UNIVERSITÁ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

IMPLEMENTAZIONE SU MICROCONTROLLORI PIC DELLO STRATO DI CONVERGENZA TRA PROTOCOLLI ZIGBEE E IEEE1451 IN RETI DI SENSORI WIRELESS

LAUREANDO: Paolo Zigoni

RELATORE: Prof. Claudio Narduzzi

CORSO DI LAUREA TRIENNALE IN INGEGNERIA ELETTRONICA

Padova, 18 Febbraio 2010

Anno Accademico 2009/2010

Alla mia famiglia.

Sommario

Scopo di questo elaborato è l'implementazione dello strato di convergenza tra protocolli ZigBee e 1451 in una rete di sensori wireless. Il progetto prevede la realizzazione di una rete di smart sensors plug 'n' play che rispondano alle specifiche dello standard IEEE 1451 il quale disciplina le comunicazioni tra dispositivi wireless utilizzanti i più diffusi protocolli di ricetrasmissione.
Ciò permette di facilitare la creazione di reti wireless con diverse tecnologie in quanto, vincolando queste reti alle specifiche di uno standard comune, la loro creazione, utilizzo e gestione risultano indipendenti dal tipo e dalla tecnologia della rete stessa. Altro aspetto fondamentale è che abbracciando lo standard IEEE1451 è possibile interrogare da remoto reti diverse (ZigBee e Wi-Fi per esempio) utilizzando i medesimi comandi.
Nello specifico il progetto prevede, utilizzando lo standard di comunicazione ZigBee, la realizzazione di una connessione P2P tra NCAP (ZigBee Coordiantor) e WTIM (ZigBee EndDevice) . Il WTIM è un sensore wireless utilizzato come sonda di temperatura mentre l' NCAP è un dispositivo che ha il compito di creare e gestire la comunicazione con il WTIM, e permettere l'accesso e l'interrogazione di quest'ultimo da remoto.
La necessità di rispettare scrupolosamente gli standard e svariati problemi riscontrati in fase di sviluppo nell'utilizzo di software preesistente hanno portato alla riscrittura di due firmware nuovi per NCAP e WTIM, impostati in modo tale da consentirne un ulteriore sviluppo in futuro date le enormi opzioni messe a disposizione dallo standard IEEE1451.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione allo standard IEEE1451

I trasduttori, definiti qui come sensori o attuatori, vengono impiegati nei più diversi rami dell'industria e vengono utilizzati oggi per le più svariate applicazioni, che vanno dal controllo automatico di macchinari per la produzione industriale all'accensione di un antifurto con un SMS al controllo dei satelliti geostazionari e via dicendo. Esistono dunque molteplici settori del mercato in cui si possono collocare i trasduttori, e per ognuno di essi esistono diverse aziende che producono trasduttori e che sono sempre alla ricerca di soluzioni per produrre reti di sensori intelligenti cablate o wirless a basso costo. Sono quindi oggi disponibili svariate implementazioni di reti di sensori, ognuna con i suoi pregi e difetti per una determinata classe di applicazioni.
Interfacciare degli smart transducers con questa varietà di reti e supportare la molteplicità dei protocolli di comunicazione impiegati in tali reti richiede pertanto notevoli sforzi e costi alle ditte che producono trasduttori. Lo sviluppo e l'accettazione universale di standard come l' IEEE 1451 permette la risoluzione di questi problemi.

L' IEEE 1451, una famiglia di Smart Transducer Interface Standards, descrive un set di interfacce di comunicazione comuni, aperte e network-independent per connettere trasduttori (sensori o attuatori) a microprocessori, sistemi di strumentazione e reti di controllo.
Il punto chiave di questo standard è la definizione dei Transducer Electronic Data Sheets (TEDS). Il TEDS è un dispositivo di memoria collegato al trasduttore che contiene le informazioni sull' identificativo del trasduttore, la calibrazione, il range di misurazione ecc.
Lo standard definisce inoltre due dispositivi: NCAP e TIM.

- Il Transducer interface module (TIM) è il dispositivo a cui spetta la gestione del sensore/trasduttore/attuatore , il condizionamento dell'informazione rilevata, la conversione di tali informazioni e trasmissione delle stesse; è considerato un nodo terminale di rete e si appoggia al nodo della seconda categoria: l' NCAP.

- Il Network Capable Application Processor (NCAP) è il dispositivo al quale spetta la gestione della rete e l'inoltro di messaggi che circolano in essa. Tale dispositivo opera da gateway tra l' utente della rete ed i TIM, nel caso fosse previsto l' accesso dall' esterno.

L'obiettivo dello standard è permettere l'accesso ai dati forniti da trasduttori tramite un set di interfacce comuni indipendentemente dal tipo di rete a cui il trasduttore è connesso (cablata o wirless). Gli standard IEEE1451 attualmente sono sette, ognuno sviluppato per esigenze specifiche; per lo scopo di questo progetto sono stati implementati gli standard IEEE1451.0 e IEEE1451.5.

IEEE1451.0 definisce un set di comandi e operazioni comuni per la famiglia degli standard IEEE1451 tramite il quale è possibile accedere ad ogni sensore o attuatore collegato ad una rete 1451, sia essa cablata o wireless. Caratteristica principale di questo standard è che la sua funzionalità è indipendente dal mezzo fisico di comunicazione tra il TIM e l' NCAP

IEEE1451.5 definisce un'interfaccia trasduttore-NCAP e i TEDS per dispositivi wireless. Protocolli standard di comunicazione wireless come 802.11 (Wi-Fi), 802.15.1 (Bluetooth) o 802.15.4 (ZigBee) sono da considerarsi come alcune delle interfacce fisiche dell' IEEE1451.5. Lo scopo di questa interfaccia è permettere all'utenza di acquisire, utilizzando il medesimo set di comandi, dati da sensori che trasmettono con protocolli di rete diversi.

1.1 Modello di riferimento 1451

In Figura 1 è presentato lo schema funzionale di una connessione P2P tra NCAP e TIM.
Il blocco NCAP IEEE1451.0 Services fornisce le funzioni e i servizi per il modulo comunicazioni e per il modulo applicativo dell' NCAP (non implementato nel progetto). Tali funzioni comprendono l'intero set dei comandi e i TEDS. La Module Communication Interface è l'interfaccia che permette il transito delle informazioni tra il modulo 1451.0 e il modulo 1451.X.

La descrizione fisica di tale interfaccia non è presente nello standard ed è lasciata al progettista in quanto deve essere adattata di volta in volta in funzione del dispositivo utilizzato. Tutti gli altri membri della famiglia 1451 eventualmente adottati appartengono al Communication Module. Nello specifico tale modulo contiene l'implementazione dello standard IEEE1451.5 che è quello utilizzato in questo progetto e che copre le comunicazioni tra dispositivi wireless. Il modulo servizi e quello comunicazioni del TIM sono complementari a quelli dell'NCAP, mentre le interfacce tra modulo servizi e il trasduttore esulano dallo standard IEEE1451 in quanto direttamente correlate all'implementazione fisica dei dispositivi.

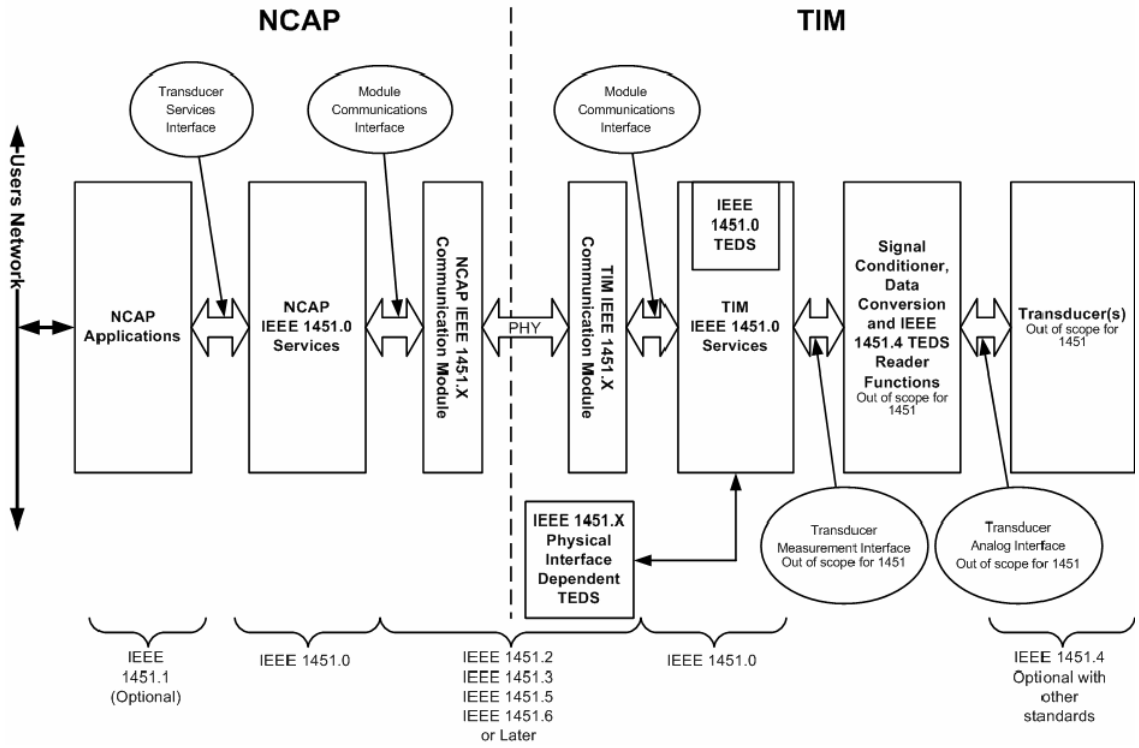


Figura 1.1: Modello di riferimento dello standard

1.2 IEEE1451.5

Lo standard IEEE1451.5 introduce il concetto di Wireless Interface Transducer Module (WTIM) connesso ad un Network-Capable Application Processor(NCAP) Service Module attraverso un canale radio approvato IEEE 802.11, IEEE 802.15, IEEE Bluetooth e ZigBee. Secondo le direttive 1451.5, un WTIM è un dispositivo che comprende un'interfaccia wireless Dot 5 Approved

Radio (Dot5AR), il condizionamento del segnale, la conversione analogica/digitale e uno o più trasduttori (sensori/attuatori). Poichè i WTIM possono avere interfacce Dot5AR diverse tra loro, l'NCAP dovrà avere almeno una Dot5AR per ogni tipo presente nei WTIM a cui deve essere associato. Lo standard IEEE1451 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 TEDS.
La nascita di questo protocollo è dovuta alla volontà di uniformare le specifiche ed accomunare più tecnologie sotto un unico standard aperto, riducendo al minimo il rischio di incompatibilità e i costi di produzione. Lo standard in questione è basato unicamente su interfacce wireless, ma non specifica le caratteristiche fisiche e tecniche né dei trasduttori né del sistema wireless.
Le specifiche dell'IEEE1451.5 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 IEEE1451.

1.2.1 Strato di convergenza (Convergence Layer)

Lo standard IEEE 1451.5 definisce il concetto di convergence layer (strato di convergenza) tra l'entità superiore (IEEE 1451.0) e la rete di trasporto (in questo caso ZigBee), ed ha il compito di tradurre i comandi provenienti dal livello superiore in comandi specifici comprensibili al livello inferiore e viceversa (Figura 1.2).

1.2.2 Bulk Transfer Profile

Le funzioni dello strato di convergenza 1451.5 sono pensate per essere integrate nell'Application Layer di Zigbee. Nello standard IEEE1451.5 viene definito un Application Profile chiamato Bulk Transfer Profile, che descrive i dispositivi riconosciuti dall'applicazione e le modalità di scambio dei dati tra tali dispositivi.
Un dispositivo definito in un profilo Zigbee prende il nome di application object. I dispositivi riconosciuti nell'Application Profile 1451.5 sono solo due: NCAP e WTIM. In ciascuno di essi è previsto un endpoint, cioè un componente identificabile univocamente che permette all'oggetto di interfacciarsi al resto dello stack ZigBee. L'entità 1451.5 è vista come un Application Object Zigbee che si interfaccia al sistema mediante un endpoint definito. La comunicazione

tra endpoint si avvale di strutture dati denominate cluster, che contengono un insieme di attributi rappresentanti lo stato del dispositivo ed i comandi che consentono di comunicare con altri application objects. A ciascun dispositivo sono associati un input ed un output cluster, in tutto quindi si ha: NCAPin, NCAPout, WTIMin, WTIMout. Nel trasferimento dati tra livello 1451.5 e ZigBee, il formato utilizzato è definito nello standard IEEE1451.5 come Bulk Transfer Profile Message Format e prevede solo due tipi di comandi (Set e Set-Response) ed è costituito da quattro campi:

- Command Type è un byte di cui i primi 4 bit più significativi sono bit di status e i 4 bit meno significativi identificano la tipologia del comando. Sono possibili solo due tipi di Command Type; Set se si deve inviare un messaggio (codice 0x01), Set-Response se si deve inviare un acknowledgement (codice 0x09).

- Packet ID è importante nel caso della frammentazione di un pacchetto per la spedizione tramite ZigBee. Ai frammenti di uno stesso pacchetto viene assegnato lo stesso Packet ID e sono identificati mediante il successivo ottetto. Nelle ipotesi di lavoro fatte, non dovrebbe esserci necessità di frammentazione, quindi l'identificativo può essere posto al valore 0x00.

- Sequence Number serve ad indentificare ciascun frammento di un pacchetto. In assenza di frammentazione, anche questo può essere posto al valore 0x00.

- In Octet Data è contenuto il payload, cioè il dato utile che si vuole trasferire. Il primo byte contiene la lunghezza della stringa di ottetti.

1.3 Macchina a stati

Il funzionamento dei dispositivi 1451, per quanto riguarda la loro interazione con lo strato protocollare 1451.5, è determinato dalla definizione di macchine a stati. La macchina a stati definita dallo standard 1451.5 serve a definire il comportamento e a recuperare informazioni sullo status del modulo radio. Lo standard 1451 prevede anche una funzione per recuperare informazioni sullo stato della della macchina a stati locale e remota. L'implementazione di

tale macchina a stati è stata ereditata da un lavoro precedentemente iniziato ed è stata corretta e rivista. Non essendo però lo scopo di questo progetto e non essendo necessaria la sua trattazione per la comprensione del lavoro svolto, non verrà trattata. Si rimanda quindi ai lavori precedenti per un eventuale approfondimento.

1.4 Transducer channel

Il TIM è un dispositivo che può ospitare molteplici sensori/attuatori. Viene definito pertanto dallo standard un Transducer Channel ossia un identificativo che individua un determinato sensore nel TIM. Se ad esempio un TIM dispone di un sensore di temperatura e di un sensore di umidità si definiranno due Transducer Channels con due indirizzi diversi e univoci. In questo progetto il TIM dispone solo di un sensore di temperatura, pertanto è stato definito un unico Transducer Channel, denominato TEMPERATURE, a cui è stato assegnato l'indirizzo 0xAAAA, seguendo le direttive dello standard.

1.5 Strutture dei messaggi

Il modulo servizi 1451.0 e il modulo comunicazioni 1451.5 dialogano tramite messaggi inviati attraverso la Module Communication Interface come sequenze di ottetti, ossia parole di 8 bits. Tali messaggi possono essere di due tipi: Command Messages o Reply Messages, e per ciascuna tipologia è adottato un formato specifico.

1.5.1 Command messages

Questi messaggi sono strutturati in sei ottetti comuni per ogni comando più altri ottetti il cui numero e attributo dipendono dal tipo di comando. La Tabella 1.1 descrive la formattazione dei messaggi, in ordine dall'ottetto più significativo al meno significativo.

- *Destination Transducer Number (16 bits):* è l'identificativo unico del destinatario del messaggio.

- *Class:* i comandi sono organizzati in classi a seconda del tipo. Questo parametro identifica il tipo generico di comando, è sostanzialmente un indice.

- *Command Function:* identifica il comando specifico per la determinata classe selezionata da Command Class.

- *Lunghezza (16 bits):* indica il numero degli otteti dipendenti dal comando. Se la lunghezza del messaggio ricevunto non corrisponde al campo Lunghezza del messaggio di risposta, il messaggio viene scartato e si avvia una segnalazione di errore.

- *Otteti dipendenti dal comando:* sono delle sequenze di byte aggiuntive richieste dal particolare tipo di comando invocato.

Destination TransducerChannel Number (byte più significativo)
Destination TransducerChannel Number (byte meno significativo)
Command Class
Command Function
Lunghezza (byte più significativo)
Lunghezza (byte meno significativo)
Otteti dipendenti dal comando . . .

Tabella 1.1: Struttura messaggi di comando

1.5.2 Reply messages

Sono impiegati per rispondere ad un Command Message. Il formato del messaggio è quello di Tabella 1.2, nel quale:

- *Success/Fail Flag:* se tale ottetto è diverso da zero il comando è stato elaborato correttamente, altrimenti dovrebbe essere sollevata una condizione di errore e di relativa ricerca.

- *Lunghezza (16 bits):* indica il numero di ottetti dipendenti dalla risposta. Se il controllo del messaggio ricevuto rileva che tale parametro è diverso dall'effettiva lunghezza del

messaggio ricevuto (della sua parte reply-dependent), viene avviata la segnalazione di errore.

- *Otteti dipendenti dalla risposta:* questo campo contiene tutti gli ottetti richiesti dallo specifico comando.

I campi Lunghezza e Success/Fail Flag sono due sistemi di controllo dell'integrità del comando (risposta) inviato (ricevuta). Nel caso ci sia un errore nella trasmissione del comando il destinatario rileva incongruenze di formattazione analizzando i vari campi del messaggio e pone a zero il campo Success/Fail Flag che viene successivamente rilevato in risposta dal mittente. Se invece il destinatario riceve ed elabora correttamente il comando ma commette errori nella trasmissione della risposta ci sarà una incongruenza tra il valore del campo Lunghezza e l'effettivo numero di ottetti dipendenti ricevuti.

Success/Fail Flag)
Lunghezza (byte più significativo)
Lunghezza (byte meno significativo)
Otteti dipendenti dalla risposta . . .

Tabella 1.2: Struttura messaggi di risposta

1.5.3 Command Classes

I comandi sono organizzati in classi a seconda del tipo. Questa classificazione è strutturalmente simile all'indicizzazione di un libro; la tipologia di comando è identificata dal parametro *cmdClassID*. In Tabella 1.3 sono elencati gli insiemi dei comandi definiti nello standard.

1.5.4 Command Functions

In questo progetto, l'unica classe di comandi utilizzata è stata la `XdcrOperate`, la quale copre tutte le funzioni di lettura-scrittura da-verso un dispositivo. Per tale classe di comandi, le funzioni definite sono le seguenti:

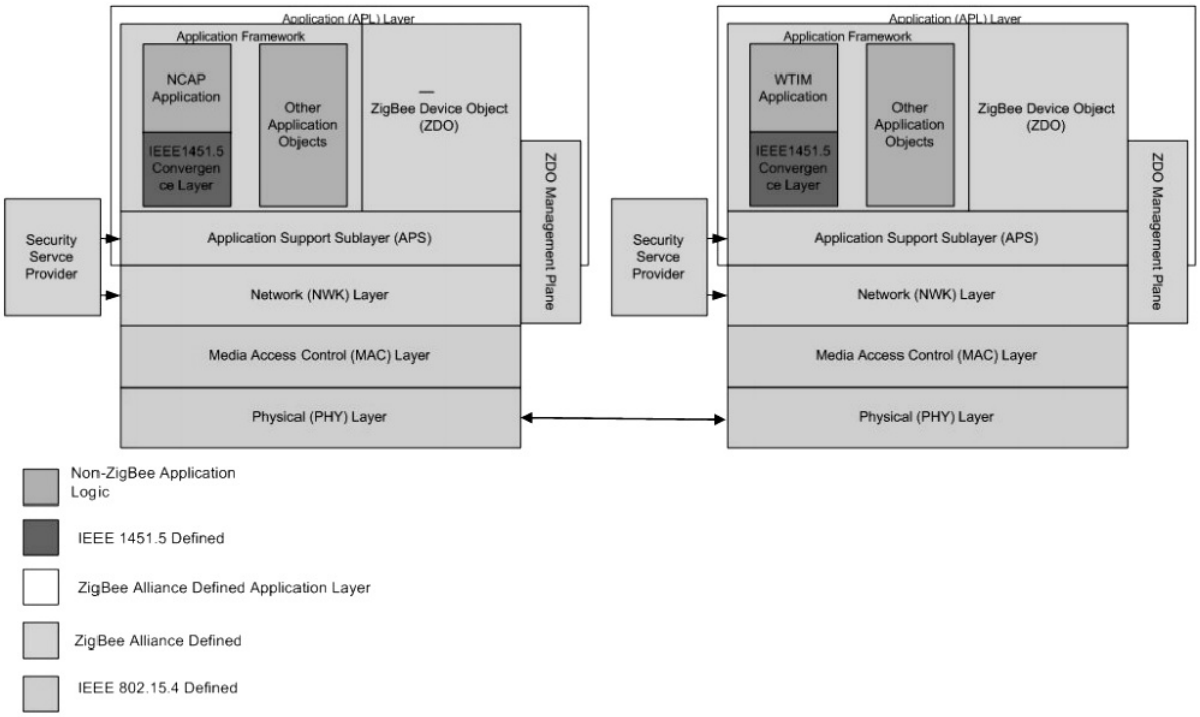


Figura 1.2: PICDEM Z

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

Tabella 1.3: Command Classes

cmdFunctionId	Command
0	Reserved
1	Read TransducerChannel data-set segment
2	Write TransducerChannel data set segment
3	Trigger command
4	Abort Trigger
5-127	Reserved
128-255	Open for manufacturers

Tabella 1.4: Command Functions

L'obbiettivo di questo progetto è la lettura di un dato di temperatura da un sensore, quindi di questa classe di funzioni si è utilizzato solamente il comando `Read TransducerChannel data-set segment`, il cui scopo è avviare la procedura di lettura dati da un sensore. Questo comando presenta ottetti dipendenti solo nel messaggio di risposta. Tali ottetti sono raggruppati in due campi:

- Read Sensor Offset: indica l'offset nel data-set, ossia il punto nel data-set in cui iniziare la lettura del dato. Nel nostro caso tale campo è posto a sempre a zero in quanto il dato di temperatura verrà letto e trasmesso integralmente in una operazione unica.
- Read Sensor Data: è il blocco di dati letto dal sensore. La dimensione di tale campo è funzione del dato letto. Nel nostro caso il dato di temperatura ha una lunghezza costante quindi il numero di byte di tale campo, così come il valore dell'attributo lunghezza saranno sempre costanti.

Read Sensor Offset
Read Sensor Data (n-byte)
. . .

Tabella 1.5: Ottetti dipendenti funzione `Read TransducerChannel data-set segment`

1.6 Protocollo di comunicazione NCAP-WTIM

Lo standard IEEE 1451 si inserisce concettualmente tra utente e protocollo fisico di comunicazione (nel nostro caso ZigBee). L'utente pertanto dialoga con il sensore usando solamente istruzioni dello standard 1451 e risulta svincolato dal conoscere come NCAP e TIM comunicano fisicamente tra loro (ZigBee). Il meccanismo per ottenere il dato di temperatura è il seguente, ed è stato implementato secondo le specifiche dello standard IEEE 1451:

- L' utente tramite terminale video richiede all' NCAP un' operazione di lettura
- NCAP fornisce la lista di TIM collegati alla rete e l' utente ne seleziona uno e il Transducer Channel desiderato (in questo caso è unico ed è il canale temperatura)

- NCAP determina il Transducer Channel Number, ossia l' identificativo univoco della comunicazione con TIM e canale selezionati.

- NCAP prepara il comando di lettura riempiendo gli attributi del Command Message NCAP passa a ZigBee il pacchetto da inviare e ZigBee lo trasmette al TIM senza essere a conoscenza di ciò che sta trasmettendo (funzione di trasporto fisico)

- TIM riceve il pacchetto, lo scompatta e rileva che si tratta di una richiesta di lettura correttamente formattata

- TIM accede al sensore di temperatura, preleva il dato e lo impacchetta secondo la struttura Reply Message.

- TIM passa la sequenza di byte a ZigBee il quale li trasmette fisicamente all' NCAP.

- NCAP riceve il Reply Message, ne verifica la validità ed estrae il dato per inviarlo all'utente.

Tutte queste operazioni sono descritte da funzioni implementate in interfacce definite nel modulo che descrive lo standard, ossia l' `IEEE1451Dot0`.

1.7 Payload

Il payload è una struttura dati definita nello standard IEEE 1451 ed è utilizzata come "contenitore". Qualsiasi sequenza di ottetti che deve essere inviata fisicamente da un dispositivo ad un altro viene inserita in questo tipo di dato. Tale struttura è semplicemente composta da un array di bytes che contiene gli ottetti da inviare, e una variabile intera che memorizza il numero di ottetti presenti nell' array. La funzione del payload è svincolare l'applicazione dal tipo di dato (cioè dal suo formato) da inviare o ricevere.

1.8 Il modulo `IEEE1451Dot0`

Tale modulo descrive il funzionamento dello standard per mezzo di due moduli API (Application Programming Interface) ossia due set di interfacce: Transducer Services API, e Module Communications API.

1.8.1 Transducer Services API

Sono state implementate due interfacce di questa API:

Interfaccia	**Descrizione**
TimDiscovery	Metodi per applicazioni per la ricerca dei moduli di comunicazione IEEE 1451.X disponibili. I TIM e i Transducer Channels sono organizzati in questa interfaccia
TransducerAccess	Contiene i metodi per accedere al Transducer Channel di un sensore/attuatore

Tabella 1.6: Transducer Services API

TimDiscovery

Tale interfaccia è fornita dal layer 1451.0 ed è usata dall' applicazione per fornire un meccanismo di individuazione dei TIMs e dei Transducer Channels disponibili. I metodi di cui è composta l'interfaccia e che sono stati implementati sono:

TimDiscovery
UInt16 reportCommModule(out:: UInt8Array moduleIds);
UInt16 reportTims(in:: UInt8 moduleId, out:: UInt16Array timIds);
UInt16 reportChannels(in:: UInt16 timId, out:: UInt16Array channelIds, out:: StringArray names);

Tabella 1.7: TimDiscovery

- `reportCommModule`: questa funzione restituisce una lista dei moduli 1451.X registrati con l'NCAP.

  Parametri: (out) "moduleIds" è un array che contiene tutti i moduli 1451.X registrati con l'NCAP.

- `reportTims`: restituisce una lista di tutti gli identificativi dei TIMs presenti nella rete che utilizzano il modulo specificato.

  Parametri: (in) "moduleId" è l'identificativo del modulo 1451.X di cui vogliamo conoscere i TIMs disponibili (out) "timIds" è una lista di tutti i TIMs disponibili per quel determinato modulo di comunicaizoni.

- `reportChannels`: fornisce i Trasducer Channels diponibili per il TIM selezionato.

  Parametri: (in) "timId" è il TIM di cui vogliamo conoscere i canali disponibili (out) "channelIds" è la lista degli identificativi dei Transducer Channels presenti nel TIM (out) "names" non implementato

In questo progetto il modulo di comunicazione è il solo 1451.5 trattandosi di rete wireless. La connessione è Point to Point (P2P) quindi il TIM è uno solo (con identificativo ZigBee 0x796F) e monta un unico sensore di temperatura quindi anche il Transducer Channel è unico ed è TEMPERATURE=0xAAAA, come già detto.
Nel sistema implementato, ogni invocazione dei metodi descritti restituirà quindi una lista contenente un unico elemento essendoci un'unica scelta possibile. L'assegnazione dei vari identificativi è pertanto statica, ad esempio, al nodo ZigBee 0x796F viene assegnato l'identificativo '1'. L'implementazione adottata permette tuttavia di aggiungere e gestire più nodi o canali semplicemente usando un meccanismo dinamico di assegnazione e gestione degli identificativi in quanto la struttura del codice rimane valida per entrambi i casi.

TransducerAccess

Questa interfaccia fornisce i metodi per accedere al Transducer Channel e poter quindi leggere o scrivere dei dati nel trasduttore. Di questa interfaccia sono stati implementati i seguenti metodi:

- `open`: questa funzione apre un canale di comunicazione con il TIM/Transducer Channel desiderato e restituisce un identificativo unico per la comunicazione *Parametri*: (in) "timId" è il TIM selezionato (in) "channelId" è il Transducer Channel selezionato per il TIM specifico (out) "transCommId" è un identificativo calcolato usando timId e channelId. Verrà usato per le chiamate successive.

TransducerAccess
UInt16 open(in Args::UInt16 timId, in Args::UInt16 channelId, out Args::UInt16 transCommId);
UInt16 close(in Args::UInt16 transCommId);
UInt16 readData (in:: UInt16 transCommId, in::TimeDuration timeout, in:: UInt8 SamplingMode, out:: ArgumentArray result);

Tabella 1.8: TransducerAccess

- `close`: chiude il canale di comunicazione associato al transCommId. *Parametri*: (in) "transCommId" è l'identificativo della comunicazione che si desidera chiudere.

- `readData`: questa funzione prepara il Command Message caricando gli attributi con i rispettivi valori. *Parametri*: (in) " transCommId" è l' identificativo calcolato in precedenza (in) "timeout" è il tempo entro il quale deve essere portata a termine l'operazione di lettura (in) "SamplingMode" non implementato (out) "result" è il payload da inviare a ZigBee.

1.8.2 Module Communications API

Di questo modulo è stata implementata un'interfaccia, mentre le altre necessarie sono state ereditate dal precedente progetto e semplicemente modificate ed aggiustate per renderle compatibili con quelle scritte.

Interfaccia	**Descrizione**
P2PComm	Fornisce i metodi per gestire una comunicazione P2P
P2PRegistration	Fornisce i metodi per registrare un' istanza 1451.X al modulo 1451.0
P2PRecieve	Fornisce i metodi di notifica al layer 1451.0 dell'avvenuta ricezione di un messaggio

Tabella 1.9: Module Communications API

P2PComm

E' 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 IEEE 1451 al sistema fisico di trasmissione che è ZigBee. Questa è l'unica

interfaccia che è completamente dipendente dalla implementazione hardware del protocollo di comunicazione vero e proprio (ZigBee)

P2PComm
UInt16 read(in:: TimeDuration timeout, in:: UInt32 maxLen, out:: OctetArray payload, out:: Boolean last)
UInt16 write(in:: TimeDuration timeout, in:: OctetArray payload, in:: Boolean last)

Tabella 1.10: P2PComm

- `read`: trasferisce i dati ricevuti dallo strato ZigBee al modulo 1451.5 il quale li formatta secondo le specifiche IEEE 1451.

  Parametri: (in) "timeout" il massimo tempo entro il quale deve essere effettuata la lettura. Segue timeout error.

  (in) "maxLen" non implementato (payload unico).

  (out) "payload" il dato ricevuto formattato secondo le specifiche.

  (out) "last" non implementato (payload unico).

- `write`: trasferisce il payload allo strato ZigBee e lo trasmette al destinatario

  Parametri: (in) "timeout" il massimo tempo entro il quale deve essere effettuata la scrittura. Segue timeout error.

  (in) "payload" il dato da trasferire a ZigBee e inviare al destinatario.

  (in) "last" non implementato (payload unico)

Abbiamo ora gli strumenti per descrivere tramite le funzioni dello standard il protocollo di comunicazione NCAP-WTIM descritto in precedenza. Ipotizzando che l' utente voglia conoscere la tempratura del WTIM 0x796F, le funzioni IEEE1451 invocate dall'NCAP saranno nell' ordine:

- **reportCommModule(moduleIds)** l'utente seleziona dalla lista ottenuta come risposta il modulo 1451.5

- **reportTims(moduleId, timIds)** anche in questo caso la risposta è un elenco da cui l'utente seleziona il TIM numero 1

- **reportChannels(timIds, channelIds)** l'utente seleziona il canale TEMPERATURE 0XAAAA

- **open(timId, channelId, transCommId)** NCAP fornisce identificativo unico e imposta i parametri di connessione

- **readData(transCommId, result)** NCAP prepara il Command Message e lo incapsula nel payload (result)

- **write(timeout, payload)** NCAP passa la richiesta di temperatura a ZigBee che la invia al WTIM.

A questo punto il WTIM accede al sensore e invia all' NCAP il dato di temperatura formattato secondo la specifiche del Reply Message. L'NCAP tramite la funzione **read(Timeout, payload)** preleva il dato da ZigBee e lo trasferisce formattato ai moduli 1451.5 e successivamente 1451.0. L'utente può ora accedere al dato tramite applicazioni.

Capitolo 2

Introduzione a Zigbee

ZigBee è un protocollo per reti wireless progettato specificatamente per bassa velocità di trasferimenti di dati e per il controllo di dispositivi e sensori su reti wireless.
Le applicazioni del protocollo ZigBee sono varie: dalle reti di automazione, ai sistemi di sicurezza per la casa, ai sistemi di controllo industriale, misure remote e periferiche per personal computer. Comparato con altri protocolli wireless, il protocollo di comunicazione wireless ZigBee ha essenzialmente, tre vantaggi:

- bassa complessità
- richiesta ridotta di risorse
- un insieme di specifiche standard.

Si valuta che un nodo ZigBee del tipo più complesso richieda solamente il 10% del codice necessario per un tipico nodo Bluetooth o Wi-Fi, mentre il più semplice dovrebbe richiederne intorno al 2%. ZigBee si basa sulle specifiche IEEE 802.15.4 includendo la possibilità di creare reti, messaggi e ricerca di dispositivi su di esse. Il protocollo di rete wireless ZigBee può assumere diversi tipi di configurazione; in tutte le configurazioni di rete, ci sono almeno due dispositivi fondamentali:

- ZigBee Coordinator (ZC): è il dispositivo più intelligente tra quelli disponibili, costituisce la radice di una rete ZigBee e può operare da ponte tra più reti. E' di tipo FFD, Full Function Device, offre piena funzionalità, e solitamente è alimentato da rete e rimane

acceso quando inattivo. Ci può essere un solo Coordinator in ogni rete. Esso è inoltre in grado di memorizzare informazioni riguardanti la propria rete e può agire da deposito per le chiavi di sicurezza.

- ZigBee End Device (ZED): include solo le funzionalità minime per dialogare con un nodo Coordinator o Router, non può trasmettere dati provenienti da altri dispositivi; è il nodo che richiede il minor quantitativo di memoria e quindi risulta spesso più economico rispetto ai ZC.

A questi dispositivi talvolta se ne aggiunge un terzo,

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

2.1 Tipi di rete

2.1.1 Configurazione di rete a stella

La configurazione di rete a stella consiste di un dispositivo Coordinator e uno o più dispositivi EndDevice. In una rete del genere, tutti i dispositivi EndDevice presenti nella rete comunicano soltanto con il nodo Coordinator. Se un dispositivo EndDevice deve trasferire dati ad un altro dispositivo EndDevice, trasferisce i dati al Coordinator e poi quest'ultimo inoltra i dati al dispositivo EndDevice di destinazione.

2.1.2 Configurazione di rete ad albero

Un altro tipo di configurazione di rete è rappresentato dalla tipologia ad albero. In questa configurazione, i dispositivi EndDevice posso entrare nella rete sia attraverso il nodo Coordinator che attraverso i nodi Router. I nodi Router, in questo caso, svolgono due funzioni: la prima riguarda l'estensione del numero di nodi che possono entrare a far parte della rete; la seconda riguarda l'estensione fisica del range della rete. Con l'aggiunta di un Router, un dispositivo EndDevice, non necessita di essere nel range radio del nodo Coordinator per comunicare nella rete. Tutti i messaggi in una rete ad albero vengono smistati lungo l'albero di trasmissione.

2.1.3 Reti a maglia

La configurazione di rete a maglia è simile a quella ad albero, ad eccezione del fatto che in questa configurazione i dispositivi FFD possono smistare messaggi direttamente ad altri FFD senza seguire la struttura ad albero. I messaggi verso i dispositivi RFD passano attraverso i nodi padri (Router). Il vantaggio di questa topologia consiste nella riduzione della latenza legata alla trasmissione dei messaggi e ad un incremento di affidabilità.

In questo progetto, dovendo realizzare una connessione punto a punto, la configurazione di rete ZigBee utilizzata è quella a stella.

2.2 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, dopodiché, una volta connesso, verrà identificato nella rete attraverso il suo network address e tramite esso potrà comunicare con gli altri dispositivi. Per la messaggistica di tipo unicast viene utilizzato il MAC address specifico, mentre per il broadcast (impiegato nelle operazioni di gestione della rete quali creazione, connessione ecc.) si utilizza il MAC address generico 0xFFFF.

2.3 Sincronizzazione

Un beacon frame è 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 è che tra un beacon e l'altro i nodi possono entrare in risparmio energetico.
Questa tipologia di rete è preferibile nei casi in cui il risparmio di energia è importante, tipicamente quando sono previsti dei nodi alimentati da una batteria ma l'utilizzo di questa souzione ne complica la gestione, richiedendo meccanismi di timing precisi, più difficili e costosi da re-

alizzare.

In una rete non-beacon enabled tutti i nodi sono costantemente attivi e perciò molto più semplici da gestire ed economici da realizzare. Tuttavia ne consegue un superiore consumo energetico. Poiché tra gli scopi del progetto non rientra la modalità risparmio energetico, la tipologia di rete adottata è 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 può iniziare la trasmissione. Se il dispositivo di destinazione è di tipo FFD il suo ricetrasmettitore sarà sempre acceso e il messaggio verrà ricevuto immediatamente. Diversamente un RFD potrà avere il ricetrasmettitore spento (in modalità risparmio energetico) al momento della ricezione, per cui, una volta acceso il ricetrasmettitore, dovrà appoggiarsi al suo FFD associato (genitore)il quale avrà temporaneamente memorizzato in un buffer i messaggi non ricevuti richiedendone l'inoltro. Tale caratteristica permette all'RFD di risparmiare energia ma, al contempo, richiede all'FFD una sufficiente quantità 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.

2.4 Profili

Un profilo ZigBee è 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 è associato un identificatore univoco. Gli attributi sono raggruppati in cluster ai quali, a loro volta, è 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 più cluster viene definito endpoint, pertanto dispositivi diversi possono comunicare tra loro facendo ricorso agli stessi endpoint (se implementati).

2.5 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 *Message* (MSG). Entrambi i formati sono associati ad un Cluster ID, ma il KVP rispetta una struttura definita, a differenza dell'MSG che non è vincolato da alcuna struttura. Il profilo in uso a livello applicazione specifica il formato dei messaggi supportato, ma non è permesso ai cluster di utilizzare entrambi i formati.
La gestione della sicurezza è 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' è 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 può 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 è omessa l'implementazione di questa sezione riguardante la sicurezza.

2.6 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 dispendio 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 è utile nelle reti con più end device.
Nello sviluppo di questo progetto il binding è ininfluente e, oltretutto, richiederebbe una gestione efficiente dell'allocazione dinamica di memoria, pertanto non verrà implementato.

2.7 Formazione della rete

La rete ZigBee può 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 è definito dal parametro ScanDuration. Per la banda 2.4 GHz, il tempo di scansione (in secondi) è determinato dalla seguente equazione:

$$T\ Scansione = 0.01536 * (2 * ScanDuration + 1)$$

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, che però non è supportata dallo stack Microchip utilizzato in questo progetto.
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 può 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; altri- menti, se tale procedura fallisce oppure il nodo orfano non ha nessun genitore memorizzato nella propria tabella dei vicini, tenterà 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 può 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 è 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 è compreso tra 10 e 75 metri, dipendentemente dall'ambiente circostante. La massima potenza trasmessa è in genere 0 dBm (1 mW).

2.8.2 MAC

La modalità base di accesso al canale, specificata da IEEE 802.15.4-2003, è il Carrier Sense Multiple Access/Collision Avoidance (CSMA/CA). Questo significa che i nodi controllano se il canale è libero quando devono 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 necessità 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 è di 127 byte, compreso un campo CRC a 16 bit per il controllo dell'integrità del frame;inoltre lo standard IEEE 802.15.4 prevede l'uso (opzionale) di un meccanismo di acknowledge tramite l'impostazione di un bit di flag di ACK all'interno dei frame inviati. Un messaggio ZigBee può essere quindi formato al più da 127 byte, cossì 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

3.1 Caratteristiche

Lo stack ZigBee fornito da Microchip, che verrà utilizzato per lo sviluppo di questo progetto, è 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 pro- tocollo ZigBee e nelle specifiche IEEE 802.15.4.
- portabilità su tutte le famiglie di microcontrollori PIC18 , PIC24.
- supporto per l'indirizzamento multi casting
- supporto per meccanismi di rejoin degli end device

Esistono però 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

3.2 Architettura Stack

Il Microchip ZigBee Stack è stato progettato rispettando il protocollo ZigBee e le specifiche IEEE 802.15.4, mantenendo cioè la nomenclatura il più coerente possibile ed organizzando i vari layer in file sorgenti separati ottenendo una libreria modulare indipendente dall'applicazione.
Detto stack è scritto in linguaggio ANSI-C ed è orientato ai microcontrollori Microchip della serie PIC18 e PIC24 montati a bordo del sistema di sviluppo PICDEM Z di Microchip, pertanto sfrutta la memoria flash interna del processore per il salvataggio dei parametri quali indirizzo MAC, tabella dei vicini e tabella di binding. Tuttavia è progettato in modo da essere facilmente portabile in altri sistemi basati su microcontrollore PIC e in modo da supportare diversi transceiver con minimi cambiamenti ai livelli più alti dell'applicazione senza intervenire nei layer più bassi.
Infine il Microchip ZigBee Stack è ideato per funzionare con i compilatori Microchip MPLAB Compiler, ma con esigue modifiche può supportare anche altri compilatori di linguaggio ANSI C.
Come mostrato in Figura 3.1, 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. Sostanzialmente lo stack ZigBee messo a punto da Microchip è ridotto a quattro strati, rispetto ai tradizionali sette previsti dal modello ISO/OSI. Con un approccio di tipo bottom-up si incontrano i seguenti 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 è perciò molto complesso e costituisce il cuore dello stack poichè, 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 agli endpoint appropriati, siano essi endpoint gestiti dal framework dell'applicazione (AFG) oppure endpoint previsti dal protocollo ZigBee (ZDO).

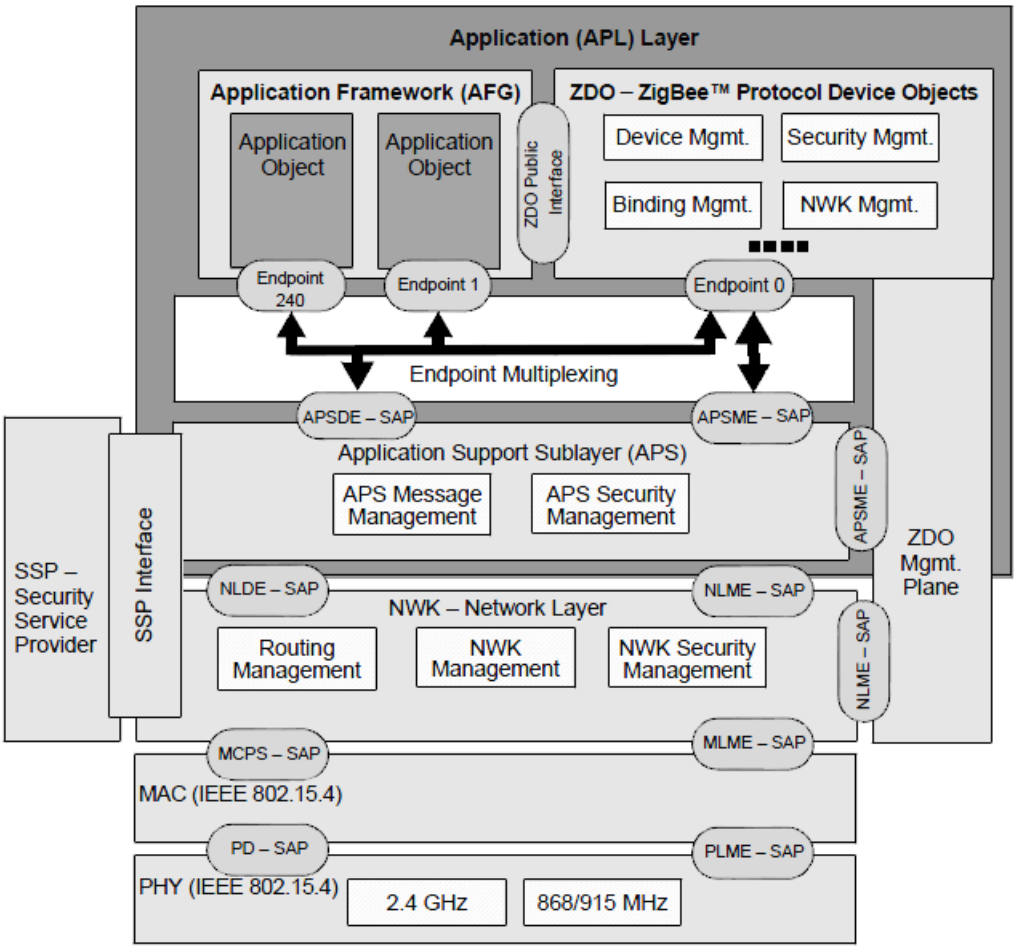


Figura 3.1: Architettura Microchip ZigBee Stack

3.3 Funzionamento Microchip Stack ZigBee

3.3.1 Macchina a stati ZigBee

Lo stack Microchip è 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.

3.3.2 Primitive ZigBee

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 è 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 è estremamente complessa ma allo stesso tempo facile da pilotare in quanto ZigBee risolve automaticamente le transizioni a tutti gli stati; è pertanto sufficiente impostare i parametri della primitiva del layer APS e passarla allo stack. ZigBee la processerà al successivo ciclo macchina e realizzerà le transizioni e le relative azioni associate autonomamente, in funzione della particolare richiesta inoltrata con la primitiva (es. invia dato).
In Figura 3.2 sono elencate le principali primitive ZigBee.

3.3.3 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;

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

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 può 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 può 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 chiamata alla routine `ZigBeeTasks()` innesca la procedura di funzionamento dello stack per la primitiva in questione. Tale funzione infatti è 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 può essere lasciato in attesa tramite l'assegnazione della primitiva speciale *NO PRIMITIVE* .
Poichè può 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 da' inoltre la possibilità di visualizzare via seriale l'output dell'applicazione attraverso una console di tipo terminale, per fare ciò 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 parità né di flusso (19200,N,8,1).

3.3.4 Funzioni utilizzate

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 modalità sleep mode.

- Sintassi: BOOL APLDisable(void);
- Input: nessuno.
- Output: TRUE se il ricetrasmettitore è stato disabilitato, FALSE se è 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 processamento 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 in corso di processamento da parte dei livelli sottostanti. Se chiamata dopo che tutto il messaggio è 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 in corso di processamento.

`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 è pronto all'invio di un messaggio.

- Sintassi: BOOL ZigBeeReady(void);

- Input: nessuno.

- Output: TRUE se è possibile caricare un nuovo messaggio nel buffer d'uscita, FALSE se il buffer è 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 continuerà 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.

3.4 Creazione e connessione ad una rete

Il nodo coordinatore, per creare una rete, esegue la primitiva *NLME_NETWORK_FORMATION _ request*. Se il dispositivo non è un coordinatore e non è 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 ciò deve innanzitutto scoprire le reti disponibili attivando la primitiva *NLME_NETWORK_DISCOVERY_request*. Dopodiché l'applicazione sceglie la rete a cui collegarsi e può farlo tramite la primitiva *NLME_JOIN_request*, questa volta con il parametro `RejoinNetwork` settato a `FALSE`.

3.5 Invio di un messaggio

Lo stack ZigBee implementato da Microchip permette di inviare un solo messaggio per volta. Perchè ciò sia possibile è necessario innanzitutto verificare che la funzione `ZigBeeReady()` restituisca il valore TRUE, ad indicare che lo stack è pronto. La funzione `ZigBeeBlockTx()` blocca le trasmissioni (di conseguenza ulteriori chiamate a `ZigBeeReady()` restituiscono il valore FALSE) ed è quindi possibile caricare il payload del messaggio da inviare nel vettore `TxBuffer` (indicizzato dalla variabile `TxData`), nonchè 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, è 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`. Poichè ogni frame è identificato da un Transaction ID univoco, esso può essere reperito chiamando la funzione `APLGetTransID()`. Lo stato di invio del messaggio è 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`.

3.6 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 `DstEndpoint`, come dice il nome stesso, indica l'endpoint di destinazione. Se il messaggio è pervenuto all'endpoint corretto può 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 mancata osservanza di questa regola porta al blocco della ricezione dei messaggi dopo il primo messaggio processato ma non

scartato.

3.7 Analisi dell'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 è 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 è capito come modificarli per creare due driver ottimizzati, uno per NCAP e uno per WTIM.
Dal punto di vista funzionale e relativo a ZigBee, i due dispositivi sono identici pertanto identica è anche la struttura dei firmware e la gestione dello stack.
Dopo aver ottimizzato i driver a livello ZigBee, si è passati all'implementazione dello standard IEEE1451 e alla sua inclusione negli applicativi.

Il cuore del firmware è il seguente ciclo, descritto all'interno della funzione main() nei files `Coordinator.c` e `RFD.c`:

Listing 3.1: `Coordinator.c`, Ciclo macchina

```
while(1)
    {
        /* Clear the watch dog timer */
        CLRWDT();

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

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

        /* do any non ZigBee related tasks  */
        ProcessNONZigBeeTasks();
```

```
    }
```

3.7.1 Watchdog timer

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

3.7.2 ZigBeeTasks

`ZigBeeTasks(&currentPrimitive);` è 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 è uguale a *NO_PRIMITIVE*, ossia la macchina è in stato idle, `ZigBeeTasks()` interroga i layer più bassi con funzioni specifiche per di ogni livello, partendo dal più 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 verrà risolta successivamente. Se il risultato finale di queste verifiche è ancora *NO_PRIMITIVE* allora il processo di verifica è finito e la funzione `ZigBeetask()` termina; in caso contrario, tramite una struttura *switch-case* viene gestita la primitiva specifica. Questo meccanismo viene iterato finchè non sono risolti tuti i processi pendenti.

Listing 3.2: `ZigBeeTasks.c`, Controllo tasks layer inferiori

```
if(ZigBeeStatus.nextZigBeeState == NO_PRIMITIVE)
        {
            ZigBeeStatus.nextZigBeeState
                = PHYTasks(ZigBeeStatus.nextZigBeeState);
        }
        if(ZigBeeStatus.nextZigBeeState == NO_PRIMITIVE)
        {
            ZigBeeStatus.nextZigBeeState
                = MACTasks(ZigBeeStatus.nextZigBeeState);
```

```
        }
        if(ZigBeeStatus.nextZigBeeState == NO_PRIMITIVE)
        {
            ZigBeeStatus.nextZigBeeState
                = NWKTasks(ZigBeeStatus.nextZigBeeState);
        }
        if(ZigBeeStatus.nextZigBeeState == NO_PRIMITIVE)
        {
            ZigBeeStatus.nextZigBeeState
                = APSTasks(ZigBeeStatus.nextZigBeeState);
        }
        if(ZigBeeStatus.nextZigBeeState == NO_PRIMITIVE)
        {
            ZigBeeStatus.nextZigBeeState
                = ZDOTasks(ZigBeeStatus.nextZigBeeState);
        }
```

Se il risultato è diverso da *NO_PRIMITIVE* allora viene risolto il processo corrente:

Listing 3.3: `ZigBeeTasks.c`, Switch-Case

```
switch(ZigBeeStatus.nextZigBeeState)
        {
            // Check for the primitives that are handled by the PHY.
            case PD_DATA_request:
            case PLME_CCA_request:
            case PLME_ED_request:
            case PLME_SET_request:
            case PLME_GET_request:
            case PLME_SET_TRX_STATE_request:
                ZigBeeStatus.nextZigBeeState
                        = PHYTasks(ZigBeeStatus.nextZigBeeState);
                break;
```

```
            // Check for the primitives that are handled by the MAC.
            case PD_DATA_indication:
            case PD_DATA_confirm:
            case PLME_ED_confirm:
            case PLME_GET_confirm:
            case PLME_CCA_confirm:
            case PLME_SET_TRX_STATE_confirm:
            case PLME_SET_confirm:
            case MCPS_DATA_request:
            case MCPS_PURGE_request:
            case MLME_ASSOCIATE_request:
            case MLME_ASSOCIATE_response:
            case MLME_DISASSOCIATE_request:
            case MLME_GET_request:
            .
            .
            .
        }
```

Questo controllo continua finché non sono risolti tutti i processi in background:

Listing 3.4: `ZigBeeTasks.c`, Ciclo Do-While

```
do
    {
    if(ZigBeeStatus.nextZigBeeState == NO\_PRIMITIVE)
        {
            ZigBeeStatus.nextZigBeeState
                = PHYTasks(ZigBeeStatus.nextZigBeeState);
                .
                .
                .
```

```
        }
    switch(ZigBeeStatus.nextZigBeeState)
        {
        .
        .
        .
        }
    }while(ZigBeeStatus.nextZigBeeState != NO\_PRIMITIVE);
        //Finche' ci sono background tasks risolvile.
```

3.7.3 ProcessZigBeePrimitives

Se il sistema non è in idle, ossia `currentPrimitive` è 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 dovrà essere processata alla prossima iterazione.

Listing 3.5: `Coordinator.c`, ProcessZigbeePrimitives

```
void ProcessZigBeePrimitives(void)
{
        switch (currentPrimitive)
        {
                        case NLME_DIRECT_JOIN_confirm:
                        .
                        .
                        .
                        case NLME_JOIN_indication:
                        .
                        .
                        .
```

```
                        case APSDE_DATA_indication:
                        .
                        .
                        .
                }
                .
                .
                .
}
```

Un altro scopo importante di questa funzione è la gestione della console video che viene effettuata al termine del processamento della primitiva chiamando la funzione specifica `ProcessMenu()`. Tale metodo è molto importante in quanto è tramite la console che è possibile 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 è una soluzione di sviluppo temporanea in quanto l'uso dello standard IEEE1451.0 permetterà in futuro 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.
La funzione `ProcessMenu()` verrà analizzata successivamente.

3.7.4 ProcessNONZigBeeTasks

Nello stack dimostrativo Michrochip questa funzione è 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 è 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. E' stato deciso pertanto di mantenere questa linea di sviluppo e lo standard è stato descritto su listati separati. La chiamata ai metodi IEEE1451 avviene quindi al-

l'interno della funzione `ProcessMenu()` e non su `ProcessNONZigBeeTasks()`. Come si vedrà in seguito, questo permette di interfacciare in modo molto semplice lo standard IEEE1451 con ZigBee

Capitolo 4

Materiale utilizzato

4.1 Microchip PICDEM Z Demonstration Kit

Il kit si compone di una coppia di dispositivi uguali(Figura 4.1) costituiti da una scheda madre che ospita:

- un microcontrollore PIC serie 18 (PIC 18LF4620) su zoccolo da 40 pin.
- un sensore di temperatura Microchip TC77 a 5 pin connesso al microcontrollore tramite Serial Peripheral Interface (SPI).
- due led collegati alle uscite digitali del microcontrollore;
- due pulsanti collegati agli ingressi digitali del microcontrollore;
- un pulsante di reset;
- un connettore RJ-11 per la programmazione del firmware;
- un connettore RS-232 per la programmazione del firmware e per l'output seriale;
- un connettore SPI per l'alloggiamento della scheda wireless;
- un connettore per l'alimentazione da batteria;
- un connettore Jack femmina per l'alimentazione esterna.

Il PIC 18 in questione è costituito da :

- microprocessore 8 bit a 40 MHz;
- 64 kByte di memoria flash per la memorizzazione del firmware e per l'esecuzione dei programmi;
- 1024 kByte di memoria EEPROM;
- 36 linee di I/O che possono essere utilizzate per la simulazione e la lettura di dati dall'esterno (sia in analogico che in digitale, grazie al convertitore A/D a 10 bit integrato).

I pulsanti,ed i led montati sulla scheda rivelano la loro utilità in fase di debug, facilitando la simulazione e la visualizzazione grafica. Il connettore SPI permette di utilizzare schede wireless diverse, a seconda dell'uso che se ne vuole fare, mentre è possibile alimentare il tutto tramite una batteria da 9 V in modo da rendere il dispositivo indipendente e trasportabile. Con il kit in questione, a causa della scarsità di memoria e delle basse prestazioni del microporcessore, verrà realizzato il WTIM, che per definizione necessita di meno risorse durante il funzionamento.

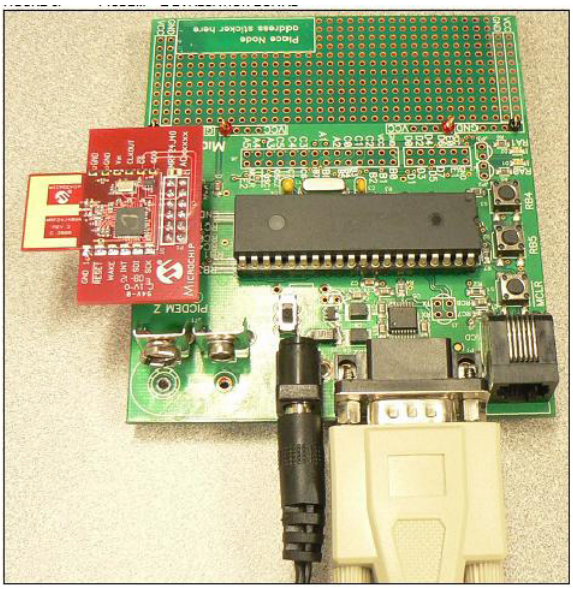


Figura 4.1: PICDEM Z

4.2 EXPLORER 16 development board

La scheda (Figura 4.2) è composta dai seguenti elementi hardware principali:

- un microcontrollore PIC serie 24 (PIC 24FJ128GA010) su zoccolo da 100 pin.
- 4 pulsanti connessi alle porte I/O del microcontrollore;
- 8 LED connessi alle porte di I/O del microcontrollore;
- memoria EEPROM da 256 kByte con interfaccia seriale ;
- sensore di temperatura;
- display LCD , 2 righe da 16 caratteri;
- potenziometro analogico;
- un connettore RJ-11 per la programmazione del firmware;
- un connettore RS-232 per la programmazione del firmware per l'output seriale;
- un connettore Jack femmina per l'alimentazione esterna.

Il PIC 24 in questione è costituito da :

- 128 kByte di memoria ash per la memorizzazione del firmware;
- 8 kByte di memoria RAM sincrona;
- 5 timer a 16 bit;
- 5 interfacce I/O di tipo capture / compare , con uscita PWM;
- 2 interfacce UART;
- 2 serial peripheral interface (SPI);
- 2 interfacce I2C;
- convertitore A/D a 10 bit, 16 canali;

- 2 comparatori; Il segnale di clock primario per il microcontrollore è fornito da un oscillatore a 8 MHZ. Con il questo kit, considerata l'ampia disponibilità di memoria, è stato realizzato l'NCAP.

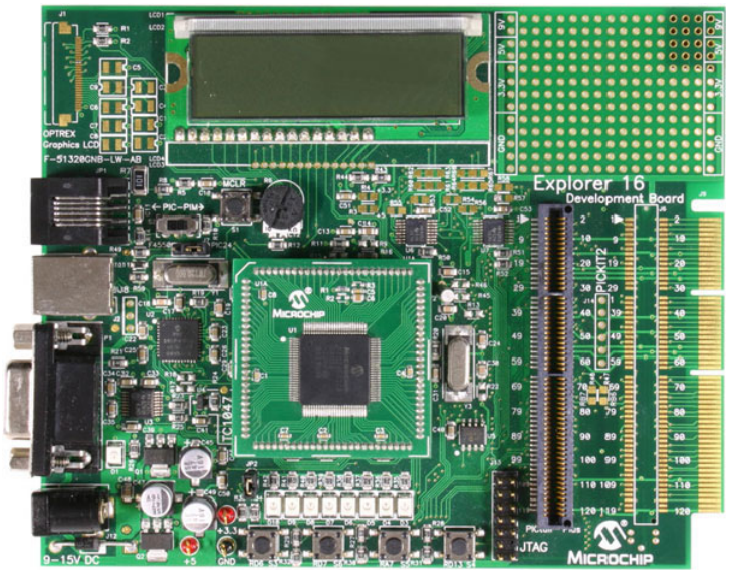


Figura 4.2: EXPLORER 16

4.3 Microchip MRF24J40

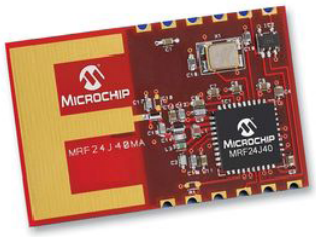


Figura 4.3: MRF24J40

Le schede wireless (Figura 4.3) con interfaccia SPI fornite di serie dal costruttore (montate sia sul PICDEM Z che su EXPLORER 16) sono adatte alla banda dei 2.4 Ghz e montano il trasmettitore/ricevitore (transceiver) Microchip MRF24J40 compatibile con lo standard IEEE 802.15.4. L'antenna è tracciata sul circuito stampato ma c'è la possibilità di collegare

un' antenna esterna tramite connettore SMA (da installare). Esistono altri transceiver compatibili, nati per essere utilizzati in bande di frequenza diverse e con standard diversi,ma in questa sperimentazione verrà utilizzato soltanto il transceiver prima citato.

4.4 Digitus USB/ Serial Adapter

Figura 4.4: Digitus USB/ Serial Adapter

Questo dispositivo è un convertitore USB maschio / RS-232 maschio per PC sprovvisti di porta seriale(Figura 4.4)

4.5 Microchip MPLAB ICD 2

Questo dispositivo (Figura 4.5) è un programmatore per microprocessori PIC. Può essere collegato al target tramite cavo a 6 poli con connettore RJ-11 e al PC sia via USB che via seriale RS-232; se collegato via USB non necessita di alimentazione esterna. Dispone di LED diagnostici (Power, Busy, Error). Oltre che per programmare il target, pu essere utilizzato per eseguire anche il real time background debugging in collaborazione con l'ambiente di sviluppo MPLAB IDE.

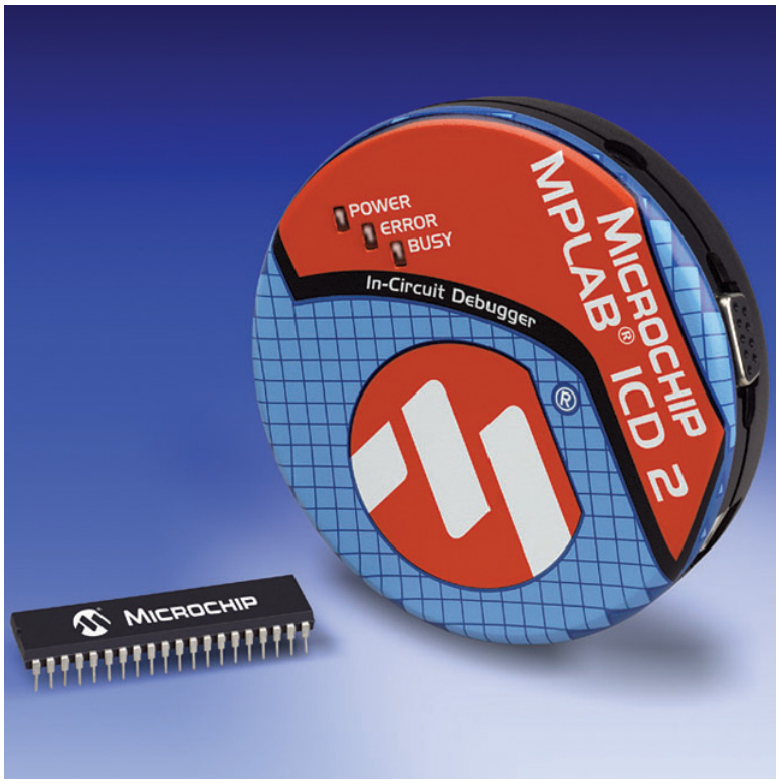


Figura 4.5: Microchip MPLAB ICD 2

4.6 Microchip MPLAB IDE

L'MPLAB Integrated Development Environment (IDE) è l'ambiente di sviluppo a 32 bit fornito da Microchip per la creazione di applicazioni basate su microcontrollori PIC. Si compone di un interfaccia grafica, di un editor di testo e di vari componenti add-on, come ad esempio i compilatori, i simulatori e i driver per i programmatori/debugger.

4.7 Compilatori

L'MPLAB C-18 Compiler è un add-on per l'ambiente di sviluppo MPLAB sopra descritto e consente la compilazione di programmi scritti in linguaggio ANSI C per microprocessori a 8 bit della serie Microchip PIC18. MPLAB C-30 Compiler è un add-on per l'ambiente di sviluppo MPLAB sopra descritto e consente la compilazione di programmi scritti in linguaggio ANSI C per microprocessori della serie Microchip PIC24.

4.8 Microchip Zigbee Stack

Si tratta di uno stack protocollare in linguaggio C e un assembler utilizzato per reti wireless a basso bit-rate che comprende le funzioni di creazione e di gestione della rete, messaggistica e ricerca di dispositivi secondo lo standard Zigbee. Si basa sui livelli PHY (Physical Layer) e MAC (Medium Access Control) definiti dallo standard IEEE 802.15.4.

4.9 Microchip ZENA

Ottimo strumento per l'analisi dello scambio dei pacchetti, Zena analyzer si è rivelato uno strumento davvero utile nella risoluzione dei problemi relativi alla gestione dei cluster Zena Analyzer lavora tramite un'apposito circuito stampato collegato via USB al pc su cui sono montati un piccolo microcontrollore ed un'antenna. Il programma consente di visualizzare su schermo con un grafico molto intuitivo tutti i pacchetti scambiati attraverso la rete wireless tra i vari device ad essa connessi. Lo sniffer di rete Zena offre importantissime informazioni sulla comunicazione di rete, non è solamente uno strumento di controllo finale ma è anche un debugger on-the- fly di ciò che che si sta implementando anche in forma non definitiva.

4.10 Trasduttore di temperatura

Il codice inserito originalmente nelle schede di valutazione Microchip comprende un dimostrativo in cui sono implementati un Coordinator e un RFD progettati per una rete PAN (Personal Area Network). Il primo passo è stato attivare all'interno dell'RFD il sensore di temperatura TC77 in modo da poter poi richiamare la misura da Coordinator. Questo sensore è accessibile per via seriale ed solitamente montato su supporti a basso costo e a dimensioni ridotte. La comunicazione con il TC77 si stabilisce attraverso interfacce compatibili con SPI e Microwire. Esso offre una accuratezza di 1.$^\circ$(max.) con un range di temperature che vanno da +$^\circ$ a +$^\circ$ e può essere configurato in modo da operare in modalità basso consumo, con assorbimento ridotto a 0.1 μA, ma non è stata implementata poichè l'alimentazione dell' RFD è sempre accesa. L'interfacciamento col sensore è eseguito dalla funzione BYTE GetTC77(char *buffer). Questo

metodo fornisce come output la lunghezza della stringa in formato BYTE, legge le informazioni dal sensore TC77 e ritorna un valore di temperatura tramite il puntatore *buffer. Se la funzione restituisce valori come 0x00 o 0xFF, significa che si è verificato un errore nell'acquisizione del dato. Il sensore TC77 fornisce in un'uscita a 13 bit ottenendo una risoluzione di 0.0625 °. Un valore pari a zero corrisponde a 0 gradi Celsius, 1 corrisponde a 0.0625 °, 2 = 0.125 ° etc. Per ottenere un valore leggibile, il dato viene convertito in un numero a virgola mobile e poi moltiplicato per 0.0625 .

Capitolo 5

Implementazione standard IEEE1451 e convergence layer

5.1 Transducer Services API

5.1.1 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.

Listing 5.1: IEEE1451dot0TimDiscovery.c

```
#include "IEEE1451dot0TimDiscovery.h"
#include "Coordinator.h"



extern UInt16Array timIdDot5[10];
extern UInt8Array moduli1451[6];
extern UInt16Array channelList[][10][10];


UInt16  reportCommModule( UInt8Array *moduleIds){


  moduleIds=moduli1451;
```

```
  return 0x0; // NO ERROR
}

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

  if(*moduleId==5)
    timIds=timIdDot5;

  return 0x0; // NO ERROR
}

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

        channelIds = &channelList[DOT_FIVE][*timId][0];

        return 0x0; // NO ERROR

}
```

Le strutture `timIdDot5[10]`, `moduli1451[6]` 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[][10][10]` è una matrice tridimensionale che 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 è unica in quanto l'unico modulo previsto epr questo progetto è 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 dovrà essere considerato in futuro quando si dovrà 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 ciò che fanno le sopracitate funzioni è semplicemente passare ad una variabile l'indirizzo di liste inserite precedentemente dal programmatore nel firmware.

5.1.2 IEEE1451dot0TransducerAccess.c

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

Listing 5.2: ,IEEE1451dot0TransducerAccess.c

```
#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;
  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.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){
  return 0x0;
}

UInt16 readData(UInt16 *transCommId, TimeDuration timeout
                                    , 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
   result->val[6]=0;               //dataOffset
   result->len=7;
   return 0x0;
}
```

- `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 nella mia realizzazione tale identificativo è 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 è stata dichiarata ma mai utilizzata.

- `readData` è la funzione che realizza la formattazione del payload secondo la struttura del Command Message.

5.2 Module Communications API

5.2.1 P2PComm

E' 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 è ZigBee.

Listing 5.3: ,`IEEE1451dot0TransducerAccess.c`

```
#include "ZigBeeTasks.h"
#include "zAPL.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, OctetArray *payload){
 UInt8 lungDatoGetString;
 int i;
 char valore;
 char a = 0;
 char b = 0;
 char c = 0;
```

```
payload->len = 0;
lungDatoGetString=APLGet();
for(i = 0; i < lungDatoGetString + 3; i++){
   valore = APLGet();
       if(valore == 0x2e){ /*valore corrispondente
                                  al punto decimale*/
         ConsolePutROMString( (ROM char *)"." );
         i=i+1;
         payload->val[payload->len]=(BYTE)".";
         payload->len = (payload->len+1);
    }
    else if(valore == 0x20){ //valore corrispondente allo spazio
     ConsolePutROMString( (ROM char *)"␣" );
     i=i+1;
     payload->val[payload->len]=(BYTE)"␣";
     payload->len=payload->len+1;
    }
    else if(valore == 0x43){ /*valore corrispondente
                                     al carattere maiuscolo C*/
        ConsolePutROMString( (ROM char *)"C" );
        i=i+1;
        payload->val[payload->len]=(BYTE)"C";
        payload->len=payload->len+1;
   }
   else{
     if((i+1)\%2){ //se i pari entra
     a = ((valore - 0x30)<< 4);
   }
   else{
     b = (valore - 0x30);
     c = a + b ;
```

```
        PrintChar(c);
        payload->val[payload->len]=c;
        payload->len=payload->len+1;
      }
    }
  return 0;
}

 UInt16 write(TimeDuration  timeout, OctetArray *payload
     ,Boolean last){

 ZigBeeBlockTx();

 int i;
 TxBuffer[TxData++] = payload->len; // Lunghezza

 for(i=0 ; i < payload->len; i++){
   TxBuffer[TxData++]=payload->val[i]; //trasmette il payload
 }

 params.APSDE_DATA_request.TxOptions.Val = 0;
 params.APSDE_DATA_request.TxOptions.bits.acknowledged = 1;
 params.APSDE_DATA_request.ClusterId.Val
     = NCAP_OUTPUT_CLUSTER;

 //Imposto la primitiva
 currentPrimitive = APSDE_DATA_request;

 //Ciclo macchina ricorsivo
 while(timeout.nsec>0 && currentPrimitive!=NO_PRIMITIVE)
 {
```

```
    CLRWDT();

    //Aggiorna macchina a stati IEEE1451
    IEEEdot5ProcessNextState();

    ConsolePutROMString( (ROM char *)currentPrimitive );

    ZigBeeTasks(&currentPrimitive);

    ProcessZigBeePrimitives();
    timeout.nsec--;
  }

  //Aggiorna macchina a stati IEEE1451
  IEEEdot5ProcessNextState();

  if(!IEEEPrimitive)
    return ERROR_CODE=5; //CORRUPT COMUNICATION NETWORK FALIURE CODE
  else if(timeout.nsec==0)
    return ERROR_CODE=3; //OPERATION TIMEOUT CODE
  else{
     last= 1;
  }
  return ERROR_CODE = 0;
}
```

- `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 `currenPrimitive` con la primitiva APSDE_DATA_request, la cui funzione è preparare e avviare una co-

municazione radio. Alla prossima iterazione del ciclo macchina, ZigBee processerà tale primitiva e trasmetterà il contenuto del buffer di trasmissione al WTIM. Il WTIM risponderà con un Reply Message contenente il dato di temperatura letto dal sensore, e la funzione `read()` successivamente trasmetterà il dato al modulo 1451.5. La caratteristica di tale funzione è che l'elaborazione della primitiva APSDE_DATA_request non avviene nel ciclo macchina principale, bensì all'interno della funzione stessa. Ciò è 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 realizzate su un'unità 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.

Capitolo 6

Clusters e ProcessMenu()

6.1 Clusters

I cluster sono dei messaggi a livello applicazione che possono contenere uno o più attributi; ogni messaggio ZigBee viene identificato con un cluster. Ogni messaggio è costruito su misura del cluster e quest'ultimo ne definisce gli attributi nonchè il metodo di estrazione ed elaborazione. Ogni cluster ha un identificativo che viene spedito assieme al messaggio. Il dispositivo ricevente estrae l'identificativo del cluster e in base al valore di quest'ultimo elabora i bytes contenuti nel campo dati del messaggio.

I cluster utilizzati sono definiti nell'endpoint di default e sono diversi nell'NCAP rispetto al WTIM, adattandosi alle esigenze specifiche delle funzioni. Requisito fondamentale è programmare i cluster in modo da collegare quello dell'uscita di un dispositivo con il l'input cluster dell'altro e viceversa. Questo si ottiene facendo coincidere il `clusterID` del cluster NCAPout con il `clusterID` del cluster WTIMin, e viceversa.

Lo standard 1451.5 definisce un profilo specifico, definito Bulk Transfer, che prevede due dispositivi, denominati NCAP e WTIM. A ciascuno di essi sono associati due cluster, rispettivamente: NCAPout, NCAP in e WTIMin, WTIMout. Per la realizzazione di questo progetto però non è stato necessario utilizzare tutti e quattro i cluster. I cluster di uscita infatti servono per gestire richieste di lettura/scrittura inoltrate dal WTIM verso l'NCAP. Dovendosi limitare solo all'invio di un dato di temperatura il WTIM non ha quindi necessità di inoltrare richieste all'NCAP pertanto tali cluster sono sostanzialmente inutili in quanto ad essi non è associata nessuna

azione.

A WTIMin e NCAPout si è attribuito l'identificativo 0x0006 mentre a WTIMout e NCAPin si è attribuito l'identificativo 0x0005.

Quando NCAP prepara il messaggio inserisce il `clusterID` NCAPout (cioè 0x0006) che equivale all'ID di WTIMin; alla ricezione del messaggio, il TIM eseguirà le azioni previste da WTIMin e imposterà per la risposta l'ID 0x0005 associato a WTIMout. Ma WTIMout corrisponde a NCAPin e quando NCAP riceverà il messaggio di risposta lo analizzerà in base a quanto definito su NCAPin. Pertanto i cluster di uscita non vengono mai utilizzati.

Nel meccanismo di comunicazione utilizzato in questo progetto è stato quindi necessario sviluppare solo i cluster WTIMin e NCAPin; nel primo è indicato come scompattare, decodificare ed eseguire il Command Message ricevuto dall'NCAP, mentre nel secondo è descritto come estrarre e formattare il dato di temperatura ricevuto. Quando il dispositivo riceve un messaggio viene lanciata un'interruzione e viene settata la primitiva *APSDE_DATA_indication* nella variabile `currentPrimitive`. Tale primitiva è l'handler dei cluster: al successivo ciclo macchina infatti la funzione `ProcessZigBeePrimitives` entrerà in *case:APSDE_DATA_indication*, leggerà l'id del cluster ed eseguirà le azioni richieste.

6.1.1 NCAP_INPUT_CLUSTER

Questo cluster scompatta il Reply Message arrivato dal WTIM e chiama la funzione `read()` dello standard IEEE1451, il cui compito è quello di recuperare il dato di temperatura e passarlo al modulo IEEE1451.5

Listing 6.1: ,`Coordinator.c` NCAPin

```
void ProcessZigBeePrimitives(void)
{
 switch (currentPrimitive)
    {
    .
    .
    .
```

```
case NCAP_INPUT_CLUSTER:
{
  UInt8 success;
  UInt16 len;
  UInt8 dataOffset;

  success=APLGet();
  if(success==SUCCESS)
  {
    len=APLGet();
    len=(len<<8);
    len=len|APLGet();
    if(len==0x000B)
    {
      dataOffset=APLGet();

      //stampa a console lo short address del mit. del messaggio
      printf("\r\n");
      printf("From␣Address:␣");
      PrintChar(params.APSDE_DATA_indication
        .SrcAddress.ShortAddr.byte.MSB);
      PrintChar(params.APSDE_DATA_indication
        .SrcAddress.ShortAddr.byte.LSB);
            printf("\r\n");


          ConsolePutROMString( (ROM char *)"Received␣" );
    }
    printf("\r\n");
    TimeDuration time;
    time.nsec=5000;
```

```
        ERRORCODE = read( time, &PAYLOAD);
        }
    }
    break;
    .
    .
    .
}
```

6.1.2 WTIM_INPUT_CLUSTER

Questo cluster scompatta il Command Message ed in base al valore dei campi cmdClass e cmdFunct letti esegue delle spefiche operazioni, compone il relativo Reply Message e lo rispedisce all'NCAP. Nello specifico l'azione eseguita è la lettura del trasduttore di temperatura. Tale cluster è stato sviluppato in modo tale da permettere in futuro di aggiungere la decodifica di tutte le istruzioni IEEE1451.

Listing 6.2: ,RFD.c WTIMin

```
case WTIM_INPUT_CLUSTER:
{
  UInt8 rplyLenHi;
  UInt8 rplyLenLo;
  UInt8 trCommIdHi;
  UInt8 trCommIdLo;
  UInt8 dataOffset;
  /*Scompatta il Command Message*/
  UInt8 lenght=APLGet();
  trCommIdHi=APLGet();
  trCommIdLo=APLGet();
  cmdClass=APLGet();
  cmdFunct=APLGet();
```

```
rplyLenHi=APLGet();
rplyLenLo=APLGet();
dataOffset=APLGet();
/*Switch sulla classe di comando*/
switch(cmdClass){
  case COMMONCMD :
  break;
  case XDCRIDLE:
  break;
  case XDCROPERATE:
  {
    /*Switch sulla funzione specifica*/
    switch(cmdFunct)
    {
    case READDATASET:
    {
      /*Costruisce Reply message*/
      char *tr;
      rplyLenHi=0x00;
      rplyLenLo=0x0B;
      TxBuffer[TxData++] = SUCCESS;
      TxBuffer[TxData++] = rplyLenHi;
      TxBuffer[TxData++] = rplyLenLo;
      TxBuffer[TxData++] = dataOffset;

      tr = (char *) &(TxBuffer[TxData]);
      tr++;
      /*Legge il trasduttore di temperatura*/
      TxBuffer[TxData] = GetTC77String( tr );
      ConsolePutString( (BYTE *)tr );
      TxData += TxBuffer[TxData] + 1;
```

```
    /* don't bother sending data to myself */
    if(params.APSDE_DATA_indication.SrcAddress.ShortAddr.Val
        ==(macPIB.macShortAddress.Val))
    {
      APSDiscardRx();
      currentPrimitive = NO_PRIMITIVE;
      break;
    }
    /* package and send response */
    ZigBeeBlockTx();
    params.APSDE_DATA_request.DstAddrMode
      = params.APSDE_DATA_indication.SrcAddrMode;
    params.APSDE_DATA_request.DstAddress.ShortAddr
      = params.APSDE_DATA_indication.SrcAddress.ShortAddr;
    params.APSDE_DATA_request.RadiusCounter = DEFAULT_RADIUS;
    params.APSDE_DATA_request.DiscoverRoute
      = ROUTE_DISCOVERY_SUPPRESS;
    params.APSDE_DATA_request.TxOptions.Val = 0;
    i = params.APSDE_DATA_indication.SrcEndpoint;
    params.APSDE_DATA_request.SrcEndpoint
      = params.APSDE_DATA_indication.DstEndpoint;
    params.APSDE_DATA_request.DstEndpoint =  i;
    params.APSDE_DATA_request.ClusterId.Val
      = WTIM_OUTPUT_CLUSTER;
    currentPrimitive = APSDE_DATA_request;
    break;
  }
  /*Tutti gli altri casi*/
  case WRITEDATASET:
  break;
```

```
        case TRIGGERCOMMAND:
        break;

        case ABORTTRIGGER:
        break;
        }
    }
    /*Tutti gli altri casi*/
    case XDCREITHER:
    break;
    case TIMSLEEP:
    break;
    case TIMACTIVE:
    break;
    case ANYSTATE:
    break
  }
  break;
}
```

6.2 ProcessMenu()

Questa funzione all'apparenza banale, è in realtà fondamentale; al suo interno infatti è descritto, mediante l'uso delle funzioni sviluppate, tutto il protocollo di comunicazione definito nello standard IEEE1451.0 .

Questo metodo gestisce la console video (Microsoft HyperTerminal) e gli input da tastiera. Scegliendo di leggere il dato di temperatura la funzione chiama in successione tutti i metodi descritti in precedenza secondo le regole definite nello standard.

Listing 6.3: ,RFD.c WTIMin

```
void ProcessMenu( void )
```

```
{
.
.
.
switch (c)
{
/*RICHIESTA DATO DI TEMPERATURA A WTIM TRAMITE METODO READ */
case '1':
{
UInt8Array *moduleIds;
UInt8 moduleId;
UInt16 timId;
UInt16 chnl;
UInt16 trCommId;
UInt8 count;
Boolean last;
TimeDuration tempo;
tempo.nsec=5000;
ERRORCODE = reportCommModule(moduleIds);
do
{
   ConsolePutROMString((ROM char *)"SELECT
␣␣␣␣␣␣␣␣␣MODULE\r\n␣PRESS␣5␣FOR:␣1451.5␣");
   c = ConsoleGet();
   ConsolePut(c);
   c-=0x30; //c e' un carattere
}while(c !=0x5);
moduleId = moduleIds[c];  //SELEZIONA IL MODULO 1451.5
ERRORCODE = reportTims(&moduleId, timIds);
do
{
```

```
  ConsolePutROMString((ROM char *)"\r\n\nSELECT␣1451.5␣TIM\r\n");
  //Stampa lista con numero dei tim
  for(count=0; count<nTimsDot5; count++)
  {
    ConsolePutROMString((ROM char *)"");
    PrintChar(count+1);
    ConsolePutROMString((ROM char *)"␣␣:␣TIM_");
    PrintChar(count+1);
  }
  c = ConsoleGet();
  c-=0x30;
  timId = *(timIds+c);          //SELEZIONA IL TIM 1
}while(c!=0x1);

ERRORCODE = reportChannels(&timId, channels/*, StringArray names*/);
do
{
  ConsolePutROMString((ROM char *)"\r\n\nTIM␣CHANNEL\r\n");
  ConsolePutROMString((ROM char *)"␣␣␣0:␣TEMPERATURE␣");
  c = ConsoleGet();
  c-=0x30;
}while(c!=0x0);
//SELEZIONA IL CANALE TEMPERATURE
chnl=channelList[moduleId][timId][c];
//APRI CANALE COMUNICAZIONE
ERRORCODE = open(&timId, &chnl, &trCommId );
//IMPOSTA IL PAYLOAD
ERRORCODE = readData(&trCommId, tempo, &PAYLOAD );
//INVIA IL COMANDO AL WTIM
ERRORCODE = write( tempo, &PAYLOAD, last);
```

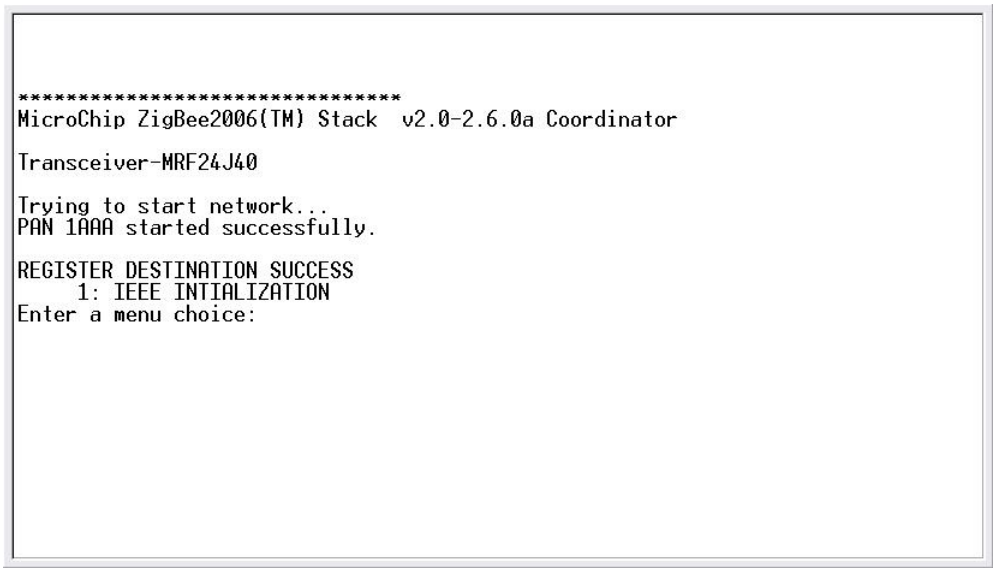
```
    ConsolePutROMString((ROM char *)"\r\nWRITE␣COMMAND␣");
    if(ERRORCODE==0)
    ConsolePutROMString((ROM char *)"SUCCESS␣");
    else
    {
      ConsolePutROMString((ROM char *)"ERROR␣");
      PrintChar((char) ERRORCODE);
    }
  }
  break;
  .
  .
  .
}
```

Capitolo 7

Riassunto, analisi con Zena e conclusioni

7.1 Riassunto

Trattandosi di un progetto che si snoda tra svariati standard e protocolli molto contorti risulta comodo riassumere il lavoro svolto per focalizzare l'attenzione su quanto è stato sviluppato. All'avvio l'NCAP crea una rete e manda a video un menu tramite il quale l'utente può inizializzare la rete per utilizzare le funzionalità dello standard IEEE1451 (Figura 7.1). L'utente tramite

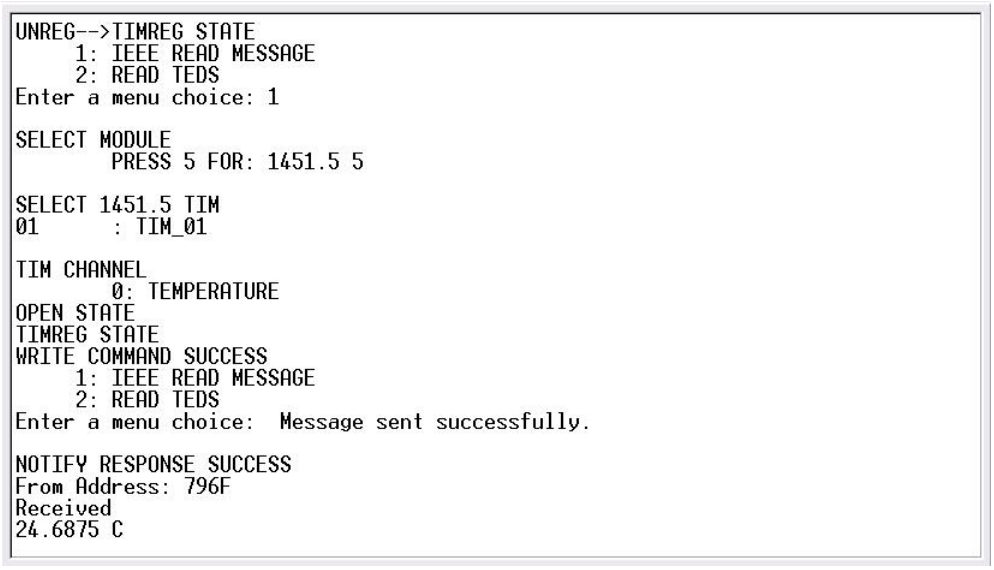


console inizializza il dispositivo (Figura 7.1). a questo punto il WTIM può collegarsi autonomamente alla rete. L'NCAP lo riconosce e lo registra inserendolo nella lista dei TIM Dot5. L'NCAP visualizza quindi un menù con le opzioni di comunicazioni specifiche per quel TIM (Figura7.1) L'utente a questo punto seleziona la lettura del messaggio (il dato di temperatura)

```
*********************************
MicroChip ZigBee2006(TM) Stack  v2.0-2.6.0a Coordinator

Transceiver-MRF24J40

Trying to start network...
PAN 1AAA started successfully.

REGISTER DESTINATION SUCCESS
     1: IEEE INTIALIZATION
Enter a menu choice: 1
INITIALIZATION SUCCESS
```

```
*********************************
MicroChip ZigBee2006(TM) Stack  v2.0-2.6.0a Coordinator

Transceiver-MRF24J40

Trying to start network...
PAN 1AAA started successfully.

REGISTER DESTINATION SUCCESS
     1: IEEE INTIALIZATION
Enter a menu choice: 1
INITIALIZATION SUCCESS
Node 796F With MAC Address 0000000000000002 just joined.

UNREG-->TIMREG STATE
     1: IEEE READ MESSAGE
     2: READ TEDS
Enter a menu choice:
```

e l'NCAP fornisce le liste di moduli, TIM e canali presenti da cui l'utente può selezionare il dispositivo desiderato (Figura 7.1). In questa fase l'utente 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. A questo



punto parte la sequenza automatica di invio ricezione:

- 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 la primitiva *APSDE_DATA_request* al ciclo macchina interno.

- 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, decodificata la richiesta del dato di 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.

```
UNREG-->TIMREG STATE
     1: IEEE READ MESSAGE
     2: READ TEDS
Enter a menu choice: 1

SELECT MODULE
        PRESS 5 FOR: 1451.5 5

SELECT 1451.5 TIM
01      : TIM_01

TIM CHANNEL
        0: TEMPERATURE
OPEN STATE
TIMREG STATE
WRITE COMMAND SUCCESS
     1: IEEE READ MESSAGE
     2: READ TEDS
Enter a menu choice:  Message sent successfully.

NOTIFY RESPONSE SUCCESS
From Address: 796F
Received
24.6875 C
```

7.2 Analisi dei pacchetti con Zena

La Figura 7.2 mostra i pacchetti trasmessi durante l'operazione di lettura del dato. Il campo in verde contiene informazioni su mittente/destinatario e sul numero del pacchetto.
Il campo in giallo evidenzia il tipo di pacchetto, il clusterId con il quale tale messaggio va scompattato e le informazioni sull'endpoint di origine e quello di destinazione.
Il campo in azzuro rappresenta il dato vero e proprio associato a quel pacchetto.

Il primo pacchetto in alto è il messaggio di richiesta del dato inviato dall'NCAP; è interessante notare l'indirizzo del mittente 0x796f (l'NCAP), il clusterId 0x0006 (corrispondente a

NCAPout o WTIMin) e il contenuto del dato trasmesso. Analizzando quest'ultimo si possono notare tutti i campi del Command Message relativo all'invio del comando Read Data-Set:

- 0x07: è la lunghezza del payload (7 bytes)
- 0xAA: è il byte più significativo del `transCommId`
- 0xAA: è il byte meno significativo del `transCommId`
- 0x03: è l'identificativo della classe del comando inviato (XdcrOperate), ossia il campo `cmdClass`
- 0x01: è l'identificativo del comando specifico per la classe selezionata (Read Transducer Channel data-set), ossia il campo `cmdFunct`
- 0x00: è il byte più significativo del numero di otetti dipendenti
- 0x01: è il byte meno significativo del numero di otetti dipendenti
- 0x00: è il valore dell'attributo dell'unico ottetto dipendente (offset)

Il secondo pacchetto è un messaggio di acknowledgement inviato dal WTIM per informare l'N-CAP che il messaggio è stato trasmesso correttamente. Il terzo pacchetto è la risposta del WTIM contenente il Reply Message formattato con il dato di temperatura. Anche qui nel payload si identificano

- 0x00: è il flag di Successo/Fallimento. In questo caso vale zero perché non ci sono stati errori di trasmissione
- 0x00: è il byte più significativo del numero di ottetti dipendenti trasmessi
- 0xB0: è il byte meno significativo del numero di ottetti dipendenti trasmessi

I successivi 11 bytes sono il dato di temperatura vero e proprio.

0x1A
NWK Frame Control
Type Ver Route Sec
DAT 0x2 SUP N
Dest Addr
0x796F
Source Addr
0x0000
Radius
0x0A
Seq Num
0x01
APS Frame Control
Type Deliv Mode Sec ACK
DAT UNI N/A N Y
Dest EP
0xF0
Cluster ID
0x0006
Profile ID
0x0103
Source EP
0x01
APS Counter
0x00
AF Data
0x07 0xAA 0xAA 0x03
0x01 0x00 0x01 0x00
FCS
0x02EB
0x1A
NWK Frame Control
Type Ver Route Sec
DAT 0x2 EN N
Dest Addr
0x0000
Source Addr
0x796F
Radius
0x0A
Seq Num
0x3E
APS Frame Control
Type Deliv Mode Sec ACK
ACK UNI N/A N N
Dest EP
0x01
Cluster ID
0x0006
Profile ID
0x0103
Source EP
0xF0
APS Counter
0x00
FCS
0x02E8
0x1A
MAC Payload
0x04
FCS
0x02E9
0x1A
MAC Payload
0x04
FCS
0x02EB
0x1A
NWK Frame Control
Type Ver Route Sec
DAT 0x2 SUP N
Dest Addr
0x0000
Source Addr
0x796F
Radius
0x0A
Seq Num
0x3F
APS Frame Control
Type Deliv Mode Sec ACK
DAT UNI N/A N N
Dest EP
0x01
Cluster ID
0x0005
Profile ID
0x0103
Source EP
0xF0
APS Counter
0x02
AF Data
0x00 0x00 0x0B 0x00 0x09 0x32 0x35
0x2E 0x31 0x38 0x37 0x35 0x20 0x43
FCS
0x02EB
0x1A
FCS
0x01E8
0x1A
FCS
0x01E9

7.3 Conclusioni

Lo standard IEEE1451 offre delle notevoli potenzialità, ma allo stato attuale delle cose il programmatore ha a mio avviso troppa libertà di interpretazione/implementazione dello stesso. Questo implica che l'utilizzo di questa uniformazione rimane valido fintanto che viene adottata dallo stesso team di sviluppatori di driver per dispositivi hardware. Ciò purtroppo limita l'opportunità di connessione plug ' n play a dispositivi con IEEE1451 implementato nello stesso modo. Questo però non intacca il punto di forza di tale standard: se gli sviluppatori si trovano a dover utilizzare dispositivi con protocolli di comunicazione diversi che devono essere connessi nella medesima rete, hanno in mano uno strumento potentissimo.

Relativamente allo sviluppo di un progetto definito, IEEE1451 fornisce pertanto una rapida e semplice soluzione di interfacciamento.

I moduli per interrogare i dispositivi da remoto sono stati implementati con una struttura tale da permetterne un'adattamento semplice in funzione del tipo di accesso che si vuole utilizzare.

L'obbiettivo principale raggiunto in questo progetto, al di là dell'implementazione dello standard, è stato rendere ZigBee trasparente al dato trasmesso. ZigBee non è quindi a conoscenza di ciò che sta trasmettendo ma tratta i dati caricati nel suo buffer di trasmissione come dei bytes qualsiasi. E' compito di IEEE1451 ricomporre i pacchetti in modo tale da estrarne l'informazione associata.

Qui sta il punto di forza di questa implementazione: non è necessario informare il protocollo fisico impiegato per la comunicazione sul tipo di dato che sta inviando.

Capitolo 8

Bibliografia

IEEE Std. 1451.0. IEEE Instrumentation and Measurement Society, 2007.

IEEE Std. 1451.5. IEEE Instrumentation and Measurement Society, 2007.

ZigBee Specifications. ZigBee Alliance, 2008.

AN1232, Microchip ZigBee-2006 Residential Stack Protocol. Microchip Tech- nology Inc., 2006.

PICDEM Z Demonstration Kit User's Guide. Microchip Technology Inc., 2007.

M. Marino. Introduzione al protocollo wireless ZigBee. M.M.Electronics, 2008