
//============================================================
/* Application Level Module Selection
 * Uncomment or comment the following lines to enable or
 * disabled the following high-level application modules.
 */
//#define STACK_USE_UART     // Application demo using UART for IP address display
//#define STACK_USE_UART2TCP_BRIDGE  // UART to TCP Bridge application example
//#define STACK_USE_IP_GLEANING
#define STACK_USE_ICMP_SERVER   // Ping query and response capability
#define STACK_USE_ICMP_CLIENT   // Ping transmission capability
//#define STACK_USE_HTTP_SERVER   // Old HTTP server
#define STACK_USE_HTTP2_SERVER   // New HTTP server with POST, Cookies, Authentication, etc.
//#define STACK_USE_SSL_SERVER   // SSL server socket support (Requires SW300052)
//#define STACK_USE_SSL_CLIENT   // SSL client socket support (Requires SW300052)
#define STACK_USE_AUTO_IP // Dynamic link-layer IP address automatic configuration protocol
#define STACK_USE_DHCP_CLIENT   // Dynamic Host Configuration Protocol
#define STACK_USE_DHCP_SERVER   // Single host DHCP server
//#define STACK_USE_FTP_SERVER   // File Transfer Protocol (old)
//#define STACK_USE_SMTP_CLIENT   // Simple Mail Transfer Protocol for sending email
//#define STACK_USE_SNMP_SERVER   // Simple Network Management Protocol v2C Community Agent
//#define STACK_USE_TFTP_CLIENT   // Trivial File Transfer Protocol client
//#define STACK_USE_GENERIC_TCP_CLIENT_EXAMPLE // HTTP Client example in GenericTCPClient.c
//#define STACK_USE_GENERIC_TCP_SERVER_EXAMPLE // ToUpper server example in GenericTCPServer.c
//#define STACK_USE_TELNET_SERVER   // Telnet server
//#define STACK_USE_ANNOUNCE   // Microchip Embedded Ethernet Device Discoverer server/client
#define STACK_USE_DNS     // Domain Name Service Client for resolving hostname strings to IP
#define STACK_USE_NBNS     // NetBIOS Name Service Server for repsonding to NBNS hostname
#define STACK_USE_REBOOT_SERVER   // Module for resetting this PIC remotely.
#define STACK_USE_SNTP_CLIENT   // Simple Network Time Protocol for obtaining current date/tim
//#define STACK_USE_UDP_PERFORMANCE_TEST // Module for testing UDP TX performance characteristics.
//#define STACK_USE_TCP_PERFORMANCE_TEST // Module for testing TCP TX performance characteristics
//#define STACK_USE_DYNAMICDNS_CLIENT  // Dynamic DNS client updater module
//#define STACK_USE_BERKELEY_API   // Berekely Sockets APIs are available
```

Listing 5.2: Modulo di selezione dei servizi, TCPIPConfig.h

ICMP_SERVER e ICMP_CLIENT: Internet Control Message Protocol, non strettamente necessario per il funzionamento del server HTTP in quanto non utilizzato per lo scambio dati, ma utile in fase di debug per verificare la presenza del dispositivo sulla rete[1].

HTTP2_SERVER: Questo é il vero e proprio HyperText Tranfer Protocol server, del quale si utilizza la seconda versione, piú completa dal punto di vista delle funzionalitá. Essenziale ai fini del progetto.

AUTO_IP, DHCP_CLIENT e DHCP_SERVER: Non sono strettamente necessari, ma abilitando il Dynamic Host Configuration Protocol e l'auto-assegnazione

[1] Quando layer di comunicazione piú complessi hanno manifestato malfunzionamenti, si é rivelato utile poter almeno "pingare" il dispositivo.

dell'indirizzo IP, si rende piú agevole l'inserimento del dispositivo in una rete di computer.

DNS e NBNS Domain Name Service e NetBIOS Name Service, non strettamente necessari, permettono di interrogare il dispositivo posto in una rete di computer, senza conoscerne l'indirizzo IP, ma digitando nella barra degli indirizzi del browser un "nome proprio" definito in fase di programmazione.

REBOOT_SERVER: Il servizio di riavvio da remoto del microcontrollore, sempre utile, diventa necessario a seguito - ad esempio - di un aggiornamento remoto dell'immagine mpfs[2] caricata sul dispositivo.

SNTP_CLIENT: Simple Network Time Protocol, non necessario, permette al dispositivo di ottenere data e ora.

Alcuni servizi sono stati disabilitati per necessitá:

UART e UART2TCP_BRIDGE: Lo Uart viene utilizzato (attraverso la porta seriale del dispositivo) per le operazioni di debug della macchina a stati ZigBee. É pertanto indispensabile che nessun altro servizio vada ad occupare la stessa linea di comunicazione.

HTTP_SERVER: Prima versione di HyperText Tranfer Protocol server, estremamente limitata nelle funzionalitá.

Sono poi stati esclusi tutti i servizi puramente dimostrativi di Microchip, quelli totalmente inutili ai fini del progetto ed alcuni, non essenziali, che peró meriterebbero un approfondimento[3] quali:

SSL_SERVER e SSL_CLIENT: Secure Sockets Layer, protocollo crittografico per garantire la sicurezza delle comunicazioni sulla rete.

SMTP_CLIENT: Simple Mail Transfer Protocol, per gestire l'invio di email dal dispositivo

[2] Si veda pg.47
[3] Si veda pg.75

TELNET_SERVER: TErminaL NETwork, protocollo di rete, per fornire una comunicazione bidirezionale, testuale, con il dispositivo.

```
//============================================================
// Data Storage Options
//============================================================

/* MPFS Configuration
 * MPFS is automatically included when required for other
 * applications. If your custom application requires it
 * otherwise, uncomment the appropriate selection.
 */
//#define STACK_USE_MPFS
//#define STACK_USE_MPFS2

/* MPFS Storage Location
 * If html pages are stored in internal program memory,
 * comment both MPFS_USE_EEPROM and MPFS_USE_SPI_FLASH, then
 * include an MPFS image (.c or .s file) in the project.
 * If html pages are stored in external memory, uncomment the
 * appropriate definition.
 *
 * Supported serial flash parts include the SST25VFxxxB series.
 */
#define MPFS_USE_EEPROM
//#define MPFS_USE_SPI_FLASH
```

Listing 5.3: Opzioni per la memorizzazione dei dati, HardwareProfile.h

Per quel che riguarda l'archiviazione delle pagine web del server HTTP, come mostrato nel listato 5.3, si é ricorso all'archiviazione mediante Multi Protocol File System, salvato sulla eeprom della scheda di sviluppo. Questa soluzione abbassa lievemente le prestazioni del dispositivo, aumentando i tempi di risposta, ma permette di ridurre notevolmente il footprint del firmware, in termini sia di data memory che di program memory. Se in futuro la quantitá di dati da immagazzinare dovesse eccedere la capacitá della eeprom residente, sará sempre possibile abilitare l'archiviazione su supporto flash (scheda di memoria Secure Digital, ad esempio) collegato al bus SPI.

5.3 Configurazione dell'hardware

I file sorgente dello Stack TCP/IP di Microchip sono concepiti e strutturati in modo da poter essere compilati per hardware diversi, permettendone cosí l'utilizzo su svariate demo board. Per questo progetto era disponibile una scheda di sviluppo di tipo Explorer_16, dotata di microcontrollore PIC24FJ128GA010 con pictail ethernet ENC28J60. Quest'hardware é pienamente supportato dai file di configurazione e dalle applicazioni

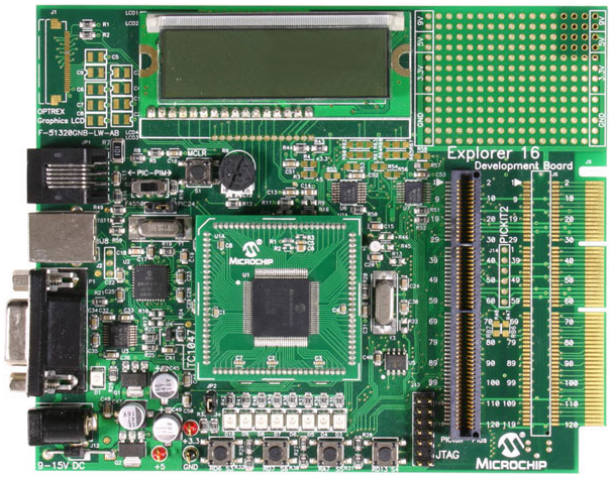

Figura 5.3: Explorer_16

Figura 5.4: ENC28J60

dimostrative fornite dallo Stack con la sola accortezza di rispettare le connessioni di default, ovvero con il pictail ethernet (ENC28J60) inserito nel bus SPI1 del primo slot (J5). La prevista integrazione di una antenna ZigBee (MRF24J40) ha indotto la necessitá di scalare questa[4] connessione al bus SPI2 del secondo slot (J6). Le variazioni essenziali a garantire il corretto funzionamento della nuova configurazione sono riportate nell'estratto di codice 5.4, e sono state ricavate da un'attenta analisi degli schemi di figura 5.5, tratta da [7].

```
1  // ENC28J60 I/O pins
2      #define ENC_CS_TRIS   (TRISFbits.TRISF12) //rr D14
3      #define ENC_CS_IO   (PORTFbits.RF12) //rr D14
4      //#define ENC_RST_TRIS  (TRISDbits.TRISD15) // Not connected by default.
5      //#define ENC_RST_IO   (PORTDbits.RD15)
6      // SPI SCK, SDI, SDO pins are automatically controlled by the
7      // PIC24/dsPIC/PIC32 SPI module
8      #if defined(__C30__) // PIC24F, PIC24H, dsPIC30, dsPIC33
9          #define ENC_SPI_IF   (IFS2bits.SPI2IF) //rr(IFS0bits.SPI1IF)
10         #define ENC_SSPBUF   (SPI2BUF)   //rr(SPI1BUF)
11         #define ENC_SPISTAT   (SPI2STAT)   //rr(SPI1STAT)
12         #define ENC_SPISTATbits  (SPI2STATbits)  //rr(SPI1STATbits)
13         #define ENC_SPICON1   (SPI2CON1)   //rr(SPI1CON1)
14         #define ENC_SPICON1bits  (SPI2CON1bits)  //rr(SPI1CON1bits)
15         #define ENC_SPICON2   (SPI2CON2)   //rr(SPI1CON2)
16     #else    // PIC32
17         #define ENC_SPI_IF   (IFS0bits.SPI1RXIF)
18         #define ENC_SSPBUF   (SPI1BUF)
19         #define ENC_SPISTATbits  (SPI1STATbits)
20         #define ENC_SPICON1   (SPI1CON)
21         #define ENC_SPICON1bits  (SPI1CONbits)
22         #define ENC_SPIBRG   (SPI1BRG)
23     #endif
```

Listing 5.4: Mappatura dell'hardware, HardwareProfile.h

[4]Si é constatato che scalare in SPI2 il pictail ZigBee avrebbe comportato maggiori complicazioni.

FIGURE A-4: EXPLORER 16 BOARD SCHEMATIC, SHEET 3 OF 8 (MPLAB® ICD 2, JTAG, PICkit™ 2 AND PICtail™ Plus CONNECTORS)

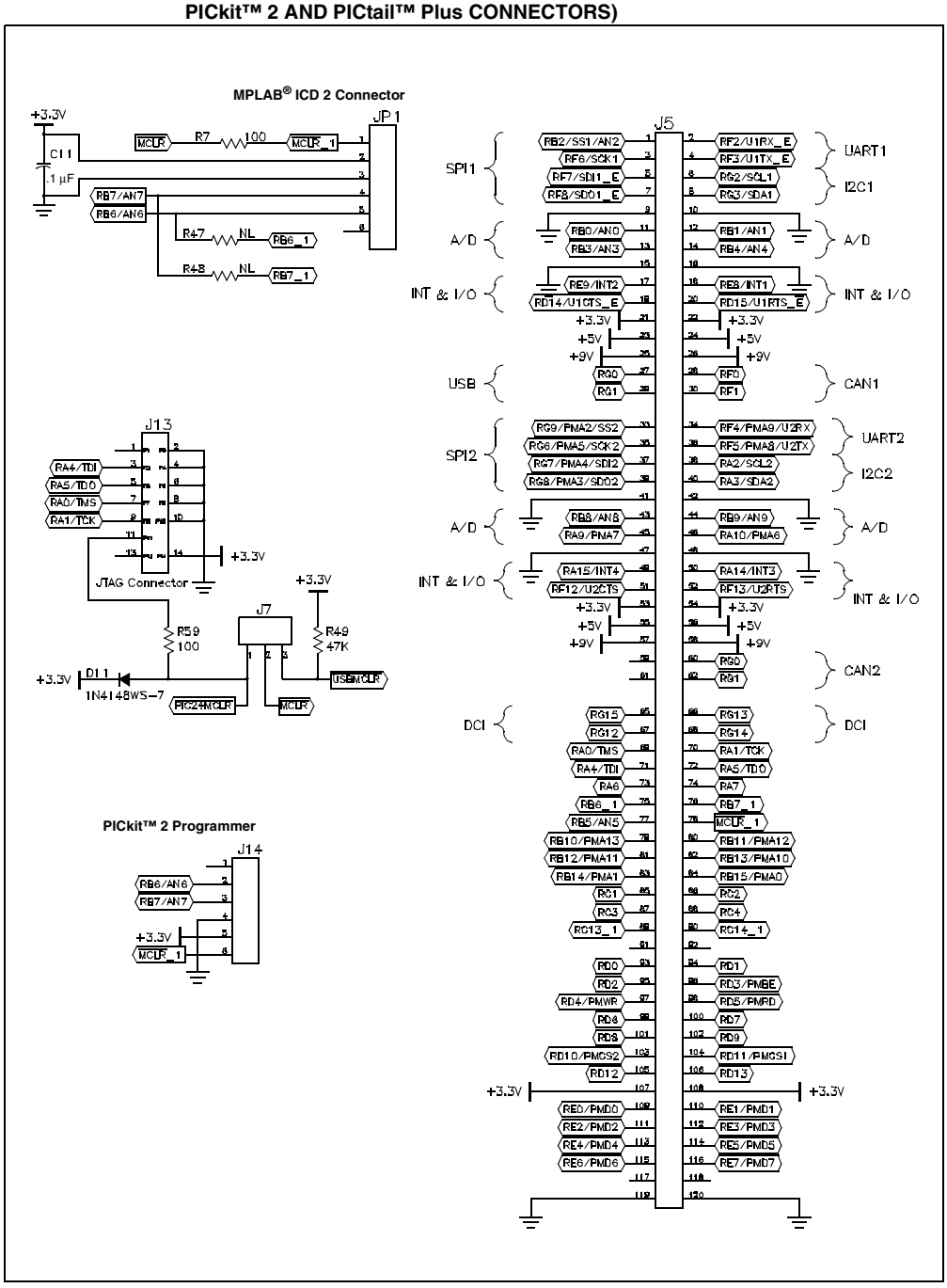

Figura 5.5: Connettore J5 su Explorer_16

5.4 Test

Per testare la configurazione é stato utilizzato un clone del sito dimostrativo fornito con lo Stack, opportunamente modificato per verificare buona parte delle funzionalitá della scheda, visualizzando e impostando in tempo reale lo stato dei componenti principali. Le pagine, scritte in HTML con buone integrazioni di AJAX e JAVAscript, sono anche un'ottimo esempio di utilizzo delle variabili dinamiche che dovranno poi essere utilizzate in maniera estensiva nella realizzazione del gateway. Gli estratti di codice 5.5 e 5.6, rappresentano il vero nucleo del file index.htm, al di la del layout della pagina e degli elementi decorativi. Nell'estratto 5.5, si riconosce lo script che si occupa, usando il

```
<script type="text/javascript">
<!--
// Parses the xmlResponse from status.xml and updates the status box
function updateStatus(xmlData) {

	// Check if a timeout occurred
	if(!xmlData)
	{
		document.getElementById('display').style.display = 'none';
		document.getElementById('loading').style.display = 'inline';
		return;
	}

	// Make sure we're displaying the status display
	document.getElementById('loading').style.display = 'none';
	document.getElementById('display').style.display = 'inline';

	// Loop per tutti i leds
	for(i = 0; i < 8; i++) {
		if(getXMLValue(xmlData, 'led'+i) == '1')
			document.getElementById('led' + i).style.color = '#090';
		else
			document.getElementById('led' + i).style.color = '#ddd';
	}

	// Loop per tutti i pulsanti
	for(i = 0; i < 4; i++) {
		if(getXMLValue(xmlData, 'btn'+i) == 'up')
			document.getElementById('btn' + i).innerHTML = '&Lambda;';
		else
			document.getElementById('btn' + i).innerHTML = 'V';
	}

	// Update del Potenziometro
	document.getElementById('pot0').innerHTML = getXMLValue(xmlData, 'pot0');
}
setTimeout("newAJAXCommand('status.xml', updateStatus, true)",500);
//-->
</script>
```

Listing 5.5: Status report della board, index.htm

metodo "getElementById" di recuperare dal file "status.xml" lo stato dei componenti presenti sulla board, permettendone la visualizzazione. L'estratto 5.6 invece mostra come

```
<div id="status">

    <div id="loading" style="display:none">Error:<br />Connection to demo board was lost.</div>

    <div id="display">
        <span style="float:right;font-size:9px;font-weight:normal;padding-top:8px;text-indent:0px">
        (click to toggle)
        </span>

        <p>LEDs:<br /><span class="leds">
        <a id="led7" onclick="newAJAXCommand('leds.cgi?led=7');">&bull;</a>
        <a id="led6" onclick="newAJAXCommand('leds.cgi?led=6');">&bull;</a>
        <a id="led5" onclick="newAJAXCommand('leds.cgi?led=5');">&bull;</a>
        <a id="led4" onclick="newAJAXCommand('leds.cgi?led=4');">&bull;</a>
        <a id="led3" onclick="newAJAXCommand('leds.cgi?led=3');">&bull;</a>
        <a id="led2" onclick="newAJAXCommand('leds.cgi?led=2');">&bull;</a>
        <a id="led1" onclick="newAJAXCommand('leds.cgi?led=1');">&bull;</a>
        <a id="led0">&bull;</a>
        </span></p>

        <p>Pulsanti:<br />
        <span id="btn3">?</span>  
        <span id="btn2">?</span>  
        <span id="btn1">?</span>  
        <span id="btn0">?</span></p>
        <p>Potenziometro: <span id="pot0" style="font-weight:normal">?</span></p>
    </div>

</div>
```

Listing 5.6: Status report della board, index.htm

sia possibile, usando i comandi AJAX, abbinare all'evento "onclick" la variazione di una variabile dinamica locata nel file "leds.cgi", permettendo cosí di cambiarne il valore logico, accendendo fisicamente l'indicatore sulla board. Per caricare le pagine HTML del sito nella memoria interna del dispositivo, é necessario comprimere tutta la struttura in un unico file immagine mpfs con estensione ".bin". Questa operazione viene svolta direttamente da un'utility gratuita distribuita da Microchip con lo Stack stesso.

Il file binario puó essere integrato ai file sorgenti in fase di compilazione o, sfruttando il servizio di uploading, caricato direttamente da remoto; per tutti i test eseguiti si é utilizzata questa seconda procedura. Collegando ad una rete di computer il dispositivo programmato, ma senza pagine HTML precaricate, questo viene immediatamente riconosciuto. Dopo una prima procedura di autoidentificazione e autoassegnazione dell'indirizzo IP, visualizzato per comoditá sul display in dotazione, la board risulta operativa. Interrogando la scheda da browser, mediante indirizzo IP o corrispettivo mnemonico assegnato (mchpboard), si ottiene peró un messaggio di errore che reindirizza l'utilizzatore ad una pagina di upload (Figura 5.6) nella quale si richiede di indicare la posizione del file binario che

che si intende caricare. Al termine del trasferimento dati, dopo un messaggio di conferma, la board si riavvia e presenta le pagine desiderate (Figura 5.7). Da queste pagine, come

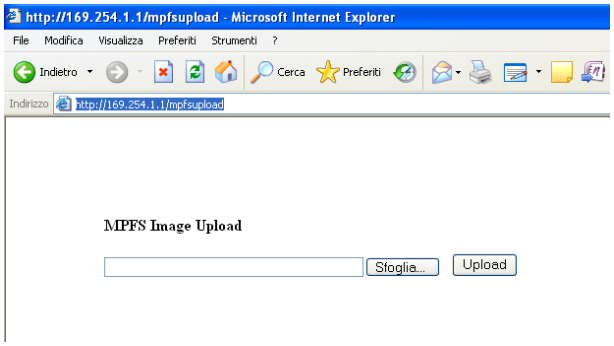

Figura 5.6: Pagina di upload

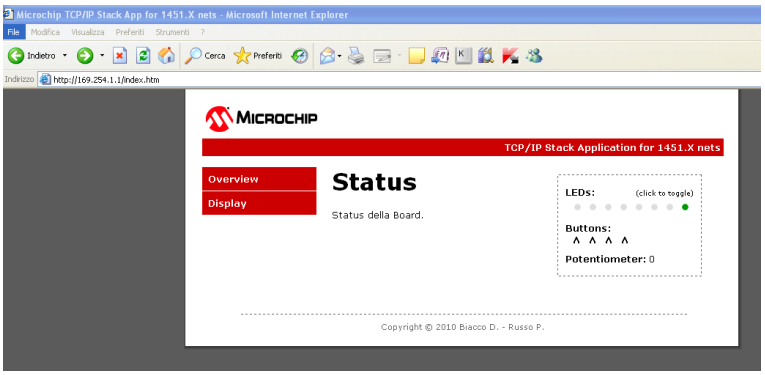

Figura 5.7: Pagine di test

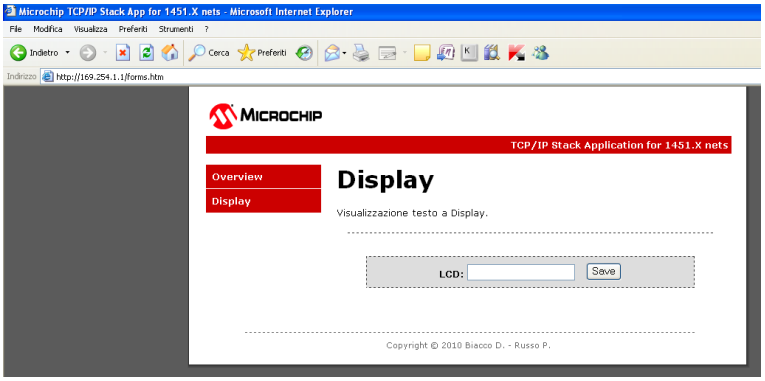

Figura 5.8: Pagine di test

detto, si ha un rapporto immediato sullo stato dei componenti della scheda (leds, pulsanti, display e potenziometro) e si ha la possibilitá di pilotare direttamente display e leds.

Capitolo 6

Integrazione dei due Stack

6.1 File e cartelle

Il primo passo per l'integrazione dei due stack, é stato la rimozione dallo stack TCP/IP di tutte le componenti superflue, dimostrative o riguardanti altro hardware opzionale.

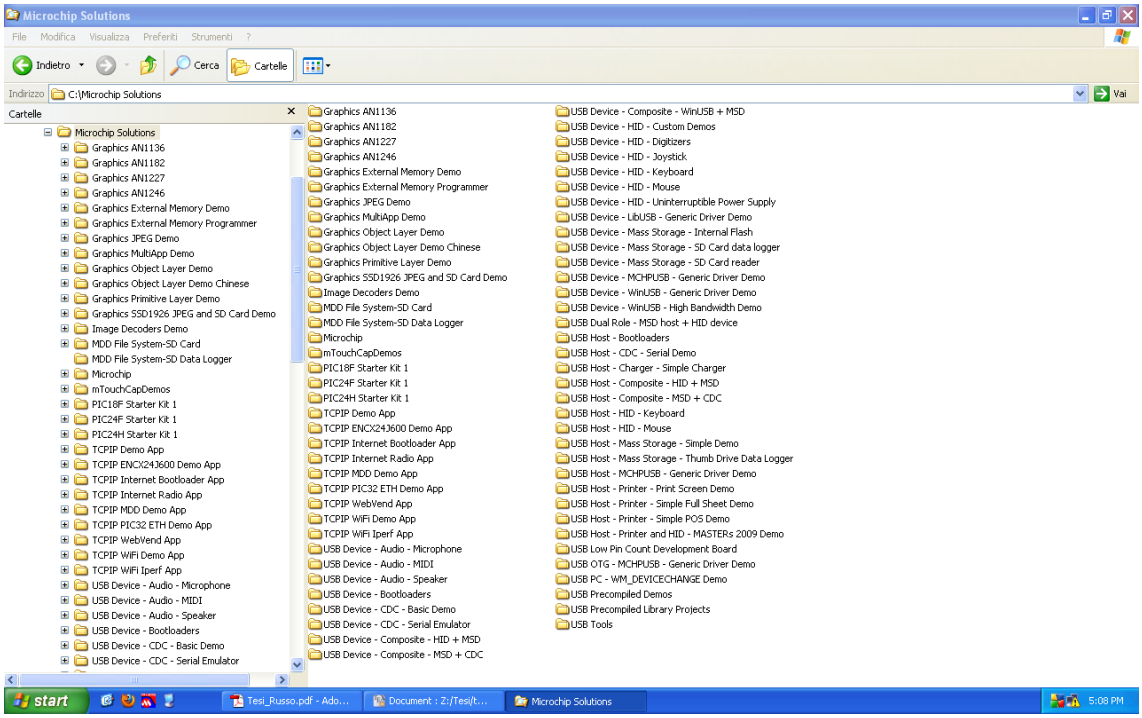

Figura 6.1: Stack TCP/IP completo

Come si puó vedere dal confronto tra Figura 6.1 e 6.2, infatti, é stato rimosso tutto il supporto per i display grafici, per i microcontrollori diversi da quelli della famiglia 24F,

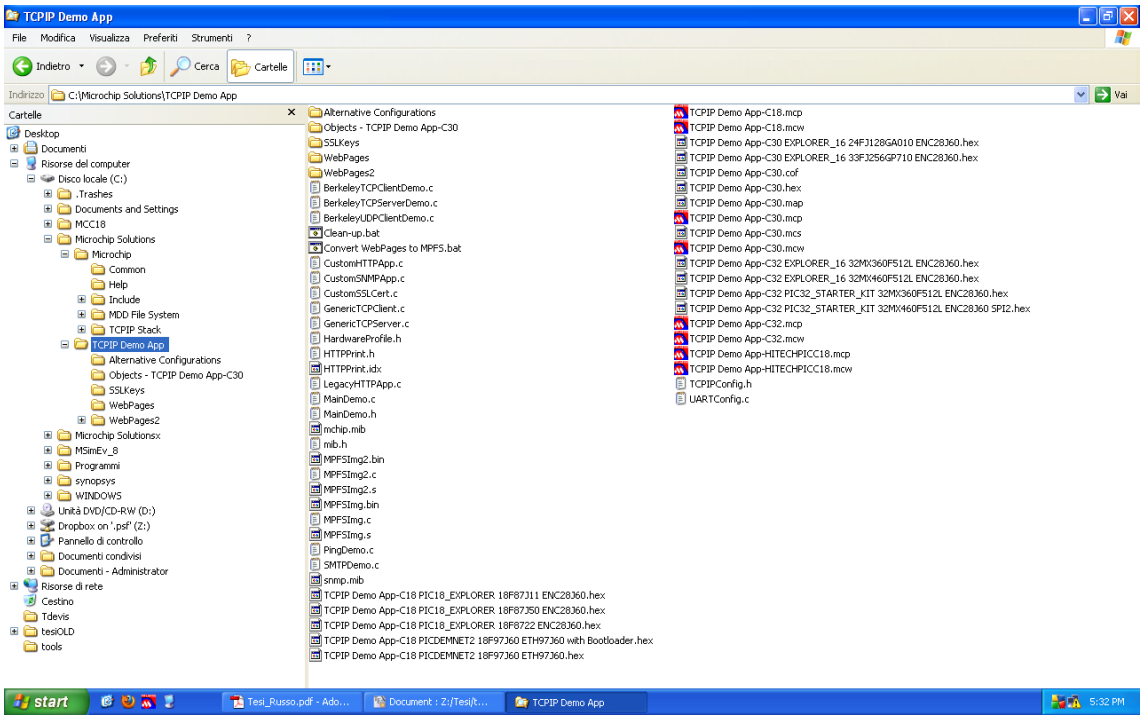

Figura 6.2: Stack TCP/IP ridotto

per i transceiver diversi da quello in dotazione e per tutto l'hardware basato su USB. Sono stati rimossi anche tutti gli esempi di codice e le application note non pertinenti. Questa scelta, fatta unicamente per snellire l'archiviazione del progetto e per non creare confusione a chi - in futuro - dovrá riprenderne lo sviluppo, non é assolutamente una limitazione. Sará sempre possibile reintegrare dette funzionalitá semplicemente riaggiungendo i necessari file, in virtú della struttura modulare dello stack.

Si é proceduto poi con l'unione dei file e delle cartelle che compongono gli stack. Per fare questo si é cercato di mantenere una certa separazione logica e la gerarchia comune agli standard Microchip. (Figure 6.3,6.4 e 6.5).

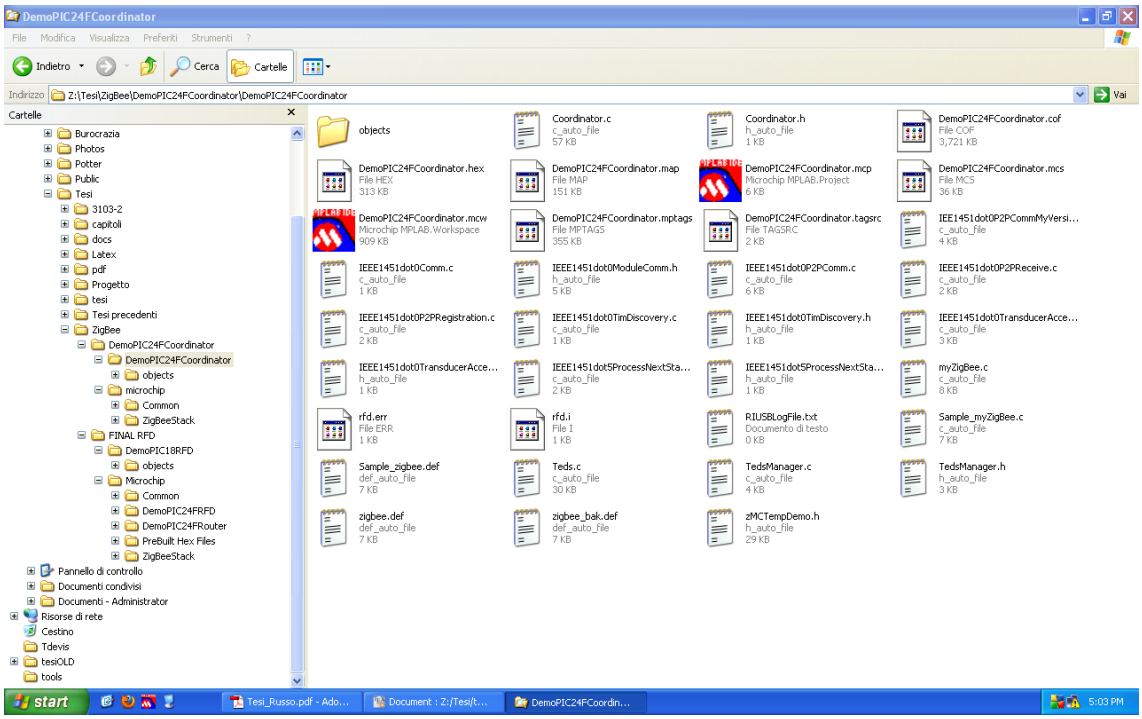

Figura 6.3: File ZigBee Stack

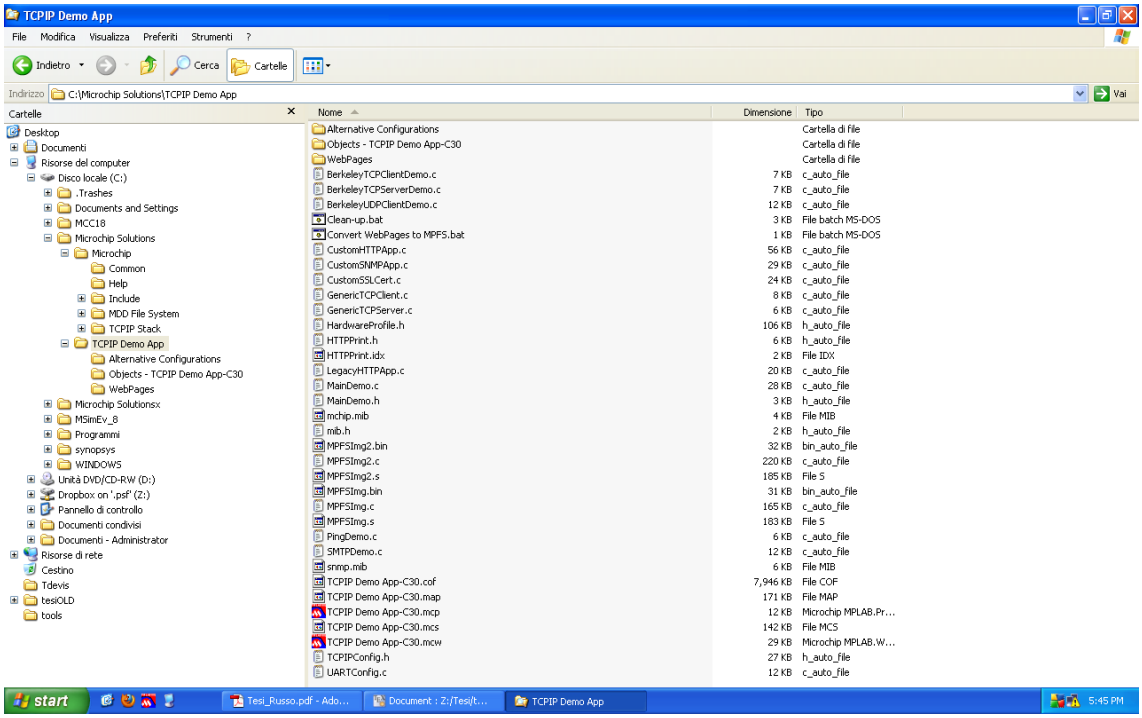

Figura 6.4: File TCP/IP Stack

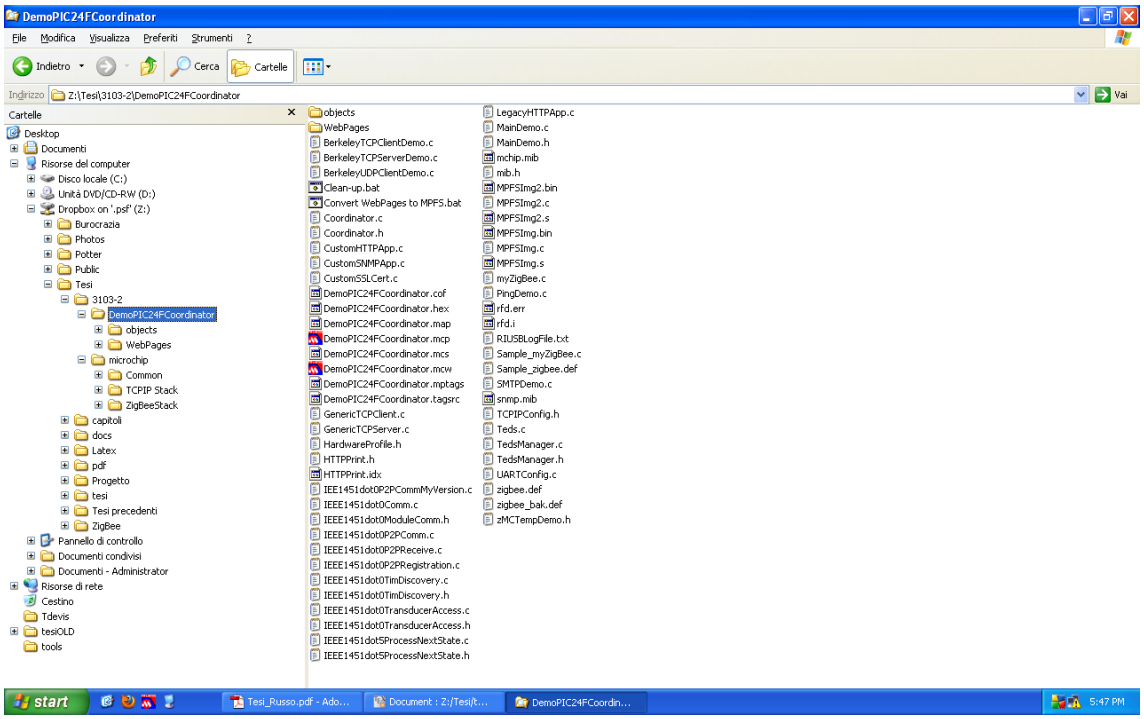

Figura 6.5: Unione risultante

6.2 Workspace

Una volta preparata la cartella con tutti i file necessari al progetto, questi sono stati importati nello spazio di lavoro del Microchip MPLAB IDE. (Si faccia riferimento a [6]). Come illustrato in Figura 6.6 anche qui si é cercato di dividere i files per stack di ap-

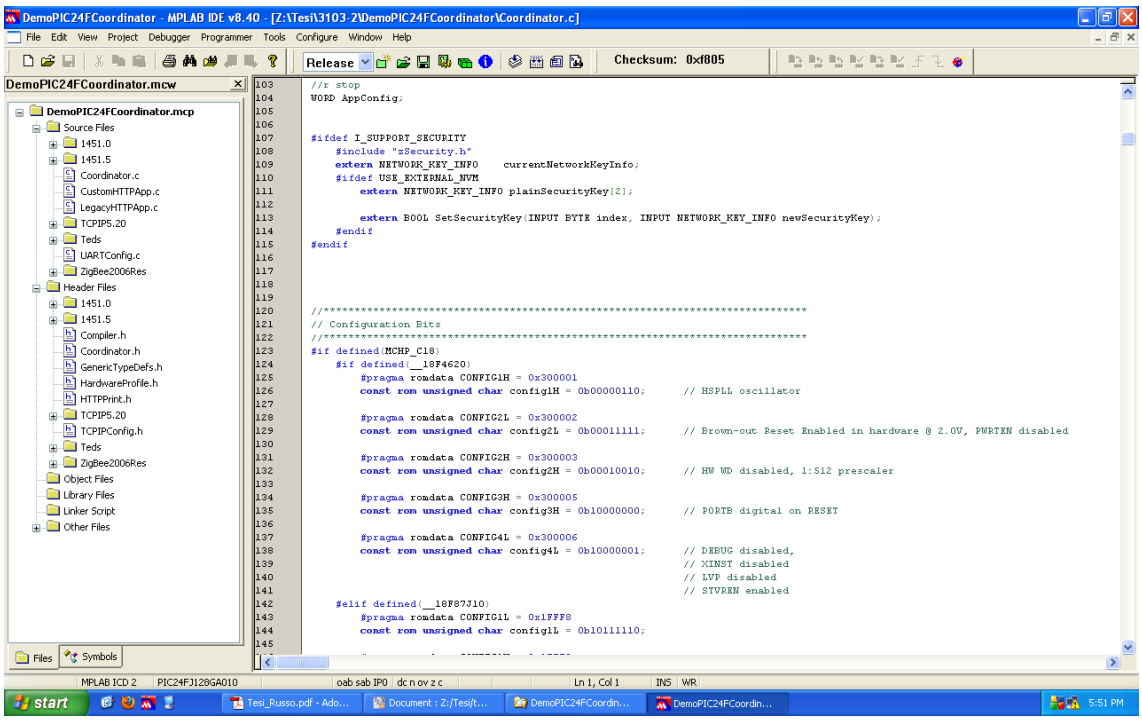

Figura 6.6: Workspace riordinato

partenenza, mettendo bene in evidenza anche i file riguardanti i TEDS e lo strato di convergenza 1451.X.

6.3 Analisi dei problemi incontrati

6.3.1 Definizione dei tipi di dato

Il primo grosso problema incontrato nell'integrazione dei due stack, riguarda la definizione di tipi di dati non standard. Ciascuno stack infatti viene fornito da Microchip corredato da un file header nel quale sono raccolte le definizioni delle strutture dati non elementari necessarie per eseguire procedure e funzioni.

Un esempio di queste strutture é visibile nel frammento di codice 6.1, dove viene definito

```
typedef union _WORD_VAL
{
    BYTE v[2];
    WORD Val;
    struct
    {
        BYTE LSB;
        BYTE MSB;
    } byte;
} WORD_VAL;

#define LSB(a) ((a).v[0])
#define MSB(a) ((a).v[1])
```

Listing 6.1: Definizione del tipo dato "WORDVAL", generic.h

il tipo di dato WORD_VAL. É interessante notare come mediante l'utilizzo della coppia di parole chiave "typedef" e "struct" si possa dichiarare un tipo di dato senza conoscerene a priori le dimensioni, lasciando al compilatore l'onere di eseguire il cast corretto. (Si veda [3]). Una variabile di tipo WORD_VAL cosí definita, infatti, puó essere utilizzata come un vettore di byte, una word o una coppia di byte separati.

Ai fini del progetto é stato necessario unire i due file header, partendo da "GenericTypeDefs.h" e integrando alcune definizioni, ma soprattutto uniformando quelle comuni. Si é riscontrato infatti che il file "GenericTypeDefs.h", distribuito con lo stack TCP/IP é piú recente dell'analogo "generic.h" distribuito con lo stack ZigBee, ed in esso sono presenti molti tipi di dato comuni, ma definiti con strutture differenti, diverse nella nomenclatura e notevolmente arricchite in termini di campi. Si confrontino ad esempio i listati 6.2 e 6.3: Nella nuova versione del file all'operatore é consentita l'indicizzazione bit a bit di una variabile di tipo WORD_VAL (necessaria per alcune funzioni TCP/IP), ma il nome dei campi byte (LB ed HB), non coerenti con le precedenti definizioni, crea numerosi errori in fase di compilazione del codice relativo allo stack ZigBee.

Per risolvere questi conflitti si é ricorso alla verifica dell'uso di ogni singolo campo di ogni struttura, cercando quale stack lo utilizzasse con maggior frequenza. Qualora il campo fosse utilizzato unicamente da file dello stack TCP/IP, avrebbe mantenuto il nome corrente, in caso contrario, avrebbe assunto il nome della versione precedente. Nell'estratto di codice 6.4 é possibile vedere il risultato di questa integrazione, limitato al giá citato tipo di dato WORD_VAL[1]. Le righe di codice modificate appaiono con il marcatore "//rr".

[1] Il processo é stato eseguito su tutte le definizioni di tipo di dato.

```
typedef union _WORD_VAL
{
    BYTE v[2];
    WORD Val;
    struct
    {
        BYTE LSB;
        BYTE MSB;
    } byte;
} WORD_VAL;

#define LSB(a) ((a).v[0])
#define MSB(a) ((a).v[1])
```

Listing 6.2: Definizione del tipo dato "WORDVAL", generic.h

```
typedef union
{
     WORD Val;
     BYTE v[2] __PACKED;
    struct __PACKED
    {
          BYTE LB;
          BYTE MB;
    } byte;
    struct __PACKED
    {
        __EXTENSION BYTE b0:1;
        __EXTENSION BYTE b1:1;
        __EXTENSION BYTE b2:1;
        __EXTENSION BYTE b3:1;
        __EXTENSION BYTE b4:1;
        __EXTENSION BYTE b5:1;
        __EXTENSION BYTE b6:1;
        __EXTENSION BYTE b7:1;
        __EXTENSION BYTE b8:1;
        __EXTENSION BYTE b9:1;
        __EXTENSION BYTE b10:1;
        __EXTENSION BYTE b11:1;
        __EXTENSION BYTE b12:1;
        __EXTENSION BYTE b13:1;
        __EXTENSION BYTE b14:1;
        __EXTENSION BYTE b15:1;
    } bits;
} WORD_VAL, WORD_BITS;
```

Listing 6.3: Definizione del tipo dato "WORDVAL", GenericTypeDefs.h

```
typedef union _WORD_VAL //rr
{
     BYTE v[2] __PACKED;
    WORD Val;
    struct __PACKED
    {
        //rr BYTE LB;
        //rr BYTE HB;
          BYTE LSB;
          BYTE MSB;
    } byte;
    struct __PACKED
    {
        __EXTENSION BYTE b0:1;
        __EXTENSION BYTE b1:1;
        __EXTENSION BYTE b2:1;
        __EXTENSION BYTE b3:1;
        __EXTENSION BYTE b4:1;
        __EXTENSION BYTE b5:1;
        __EXTENSION BYTE b6:1;
        __EXTENSION BYTE b7:1;
        __EXTENSION BYTE b8:1;
        __EXTENSION BYTE b9:1;
        __EXTENSION BYTE b10:1;
        __EXTENSION BYTE b11:1;
        __EXTENSION BYTE b12:1;
        __EXTENSION BYTE b13:1;
        __EXTENSION BYTE b14:1;
        __EXTENSION BYTE b15:1;
    } bits;
} WORD_VAL, WORD_BITS;
#define LSB(a)   ((a).v[0]) //rr
#define MSB(a)   ((a).v[1]) //rr
```

Listing 6.4: Definizione del tipo dato ”WORDVAL”, GenericTypeDefs.h

Interessante notare come, in questa fase, sia importante anche l'ordine della dichiarazione dei campi.

6.3.2 Direttive di compilazione

Un altro problema affrontato nella fase di integrazione dei due stack, é la definizione diversa delle direttive di compilazione. Ciascuno stack infatti viene fornito di un file header "compiler.h" nel quale vengono definite direttive e parole chiave utili per aumentare la compatibilitá tra compilatori differenti. Per la realizzazione del progetto si é deciso di mantenere il file originale dello stack ZigBee, integrandolo con le librerie e alcune definizioni necessarie per la compilazione dello stack TCP/IP. (Nel codice 6.5 si possono identificare le righe aggiunte grazie al marcatore "//rr").

```
#ifndef COMPILER_H
#define COMPILER_H

#if !defined(_WIN32) && !defined(__C30__)
    #define MCHP_C18
#endif

#if defined(MCHP_C18) && defined(HITECH_C18)
    #error "Invalid Compiler selection."
#endif

#if !defined(MCHP_C18) && !defined(__C30__)
    #error "Compiler not supported."
#endif

#if defined(MCHP_C18)
    #include <p18cxxx.h> // p18cxxx.h must have current processor
                         // defined.
     #include <stdlib.h>

#elif defined(__PIC24F__) // Microchip C30 compiler
      // PIC24F processor
      #include <p24Fxxxx.h>
#elif defined(__PIC24H__) // Microchip C30 compiler
      // PIC24H processor
      #include <p24Hxxxx.h>
#elif defined(__dsPIC33F__) // Microchip C30 compiler
      // dsPIC33F processor
      #include <p33Fxxxx.h>
#else
      #error Unknown processor or compiler. See Compiler.h
#endif
```

Listing 6.5: Direttive di compilazione, compiler.h

```
33 #include <stdio.h>  //rr
34 #include <stdlib.h>  //rr
35 #include <string.h>  //rr
36
37 #define PTR_BASE  WORD //rr
38 #define ROM_PTR_BASE WORD //rr
39
40 //rr Definitions that apply to all except Microchip MPLAB C Compiler for PIC18 MCUs (formerly C18)
41 #if !defined(__18CXX) || (defined(HI_TECH_C) && defined(__PICC18__))
42     #define memcmppgm2ram(a,b,c) memcmp(a,b,c)
43     #define strcmppgm2ram(a,b)  strcmp(a,b)
44     #define memcpypgm2ram(a,b,c) memcpy(a,b,c)
45     #define strcpypgm2ram(a,b)  strcpy(a,b)
46     #define strncpypgm2ram(a,b,c) strncpy(a,b,c)
47     #define strstrrampgm(a,b)  strstr(a,b)
48     #define strlenpgm(a)   strlen(a)
49     #define strchrpgm(a,b)   strchr(a,b)
50     #define strcatpgm2ram(a,b)  strcat(a,b)
51 #endif
52
53
54 #if defined(MCHP_C18)
55     #define ROM rom
56     #define NOP() Nop()
57     #define CLRWDT() ClrWdt()
58     #define RESET() Reset()
59     #define SLEEP() Sleep()
60     #define DISABLE_WDT() (WDTCONbits.SWDTEN = 0)
61     #define ENABLE_WDT() (WDTCONbits.SWDTEN = 1)
62     #define TBLWTPOSTINC() _asm tblwtpostinc _endasm
63
64 #elif defined(__C30__)
65     #define ROM     const
66     #define memcmppgm2ram(a,b,c) memcmp(a,b,c)
67     #define memcpypgm2ram(a,b,c) memcpy(a,b,c)
68     #define strcpypgm2ram(a, b)  strcpy(a,b)
69     #define Reset()     asm("reset")
70     #define SLEEP()     Sleep()
71     #define CLRWDT()    ClrWdt()
72     #define NOP()     Nop()
73     #define DISABLE_WDT()   (RCONbits.SWDTEN = 0)
74     #define ENABLE_WDT()   (RCONbits.SWDTEN = 1)
75
76 #endif
77
78 #ifdef MCHP_C18
79
80     #define RF_INT_PIN PORTBbits.RB0
81     #define RFIF  INTCONbits.INT0IF
82     #define RFIE  INTCONbits.INT0IE
83     #define TMRL  TMR0L
84     #define TMRH  TMR0H
85
86 #elif defined(__C30__)
87
88     #define RF_INT_PIN PORTEbits.RE8
89     #define RFIF IFS1bits.INT1IF
90     #define RFIE IEC1bits.INT1IE
91     #define TMRL  TMR2
92     #define TMRH  TMR3
93
94 #endif
95
96 #endif
```

Listing 6.6: Direttive di compilazione - segue da 6.5, compiler.h

6.3.3 Definizione di funzioni comuni

L'ultimo grosso ostacolo all'integrazione dei due stack é rappresentato dalla definizione nei due stack, di funzioni omonime. Entrambi gli stack, per inizializzare il MAC layer, utilizzano una funzione chiamata "MACinit", per ricevere il pacchetto dal buffer usano "MACGet" e per scartarlo, in maniera da poter processare il pacchetto successivo, usano "MACDiscardRx". Questa notazione, se da un lato aiuta il programmatore (permette - in linea di principio - di scrivere applicazioni indipendenti dallo standard utilizzato) dall'altro pone delle grosse limitazioni e lascia intendere che gli stack non siano pensati per cooperare. Con questa implementazione infatti, tralasciando gli ovvi errori in compilazione (duplice definizione di funzione), non ci sarebbe modo di distinguere, a livello di applicazione, quale mac layer inizializzare o quale pacchetto scartare. Per risolvere il problema, si é posto il prefisso "e" a tutte le funzioni relative il mac layer dello stack ZigBee; sia in fase di definizione, che a livello applicazione. Entrambi gli stack, per la

```
1  void MACInit(void)
2  {
3      BYTE i;
4
5      TxHeader = TX_HEADER_START;
6      TxData = TX_DATA_START;
7
8      /* clear the indirect buffers */
9      for(i=0;i<(sizeof(macIndirectBuffers)/sizeof(MAC_INDIRECT_BUFFER));i++)
10     {
11         macIndirectBuffers[i].buffer = NULL;
12     }
13
14     params.MLME_RESET_request.SetDefaultPIB = TRUE;
15     MACTasks(MLME_RESET_request);
16
17     macTasksPending.Val = 0;
18     currentPacket.info.Val = 0;
19
20     #if defined(I_AM_FFD)
21         macStatus.Val = 0;
22     #endif
23
24     #if defined(I_SUPPORT_FREQUENCY_AGILITY) && defined(I_AM_NWK_MANAGER)
25         EdRecords = NULL;
26     #endif
27 }
```

Listing 6.7: Funzione MACInit, zMACMRF24J40.c

gestione delle temporizzazioni fanno un uso intensivo della funzione "TickGet", che restituisce il tempo trascorso dall'attivazione di un timer interno al microcontrollore. Anche per questa funzione si é adottata la soluzione del prefisso "e" ma come si vedrá in segui-

```
void eMACInit(void)
{
    BYTE i;

    // Set up the SPI module on the PIC for communications with the ENC28J60
    ENC_CS_IO = 1;
    ENC_CS_TRIS = 0; // Make the Chip Select pin an output

#if defined(__18CXX)
    ENC_SCK_TRIS = 0;
    ENC_SDO_TRIS = 0;
    ENC_SDI_TRIS = 1;
#endif

    // If the RESET pin is connected, take the chip out of reset
#if defined(ENC_RST_IO)
    ENC_RST_IO = 1;
    ENC_RST_TRIS = 0;
#endif

    // Set up SPI
    ClearSPIDoneFlag();
#if defined(__18CXX)
    ENC_SPICON1 = 0x20; // SSPEN bit is set, SPI in master mode, FOSC/4,
                        // IDLE state is low level
    ENC_SPISTATbits.CKE = 1;// Transmit data on rising edge of clock
    ENC_SPISTATbits.SMP = 0;// Input sampled at middle of data output time
#elif defined(__C30__)
    ENC_SPISTAT = 0; // clear SPI
    #if defined(__PIC24H__) || defined(__dsPIC33F__)
        ENC_SPICON1 = 0x0F; // 1:1 primary prescale, 5:1 secondary prescale (8MHz @ 40MIPS)
    // ENC_SPICON1 = 0x1E; // 4:1 primary prescale, 1:1 secondary prescale (10MHz @ 40MIPS, Doesn't work.
          CLKRDY is incorrectly reported as being clear. Problem caused by dsPIC33/PIC24H ES silicon bug.)
    #elif defined(__PIC24F__)
    // ENC_SPICON1 = 0x1F; // 1:1 prescale broken on PIC24F ES silicon (16MHz @ 16MIPS)
        ENC_SPICON1 = 0x1B; // 1:1 primary prescale, 2:1 secondary prescale (8MHz @ 16MIPS)
    #else // dsPIC30F
        ENC_SPICON1 = 0x17; // 1:1 primary prescale, 3:1 secondary prescale (10MHz @ 30MIPS)
    #endif
    ENC_SPICON2 = 0;
    ENC_SPICON1bits.CKE = 1;
    ENC_SPICON1bits.MSTEN = 1;
    ENC_SPISTATbits.SPIEN = 1;
#elif defined(__C32__)
    ENC_SPIBRG = (GetPeripheralClock()-1ul)/2ul/ENC_MAX_SPI_FREQ;
     ENC_SPICON1bits.SMP = 1; // Delay SDI input sampling (PIC perspective) by 1/2 SPI clock
    ENC_SPICON1bits.CKE = 1;
    ENC_SPICON1bits.MSTEN = 1;
    ENC_SPICON1bits.ON = 1;
#endif

    // RESET the entire ENC28J60, clearing all registers
    // Also wait for CLKRDY to become set.
    // Bit 3 in ESTAT is an unimplemented bit. If it reads out as '1' that
    // means the part is in RESET or there is something wrong with the SPI
    // connection. This loop makes sure that we can communicate with the
    // ENC28J60 before proceeding.
    do
    {
        SendSystemReset();
        i = ReadETHReg(ESTAT).Val;
    } while((i & 0x08) || (~i & ESTAT_CLKRDY));

    // Start up in Bank 0 and configure the receive buffer boundary pointers
    // and the buffer write protect pointer (receive buffer read pointer)
    WasDiscarded = TRUE;
    NextPacketLocation.Val = RXSTART;

    WriteReg(ERXSTL, LOW(RXSTART));
    WriteReg(ERXSTH, HIGH(RXSTART));
    WriteReg(ERXRDPTL, LOW(RXSTOP)); // Write low byte first
    WriteReg(ERXRDPTH, HIGH(RXSTOP)); // Write high byte last
```

Listing 6.8: Funzione eMACInit, ENC28J60.c

```
72      WriteReg(ERXNDL, LOW(RXSTOP));
73      WriteReg(ERXNDH, HIGH(RXSTOP));
74      WriteReg(ETXSTL, LOW(TXSTART));
75      WriteReg(ETXSTH, HIGH(TXSTART));
76
77      // Write a permanant per packet control byte of 0x00
78      WriteReg(EWRPTL, LOW(TXSTART));
79      WriteReg(EWRPTH, HIGH(TXSTART));
80      MACPut(0x00);
81
82
83      // Enter Bank 1 and configure Receive Filters
84      // (No need to reconfigure - Unicast OR Broadcast with CRC checking is
85      // acceptable)
86      // Write ERXFCON_CRCEN only to ERXFCON to enter promiscuous mode
87
88      // Promiscious mode example:
89      //BankSel(ERXFCON);
90      //WriteReg((BYTE)ERXFCON, ERXFCON_CRCEN);
91
92      // Enter Bank 2 and configure the MAC
93      BankSel(MACON1);
94
95      // Enable the receive portion of the MAC
96      WriteReg((BYTE)MACON1, MACON1_TXPAUS | MACON1_RXPAUS | MACON1_MARXEN);
97
98      // Pad packets to 60 bytes, add CRC, and check Type/Length field.
99  #if defined(FULL_DUPLEX)
100     WriteReg((BYTE)MACON3, MACON3_PADCFG0 | MACON3_TXCRCEN | MACON3_FRMLNEN |
            MACON3_FULDPX);
101     WriteReg((BYTE)MABBIPG, 0x15);
102 #else
103     WriteReg((BYTE)MACON3, MACON3_PADCFG0 | MACON3_TXCRCEN | MACON3_FRMLNEN);
104     WriteReg((BYTE)MABBIPG, 0x12);
105 #endif
106
107     // Allow infinite deferals if the medium is continuously busy
108     // (do not time out a transmission if the half duplex medium is
109     // completely saturated with other people's data)
110     WriteReg((BYTE)MACON4, MACON4_DEFER);
111
112     // Late collisions occur beyond 63+8 bytes (8 bytes for preamble/start of frame delimiter)
113     // 55 is all that is needed for IEEE 802.3, but ENC28J60 B5 errata for improper link pulse
114     // collisions will occur less often with a larger number.
115     WriteReg((BYTE)MACLCON2, 63);
116
117     // Set non-back-to-back inter-packet gap to 9.6us. The back-to-back
118     // inter-packet gap (MABBIPG) is set by MACSetDuplex() which is called
119     // later.
120     WriteReg((BYTE)MAIPGL, 0x12);
121     WriteReg((BYTE)MAIPGH, 0x0C);
122
123     // Set the maximum packet size which the controller will accept
124     WriteReg((BYTE)MAMXFLL, LOW(6+6+2+1500+4)); // 1518 is the IEEE 802.3 specified limit
125     WriteReg((BYTE)MAMXFLH, HIGH(6+6+2+1500+4)); // 1518 is the IEEE 802.3 specified limit
126
127     // Enter Bank 3 and initialize physical MAC address registers
128     BankSel(MAADR1);
129     WriteReg((BYTE)MAADR1, AppConfig.MyMACAddr.v[0]);
130     WriteReg((BYTE)MAADR2, AppConfig.MyMACAddr.v[1]);
131     WriteReg((BYTE)MAADR3, AppConfig.MyMACAddr.v[2]);
132     WriteReg((BYTE)MAADR4, AppConfig.MyMACAddr.v[3]);
133     WriteReg((BYTE)MAADR5, AppConfig.MyMACAddr.v[4]);
134     WriteReg((BYTE)MAADR6, AppConfig.MyMACAddr.v[5]);
135
136     // Disable the CLKOUT output to reduce EMI generation
137     WriteReg((BYTE)ECOCON, 0x00); // Output off (0V)
138     //WriteReg((BYTE)ECOCON, 0x01); // 25.000MHz
139     //WriteReg((BYTE)ECOCON, 0x03); // 8.3333MHz (*4 with PLL is 33.3333MHz)
```

Listing 6.9: Funzione eMACInit - segue da 6.8, ENC28J60.c

```
140     // Get the Rev ID so that we can implement the correct errata workarounds
141     ENCRevID = ReadETHReg((BYTE)EREVID).Val;
142
143     // Disable half duplex loopback in PHY. Bank bits changed to Bank 2 as a
144     // side effect.
145     WritePHYReg(PHCON2, PHCON2_HDLDIS);
146
147     // Configure LEDA to display LINK status, LEDB to display TX/RX activity
148     SetLEDConfig(0x3472);
149
150     // Set the MAC and PHY into the proper duplex state
151 #if defined(FULL_DUPLEX)
152     WritePHYReg(PHCON1, PHCON1_PDPXMD);
153 #elif defined(HALF_DUPLEX)
154     WritePHYReg(PHCON1, 0x0000);
155 #else
156     // Use the external LEDB polarity to determine weather full or half duplex
157     // communication mode should be set.
158     {
159         REG Register;
160         PHYREG PhyReg;
161
162         // Read the PHY duplex mode
163         PhyReg = ReadPHYReg(PHCON1);
164         DuplexState = PhyReg.PHCON1bits.PDPXMD;
165
166         // Set the MAC to the proper duplex mode
167         BankSel(MACON3);
168         Register = ReadMACReg((BYTE)MACON3);
169         Register.MACON3bits.FULDPX = PhyReg.PHCON1bits.PDPXMD;
170         WriteReg((BYTE)MACON3, Register.Val);
171
172         // Set the back-to-back inter-packet gap time to IEEE specified
173         // requirements. The meaning of the MABBIPG value changes with the duplex
174         // state, so it must be updated in this function.
175         // In full duplex, 0x15 represents 9.6us; 0x12 is 9.6us in half duplex
176         WriteReg((BYTE)MABBIPG, PhyReg.PHCON1bits.PDPXMD ? 0x15 : 0x12);
177     }
178 #endif
179
180     BankSel(ERDPTL); // Return to default Bank 0
181
182     // Enable packet reception
183     BFSReg(ECON1, ECON1_RXEN);
184 }//end eMACInit
```

Listing 6.10: Funzione eMACInit - segue da 6.8, ENC28J60.c

```
1 BYTE MACGet(void)
2 {
3     params.PD_DATA_indication.psduLength--;
4     return *params.PD_DATA_indication.psdu++;
5 }
```

Listing 6.11: Funzione MACGet, zMACMRF24J40.c

```
1  BYTE eMACGet()
2  {
3      BYTE Result;
4
5      ENC_CS_IO = 0;
6       ClearSPIDoneFlag();
7
8      #if defined(__C32__)
9      {
10         // Send the opcode and read a byte in one 16-bit operation
11         ENC_SPICON1bits.MODE16 = 1;
12         ENC_SSPBUF = RBM<<8 | 0x00; // Send Read Buffer Memory command plus 8 dummy bits to generate
               clocks for the return result
13         WaitForDataByte(); // Wait until WORD is transmitted
14         ENC_SPICON1bits.MODE16 = 0;
15     }
16     #elif defined(__C30__)
17     {
18         // Send the opcode and read a byte in one 16-bit operation
19         ENC_SPISTATbits.SPIEN = 0;
20         ENC_SPICON1bits.MODE16 = 1;
21         ENC_SPISTATbits.SPIEN = 1;
22         ENC_SSPBUF = RBM<<8 | 0x00; // Send Read Buffer Memory command plus 8 dummy bits to generate
               clocks for the return result
23         WaitForDataByte(); // Wait until WORD is transmitted
24         ENC_SPISTATbits.SPIEN = 0;
25         ENC_SPICON1bits.MODE16 = 0;
26         ENC_SPISTATbits.SPIEN = 1;
27     }
28     #else
29     {
30         // Send the opcode and read a byte in two 8-bit operations
31         ENC_SSPBUF = RBM;
32         WaitForDataByte(); // Wait until opcode/address is transmitted.
33         Result = ENC_SSPBUF;
34
35         ENC_SSPBUF = 0; // Send a dummy byte to receive the register
36                                 // contents.
37         WaitForDataByte(); // Wait until register is received.
38     }
39     #endif
40
41     Result = ENC_SSPBUF;
42     ENC_CS_IO = 1;
43
44     return Result;
45 }//end eMACGet
```

Listing 6.12: Funzione eMACGet, ENC28J60.c

```
1  void MACDiscardRx(void)
2  {
3      nfree(CurrentRxPacket);
4  }
```

Listing 6.13: Funzione MACDiscardRx, zMACMRF24J40.c

```
1  void eMACDiscardRx(void)
2  {
3      WORD_VAL NewRXRDLocation;
4
5      // Make sure the current packet was not already discarded
6      if(WasDiscarded)
7          return;
8      WasDiscarded = TRUE;
9
10     // Decrement the next packet pointer before writing it into
11     // the ERXRDPT registers. This is a silicon errata workaround.
12     // RX buffer wrapping must be taken into account if the
13     // NextPacketLocation is precisely RXSTART.
14     NewRXRDLocation.Val = NextPacketLocation.Val - 1;
15     if(NewRXRDLocation.Val > RXSTOP)
16     {
17         NewRXRDLocation.Val = RXSTOP;
18     }
19
20     // Decrement the RX packet counter register, EPKTCNT
21     BFSReg(ECON2, ECON2_PKTDEC);
22
23     // Move the receive read pointer to unwrite-protect the memory used by the
24     // last packet. The writing order is important: set the low byte first,
25     // high byte last.
26     WriteReg(ERXRDPTL, NewRXRDLocation.v[0]);
27     WriteReg(ERXRDPTH, NewRXRDLocation.v[1]);
28 }
```

Listing 6.14: Funzione eMACDiscardRx, ENC28J60.c

```
TICK TickGet(void)

{
    TICK currentTime;

    #if defined(__18CXX)

        /* copy the byte extension */
        currentTime.byte.b2 = 0;
        currentTime.byte.b3 = 0;

        /* disable the timer to prevent roll over */
        INTCONbits.TMR0IE = 0;

        /* read the timer value */
        currentTime.byte.b0 = TMR0L;
        currentTime.byte.b1 = TMR0H;

        //if an interrupt occured after IE = 0, then we need to figure out if it was
        //before or after we read TMR0L
        if(INTCONbits.TMR0IF)
        {
            if(currentTime.byte.b0<10)
            {
                //if we read TMR0L after the rollover that caused the interrupt flag then we need
                //to increment the 3rd byte
                currentTime.byte.b2++; //increment the upper most
                if(timerExtension1 == 0xFF)
                {
                    currentTime.byte.b3++;
                }
            }
        }

        /* copy the byte extension */
        currentTime.byte.b2 += timerExtension1;
        currentTime.byte.b3 += timerExtension2;

        /* enable the timer*/
        INTCONbits.TMR0IE = 1;
    #elif defined(__dsPIC33F__) || defined(__PIC24F__) || defined(__PIC24H__)
        currentTime.word.w0 = TMR2;
        currentTime.word.w1 = TMR3;
    #else
        #error "Symbol timer implementation required for stack usage."
    #endif
    return currentTime;
}
```

Listing 6.15: Funzione TickGet, SymbolTime.c

```
DWORD eTickGet(void)
{
     GetTickCopy();
     return *((DWORD*)&vTickReading[0]);
}
```

Listing 6.16: Funzione eTickGet, Tick.c

to (Risultati, pag. 6.6) questo accorgimento é necessario ma non sufficiente a risolvere completamente il problema.

6.4 Applicazione

L'integrazione a livello applicazione é stata fatta partendo dal file "Coordinator.c" dello stack ZigBee ed aggiungendo le righe di codice necessarie. Come prima cosa sono stati inseriti gli header TCPIP e MainDemo, dove sono definite le funzioni dello stack TCP/IP. Poi é stata definita la macro THIS_IS_STACK_APPLICATION, la quale indica al compilatore che in questo file si trova il main. L'ultima operazione preliminare, a questo punto, é la dichiarazione della variabile "AppConfig", di tipo APP_CONFIG, che istanzia alcune variabili di supporto allo stack. (Si veda come riferimento l'estratto di codice 6.17). Altre

```
#include "zAPL.h"
#include "zNVM.h"
#include "TedsManager.h"
#include "Coordinator.h"
#include "Console.h"
#include "IEEE1451dot0TimDiscovery.h"
#include "IEEE1451dot0TransducerAccess.h"

//rr start
/*
 * This macro uniquely defines this file as the main entry point.
 * There should only be one such definition in the entire project,
 * and this file must define the AppConfig variable as described below.
 */
#define THIS_IS_STACK_APPLICATION

// Include all headers for any enabled TCPIP Stack functions
#include "TCPIP.h"

// Include functions specific to this stack application
#include "MainDemo.h"

// Declare AppConfig structure and some other supporting stack variables
APP_CONFIG AppConfig;
//r stop
```

Listing 6.17: Integrazione a livello Applicazione, Preambolo, Coordinator.c

aggiunte essenziali sono state fatte all'interno del main, prima del ciclo while di loop. In questa sezione (si veda l'estratto 6.18) si é aggiunta tutta la sequenza di inizializzazione dello stack, a partire dalla procedura di inizializzazione della scheda, l'inizializzazione del display e di tutto l'hardware specifico. Si noti l'inserimento qui della funzione "TickInit", cruciale per la gestione delle temporizzazioni, che si occupa di inizializzare i contatori.

Seguono poi le inizializzazioni dell'Application layer, quindi del file system, di tutte le applicazioni e della macchina a stati dello stack. Subito fuori dal main sono state inserite

```
////////////////////////////////////////////////////////////////////////////
//        MAIN
////////////////////////////////////////////////////////////////////////////
#if defined(__C30__)
    int main(void)
#else
    void main(void)
#endif
{
//rr start
      // Initialize application specific hardware
      InitializeBoard();

      #if defined(USE_LCD)
      // Initialize and display the stack version on the LCD
      LCDInit();

      DelayMs(100);
      strcypypgm2ram((char*)LCDText, "TCP/IP-ZigBee GW" "2010 By Russo P.");
      LCDUpdate();
      DelayMs(1000);
      strcypypgm2ram((char*)LCDText, "TCP/IP-ZigBee GW" " ");
      LCDUpdate();
      #endif

      // Initialize stack-related hardware components
    TickInit();

      #if defined(STACK_USE_MPFS) || defined(STACK_USE_MPFS2)
      MPFSInit();
      #endif

      // Initialize Stack and application related NV variables into AppConfig.
      InitAppConfig();

      // Initialize core stack layers (MAC, ARP, TCP, UDP) and
      // application modules (HTTP, SNMP, etc.)
    StackInit();
//rr end
```

Listing 6.18: Integrazione a livello Applicazione, Main, Coordinator.c

le definizioni delle funzioni DisplayIPValue, ProcessIO, InitializeBoard e InitAppConfig, qui non riportate perché prive di modifiche sostanziali. Nell'apposita sezione non-ZigBee, invece, é stato inserito il nucleo delle operazioni necessarie al funzionamento dello stack TCP/IP, ovvero la sequenza StackTask - StackApplications - ProcessIO. (Si veda l'estratto di codice 6.19). Si é scelto di introdurre qui anche il codice per visualizzare sul display della board eventuali cambi di indirizzo IP. Assolutamente non necessario, risulta di una certa comoditá in fase di sperimentazione.

```
void ProcessNONZigBeeTasks(void)
{
        // ********************************************************************
        // Place any non-ZigBee related processing here. Be sure that the code
        // will loop back and execute ZigBeeTasks() in a timely manner.
        // ********************************************************************
          // rr start
          RUN ^= 1;
          static DWORD dwLastIP = 0;

          // This task performs normal stack task including checking
        // for incoming packet, type of packet and calling
        // appropriate stack entity to process it.
        StackTask();

        // This tasks invokes each of the core stack application tasks
        StackApplications();

          ProcessIO();

          // If the local IP address has changed (ex: due to DHCP lease change)
        // write the new IP address to the LCD display, UART, and Announce
        // service

          if(dwLastIP != AppConfig.MyIPAddr.Val)
          {
                dwLastIP = AppConfig.MyIPAddr.Val;
                DisplayIPValue(AppConfig.MyIPAddr);
        }
          //rr end
}
```

Listing 6.19: Integrazione a livello Applicazione, non-ZigBee, Coordinator.c

6.5 Risultati

Allo stato attuale, con le modifiche e le impostazioni suggerite in questo elaborato, NCAP e relativo web server si inizializzano correttamente, entrambe le macchine a stati eseguono i loro cicli, ma manca un passo importante: il passaggio del payload dallo stack ZigBee allo stack TCP/IP cosí che possa essere rappresentato sulle pagine del webserver. Questo passaggio, cruciale nella realizzazione del gateway, non é stato affrontato a causa di un problema nella trasmissione wireless. Si é constatato infatti che pur senza manifestare errori in compilazione o nella sequenza degli stati, dopo l'integrazione con lo stack TCP/IP, l'NCAP non riesce ad inizializzare la rete, non rileva il nodo e di conseguenza non riesce ad instaurare la comunicazione. Si ritiene che alla base di questo problema ci sia una disfunzione nella gestione delle temporizzazioni, probabilmente generata da un conflitto hardware sui contatori integrati nel microcontrollore. L'accortezza di separare le funzioni di temporizzazione dei due stack dando loro un nome differente, seppure logicamente valida, non é sufficiente a garantire un corretto funzionamento; da una piú attenta analisi

di dette funzioni (Si veda ad esempio l'estratto di codice 6.15) si puó notare come queste facciano un uso intensivo di TMR0, identificatore del primo timer integrato nel microcontrollore. Seppur nominate in modo diverso, le funzioni di temporizzazione si trovano quindi ad agire sul medesimo hardware, rendendo impossibile la comunicazione.

6.6 Sviluppi futuri

Il primo aspetto fondamentale da approfondire, come detto, é la gestione delle temporizzazioni. Con l'ausilio del datasheet del microcontrollore (Si veda [9], pagine 105 e seguenti) sará necessario rimappare le funzioni di timing di uno dei due stack affinché si appoggi ad un timer diverso. Il secondo aspetto da affrontare, altrettanto cruciale, é lo studio del passaggio del payload dallo stack ZigBee allo stack TCP/IP. Il terzo aspetto da affrontare, per completare la realizzazione del gateway, é la realizzazione dell'interfaccia per la gestione della comunicazione.

Ci sarebbero poi una serie di sviluppi non essenziali per il funzionamento, ma supportati dagli stack e interessanti da implementare. Tra questi il supporto SSL per la crittografia dei dati sulla rete ethernet e il protocollo di messaggistica SMTP, il primo abbinato ad una sequenza di autenticazione di tipo nome utente e password proteggerebbe il gateway da accessi indesiderati mentre il secondo consentirebbe una gestione via email degli eventi, come ad esempio allert provenienti da sensori o report periodici dello stato degli stessi. Per un interfacciamento meno estetico ma piú orientato alle macchine a controllo numerico, si potrebbe infine sviluppare (parallelamente all'interfaccia web) un servizio di collegamento TelNet, cosd́a consentire l'interrogazione diretta della rete ZigBee via ethernet, in maniera testuale.

Conclusioni

Il lavoro svolto ha dimostrato la realizzabilitá di un gateway TCP/IP - ZigBee su una board di sviluppo Microchip Explorer_16. L'impiego di una struttura di protocolli molto articolata, nonostante il consistente impegno di risorse, risulta pienamente supportata dal processore utilizzato. I conflitti hardware incontrati, possono essere agevolmente superati ricorrendo alle impostazioni e modifiche suggerite in questo elaborato.

L'implementazione completa del gateway non é stata possibile, ma si ritiene di aver creato tutti i presupposti per una realizzazione che non dovrebbe risultare troppo complicata.

Bibliografia

[1] Ieee 1451.0 standard for a smart transducer interface for sensors and actuators common functions, communication protocols, and transducer electronic data sheet (teds) formats, 2007.

[2] Ieee 1451.5 standard for a smart transducer interface for sensors and actuators wireless communication protocols and transducer electronic data sheet (teds) formats, 2007.

[3] M. J. Augenstein A. M. Tenenbaum, Y. Langsam. *Data structures using C.* Prentice Hall, 1990.

[4] ZigBee Alliance. Zigbee specification, 2006.

[5] G. Giorgi. Reti di sensori ieee 1451, 2009. (slideshow).

[6] Microchip Technology Inc. *MPLAB IDE USERS GUIDE.* Microchip Technology Inc., 2005.

[7] Microchip Technology Inc. *Explorer 16 Development Board Users Guide.* Microchip Technology Inc., 2006.

[8] Microchip Technology Inc. *PIC24F Family Reference Manual, Sect. 23 - SPI.* Microchip Technology Inc., 2007.

[9] Microchip Technology Inc. *PIC24FJ128GA010 Family Data Sheet, 64/80/100-Pin General Purpose, 16-Bit Flash Microcontrollers.* Microchip Technology Inc., 2007.

[10] Microchip Technology Inc. *ENC28J60 Data Sheet, Stand-Alone Ethernet Controller with SPI Interface.* Microchip Technology Inc., 2008.

[11] Microchip Technology Inc. *Explorer 16 Development Board Errata.* Microchip Technology Inc., 2008.

[12] Microchip Technology Inc. *MRF24J40 Data Sheet, IEEE 802.15.4TM 2.4 GHz RF Transceiver.* Microchip Technology Inc., 2008.

[13] Derrick P. Lattibeaudiere. *AN1232 - Microchip ZigBee-2006 Residential Stack Protocol.* Microchip Technology Inc., 2008.

[14] Nilesh Rajbharti. *AN833 - The Microchip TCP/IP Stack.* Microchip Technology Inc., 2008.

Ringraziamenti

Grazie ai miei genitori e a tutta la mia Famiglia. Solo grazie a loro sono potuto arrivare a questo traguardo, per me tanto importante.

Grazie ad Eleonora, Pietro e Federico, con i quali non posso condividere questa gioia, ma che tanto mi hanno insegnato della vita.

Grazie a tutti gli amici fraterni, recenti e di vecchia data, sui quali so di poter sempre contare, che mi hanno sostenuto e supportato in questo lungo cammino.

Grazie a Monica, che mi sopporta.

Grazie al Professor Narduzzi, per l'enorme disponibilitá e per avermi dato l'opportunitá di lavorare ad un proggetto tanto appassionante.

Grazie Devis, Alberto e tutti i ragazzi del laboratorio di misure, con i quali ho passato cosí tante ore negli ultimi mesi.

Grazie a Francesca, per il tempo speso insieme a studiare e per le innumerevoli pause caffé.

LaTeX