Università degli Studi di Padova
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea Magistrale in Ingegneria Elettronica

Valutazione sperimentale delle performance di WBSN:

realizzazione e validazione di sistemi di test

Relatore
Prof.ssa Giada Giorgi

Laureando
Matteo Osti
Matr. 1106445

A.A. 2016/2017

9 Ottobre 2017

Indice

Capitolo 1

Introduzione

Una Wireless Body Sensor Network (WBSN) è una rete senza fili di sensori, progettata per interfacciarsi con dispositivi a bassa potenza e dal ridotto consumo energetico. Nella forma più generale può essere composta da più sensori di tipo eterogeneo, caratterizzati anche da rate di campionamento diversi tra loro.
I segnali acquisiti possono essere di qualunque tipo: dalle funzioni vitali (pressione sanguigna, livello di glucosio) fino a parametri bio-cinetici o ambientali, sfruttando anche sensori indossabili o impiantabili.

Si vuole realizzare un sistema di test, sufficientemente flessibile e semplice per la valutazione delle prestazioni di una WBSN, ed in particolare degli algoritmi locali di elaborazione dati. La rete costruita quindi avrà una topologia a stella, con un singolo nodo aggregatore centrale che si interfaccerà con un sistema di storage e visualizzazione (tipicamente un computer). I nodi-sensore che si intende utilizzare avranno bassi rate di comunicazione (nell'ordine della decina di pacchetti al secondo) e saranno costituiti da schede a singolo microcontrollore.

Una volta costruita, la rete deve essere caratterizzata, analizzandone problematiche e limitazioni, rappresentate da collisioni tra pacchetti provenienti da più nodi, corruzione di dati e perdite di pacchetti, senza dimenticare i tempi di latenza introdotti dal sistema a microcontrollore che gestisce ogni singolo nodo. Si vogliono inoltre testare e quantificare i limiti della rete, nel caso ad esempio si voglia fare uno streaming di dati.

Il setup di misura verrà integrato con dispositivi in grado di generare dei segnali sintetici, utilizzando un generatore di funzioni, e quindi simulando anche un generico apparato di acquisizione tramite un modulo ADC interno al nodo, o caricando direttamente i valori numerici da trasmettere.
Una volta creata e configurata la rete di trasmissione sarà necessario valutarne le caratteristiche, quali ad esempio il rate di trasmissione e la struttura dei pacchetti.
Poiché si ha a che fare con un protocollo wireless a basso bit rate, si devono poter valutare anche le caratteristiche degli algoritmi di pre-elaborazione e di compressione dei dati, sia dal punto di vista temporale sia dal punto di vista energetico, oltre che dalla valutazione dell'occupazione in memoria.

Capitolo 2

Scopo della tesi

Come già riportato nell'introduzione, una WBSN è una rete senza fili di sensori, progettata per interfacciarsi con dispositivi a bassa potenza.
La topologia tipica per questo tipo di reti è quella a stella, in cui ogni dispositivo comunica soltanto con un singolo nodo aggregatore centrale.

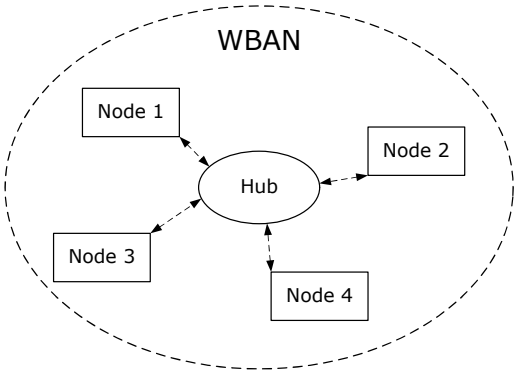

Figura 2.1: Topologia per una tipica rete WBAN.

In Figura 2.1 si può osservare una rete WBSN a stella, al cui interno l'*hub* è il nodo connesso a tutti gli altri ed è di solito quello dotato della maggiore potenza di calcolo, quindi oltre che della ricezione, si incarica dell'elaborazione dei valori ricevuti. I nodi sensore, invece, si occupano dell'acquisizione del dato e di una eventuale pre elaborazione. Le WBSN sono definite nello standard IEEE 802.15.6, anche se implementabili anche con altri protocolli, in cui si riportano proprietà e struttura di reti a breve raggio. Poiché tale protocollo è stato sviluppato per l'utilizzo anche in campo biomedicale, i dispositivi devono operare con valori molto bassi di potenza, per minimizzare il valore di SAR.

L'obiettivo della tesi è proprio quello della realizzazione di un'architettura di questo tipo, per poterne successivamente testare le caratteristiche ed i limiti di utilizzo.
Per l'implementazione fisica di questo apparato si incontra immediatamente un ostacolo: non esiste alcun transceiver commerciale conforme allo standard 802.15.6.
Questo inconveniente obbliga a cambiare tipologia di rete, nello specifico quella definita dallo standard IEEE 802.15.4, riguardante reti LR-WPAN (Low Rate Wireless Personal Area Network), con cui è possibile costruire una rete che operi analogamente ad una WBSN.
Poiché la velocità massima di trasmissione è in entrambi i protocolli relativamente bassa, ciò consente l'implementazione dei nodi anche su device la cui potenza di calcolo non è molto elevata, come ad esempio dei microcontrollori.
L'unità di elaborazione del nodo viene affiancata da un transceiver conforme allo standard

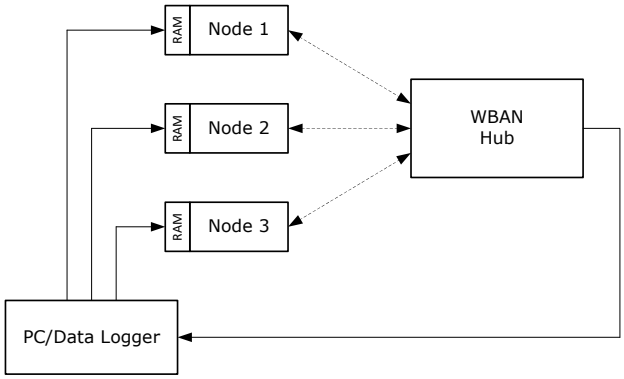

Figura 2.2: Schema di principio per il test dell'apparato, si caricano dei dati numerici nella memoria dei nodi. L'hub li ritrasmette al pc per la verifica.

802.15.4 di tipo commerciale per alleggerire il carico computazionale nei confronti del microcontrollore sull'invio e ricezione di pacchetti.
Una volta realizzata ed inizializzata la comunicazione tra i nodi, si organizza un sistema di test in cui vengono generati e ricostruiti dei segnali sintetici. Una prima soluzione, rappresentata schematicamente in Figura 2.2, utilizza una connessione seriale per il caricamento di alcune sequenze di dati all'interno della memoria locale dei nodi, che vengono confrontate con quelle in output dal nodo aggregatore. Una seconda soluzione (Figura 2.3) include l'utilizzo dell'apparato di acquisizione del nodo-sensore, connettendolo ad un generatore di funzioni. La forma d'onda viene campionata opportunamente, inviata all'hub e quindi riconvertita in analogico attraverso un DAC per poterla confrontare con quella iniziale.
Altre misure necessarie per la caratterizzazione del sistema sono quelle relative alle latenze introdotte dall'unità di elaborazione nelle varie fasi della trasmissione e ricezione del pacchetto.

Per il buon funzionamento del sistema è opportuno valutare le operazioni svolte dal microcontrollore ed identificare gli eventuali passaggi temporalmente critici, ottimizzando il più possibile il codice per ottenere prestazioni migliori, mantenendo al contempo una sufficiente flessibilità per l'esecuzione di studi di fattibilità.

Il sistema progettato sarà utile per testare sperimentalmente algoritmi di pre elaborazione, compressione dati, ecc. . Grazie a sistemi con architettura semplice è facile infatti ricavare il costo compuazionale e l'efficienza degli algoritmi implementati, anche dal punto di vista del consumo energetico.

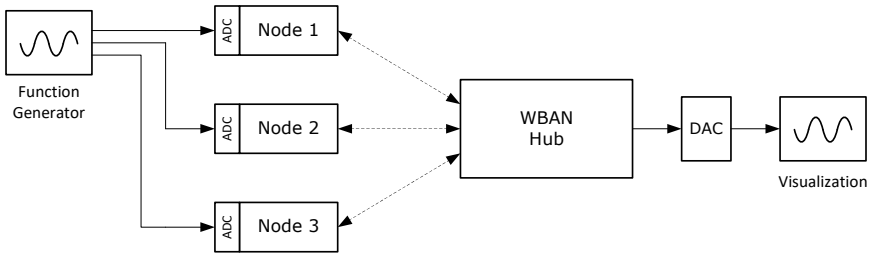

Figura 2.3: Schema di principio per il test dell'apparato, i segnali analogici generati vengono acquisiti dai nodi e ricostruiti a valle dell'hub.

Capitolo 3

Standard IEEE 802.15.4

Lo standard IEEE 802.15.4 [1] è un protocollo di comunicazione senza fili di tipo LR-WPAN, Personal Area Networks a basso bitrate.
Fa parte del gruppo più ampio regolato dallo standard 802.15, che definisce appunto proprio la tipologia Personal Area Network (PAN), comprendente anche la comunicazione Bluetooth (802.15.1).
IEEE 802.15.4 definisce le specifiche ai livelli più bassi del modello ISO/OSI, in particolare si limita a layer PHY (fisico) e al sublayer MAC (Medium Access Control). I livelli superiori vengono lasciati liberi per poter essere meglio adattati alla specifica applicazione. A partire da ciò che non è stato specificato sono stati creati altri protocolli, come ZigBee, curato da ZigBee Alliance e MiWi, sviluppato da Microchip Technology.
Gli obbiettivi principali di questo tipo di reti sono quelli di fornire una comunicazione affidabile, di semplice installazione e manutenzione, adatta anche a dispositivi alimentati a batteria, mantenendo al contempo flessibilità e semplicità.
Alcune delle caratteristiche di una rete a basso bitrate (LR-WPAN) costruita con questo protocollo sono:

- Rate di trasmissione fino a 250 kb/s
- Topologia a stella o peer-to-peer
- Allocazione opzionale di Guaranteed Time Slots (GTS)
- Accesso al canale Carrier Sense Multiple Access con Collision Avoidance (CSMA-CA)
- Protocollo con acknowledgment per trasmissioni affidabili
- Basso consumo energetico
- Energy detection (ED)
- Link quality indication (LQI)

3.1 Topologia di rete

Una rete di dispositivi costruita basandosi su questo standard può essere creata tenendo conto che i nodi presenti possono essere di due tipi: FFD, full-function device o RFD, reduced-function device. Un nodo di tipo FFD è un dispositivo in grado di gestire un flusso di dati proveniente da altri dispositivi, ed ha una capacità di calcolo sufficientemente elevata. Un nodo di questo tipo può ricoprire all'interno della rete il ruolo di PAN-coordinator, coordinator, o device; può dialogare sia con altri FFD sia con RFD. Un RFD è tipicamente identificato come un nodo sensore molto semplice, il cui traffico dati è molto limitato, che

può comunicare soltanto con un solo nodo di tipo FFD. Di conseguenza questo dispositivo è particolarmente adatto ad essere implementato su un dispositivo che usa risorse minime.

Per la creazione della rete sarà quindi necessario almeno un nodo di tipo FFD, che ricoprirà il ruolo di coordinatore.
Dipendentemente dai requisiti dell'applicazione, una rete che sfrutta questo protocollo può operare sfruttando due topologie: rete a stella o peer-to-peer.
Tutti i dispositivi operanti all'interno di una PAN, qualsiasi topologia essa abbia, devono avere un indirizzo a 64 bit unico.

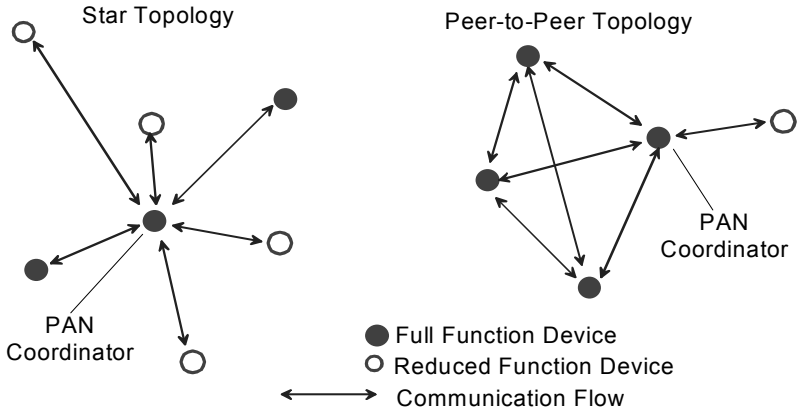

Figura 3.1: Topologie di rete per IEEE 802.15.4.

In una rete a stella la connessione viene creata tra i device ed un singolo nodo centrale, il coordinatore di rete, che può avere un ruolo specifico o può essere sfruttato anche solo per effettuare le operazioni di connessione, disconnessione o reindirizzamento dei pacchetti all'interno della rete. Il coordinatore di rete di solito viene alimentato tramite rete elettrica, per evitare che la PAN stessa sia dipendente dalle risorse energetiche disponibili, mentre gli altri nodi possono essere alimentati sia tramite batteria sia tramite alimentazione di rete. Tipiche applicazioni che fanno uso di reti a stella (non necessariamente legati a questo standard) sono le periferiche di un PC, sistemi di domotica, dispositivi legati all'health-care o reti di sensori.

Una rete peer-to-peer è composta anch'essa da un coordinatore di rete, ma differisce dalla rete a stella dal fatto che ogni dispositivo possa dialogare con un altro, purché sia all'interno del range di comunicazione. Una rete di questo tipo è sicuramente più complessa, e può prevedere più passaggi per consegnare dati da un device all'altro attraverso la rete.

Una PAN indipendente è identificata da un identificativo unico, che permette la comunicazione dei nodi tramite un indirizzo di tipo short (16 bit) e consente la comunicazione tra nodi di reti diverse, formando una cosiddetta *rete cluster*, rappresentata in Figura 3.2. Il modo in cui viene composta la rete non è prevista nello standard.

3.1.1 Formazione di una rete peer-to-peer

In una rete peer-to-peer ogni dispositivo (FFD) è in grado di comunicare con ogni altro nodo all'interno della sfera di influenza. Un nodo diviene PAN coordinator, ad esempio, se è il primo a trasmettere sul canale scelto.
Un esempio di rete peer-to-peer complessa è quella definita come *cluster tree network*. In questo tipo di rete la maggior parte dei dispositivi è di tipo FFD. In questo tipo di reti è possibile che più reti PAN si uniscano per creare una rete più complessa, che consente

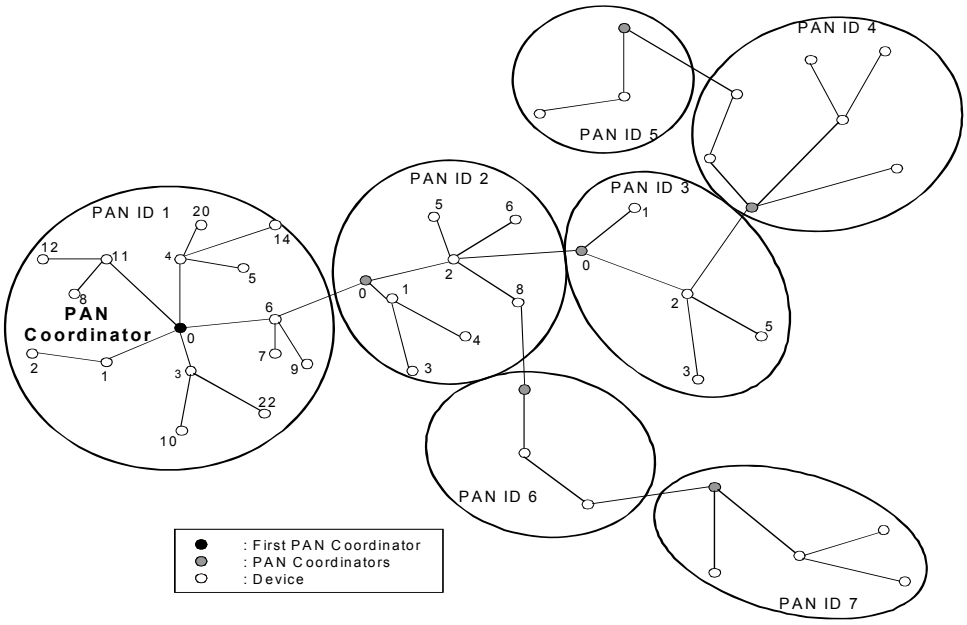

Figura 3.2: Rete cluster.

di consegnare dei messaggi anche tra nodi molto lontani tra loro. Ogni sottorete PAN ha quindi il proprio coordinatore, che gestisce le operazioni di routing. Nella figura viene riportato un esempio di rete multicluster, che ha il vantaggio di avere un range di copertura molto ampio.

3.1.2 Formazione di una rete a stella

Una volta attivato il primo dispositivo FFD, esso definisce i parametri della propria rete e ne diventa il coordinatore. Tutte le reti a stella operano indipendentemente da tutte le altre. Il PAN identifier della rete viene scelto in modo da essere unico rispetto a tutte le altre reti presenti nel raggio di comunicazione. Una volta inizializzata la rete il coordinatore può consentire agli altri device di connettersi.

3.1.3 Funzionamento della rete

Oltre alla topologia della rete, una seconda caratteristica riguarda la trasmissione dei dati. Questo standard consente, oltre ad uno scambio senza troppi vincoli dei pacchetti tra i vari nodi, anche una struttura cosiddetta a superframe, in cui i pacchetti possono essere trasmessi soltanto in determinati slot temporali.
La struttura a superframe dello standard IEEE 802.15.4 viene definita dal coordinatore della rete ed è delimitato da *network beacon*, inviati dal coordinatore stesso (Figura 3.3a). Il superframe può essere eventualmente seguito da un periodo inattivo (Figura 3.3b), in cui il coordinatore potrebbe entrare in modalità risparmio energetico. I beacon vengono utilizzati per sincronizzare i nodi connessi, per identificare la PAN e per descrivere la struttura dei superframe. Questi ultimi possono essere suddivisi in due porzioni, un Contention Access Period (CAP), in cui ogni nodo deve comunicare usando uno Slotted CSMA, ed un Contention-Free Period (CFP, o Guaranteed Time Slots, GTS), ovvero degli slot riservati per determinati nodi, in questa porzione per accedere al canale è diretto (Figura 3.3c).

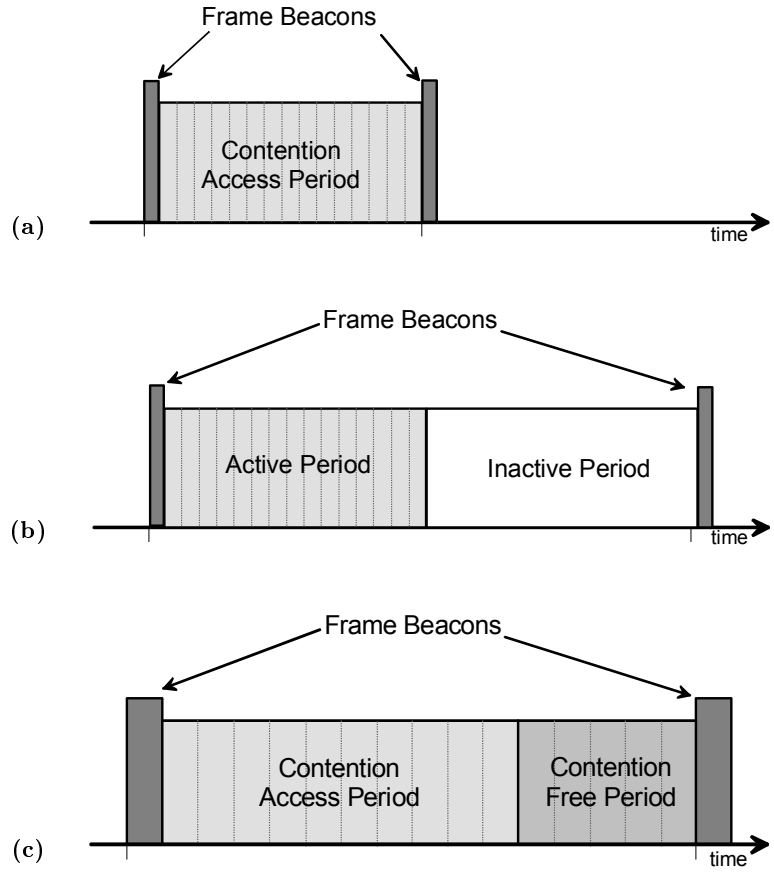

Figura 3.3: Superframe per lo standard IEEE 802.15.4. Superframe senza (a) e con (b) periodo inattivo, (c) dettaglio degli slot.

3.1.4 Comunicazione tra nodi

La comunicazione tra nodi avviene secondo delle dinamiche di dialogo ben definite, e dipende dalle azioni che si devono operare sulla rete. Ora verranno affrontate le dinamiche più comuni, e quelle che più si adattano allo scopo di questa tesi.

3.1.4.1 Trasferimento dati

Quando un device deve trasmettere dei dati ad un coordinatore in una rete beacon-enabled (Figura 3.4a), deve attendere il beacon e soltanto in seguito potrà trasmettere le informazioni, usando Slotted CSMA-CA. Il coordinatore può inviare o meno un acknowledgment. Quando invece si è in una rete non-beacon le dinamiche sono più sempllici, in quanto il nodo deve soltanto rispettare le tempistiche date dall'accesso al canale, in questo caso unslotted CSMA-CA, il coordinatore invierà un ACK, se richiesto (Figura 3.4b).
Se invece il dato deve essere trasferito al device dal coordinatore in una rete beacon-enabled (Figura 3.4c), nel beacon frame viene indicato che è presente un messaggio in coda mediante il flag di frame pending. Il device, una volta visto il flag di frame pending, invierà un frame di richiesta dati usando slotted CSMA-CA ed il coordinatore invierà un acknowledgment seguito, se possibile, dal dato richiesto. A questo punto il device notifica l'avvenuta ricezione inviando un ACK. Il messaggio appena inviato viene rimosso dalla coda dei messaggi in sospeso del coordinatore.
Infine per una rete non-beacon (Figura 3.4d), il device invierà direttamente un frame di richiesta dati, con le stesse successive comunicazioni del caso beacon.

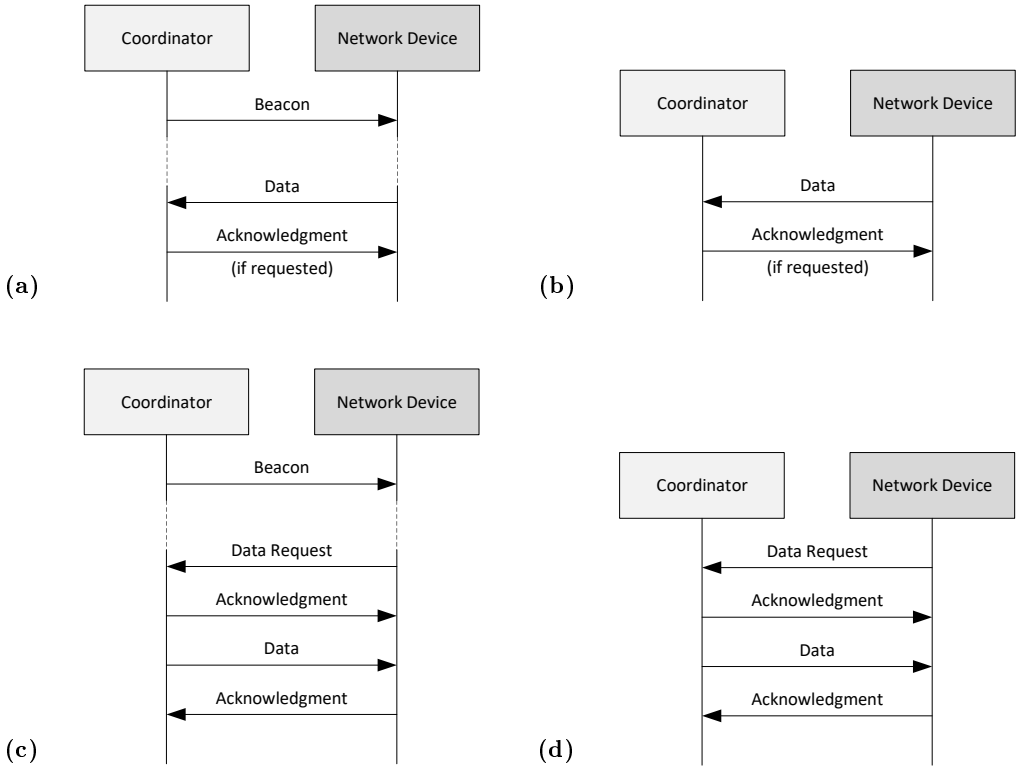

Figura 3.4: Dettaglio della trasmissione di un pacchetto dati: (a) e (b) per un'architettura beacon-enabled, (c) e (d) per una rete non-beacon.

3.1.4.2 Connessione/Disconnessione

Per l'associazione di un device alla rete PAN è necessario seguire la seguente procedura, raffigurata in Figura 3.5.
Se la rete è beacon-enabled ed il coordinatore è disponibile ad accettare nuove connessioni, il device che intende associarsi deve inviare un frame di *association request* al coordinatore, il quale risponderà con un ACK e, entro un tempo pari a `macResponseWaitTime`, comunicherà la sua decisione. Il device risponderà a sua volta con un ACK.
Il coordinatore, nella risposta data al device, comunica lo short address assegnato al nuovo dispositivo.
Soltanto in seguito allo scambio dell'ultimo ACK il device potrà essere considerato associato.

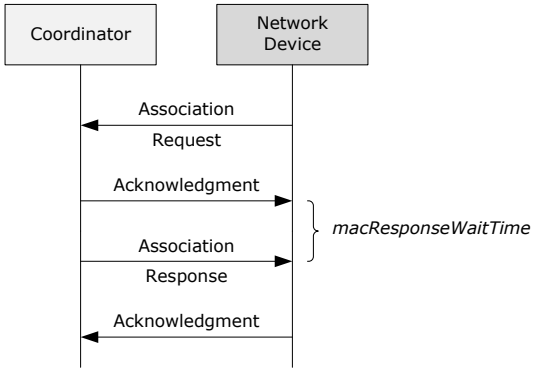

Figura 3.5: Procedura di associazione di un dispositivo ad una rete PAN.

Per quanto riguarda la disassociazione si possono verificare due scenari: il coordinatore

disconnette il device oppure il device vuole lasciare la PAN.
Nel primo caso (Figura 3.6a) il coordinatore invia un *disassociation notification* al device che vuole disconnettere, che risponderà con un ACK. In caso di pacchetto non ricevuto il coordinatore considererà il device come disconnesso. Nell'altro caso (Figura 3.6b) è il device che invia la *disassociation notification* e attende un ACK dal coordinatore. Come nel caso precedente se l'ACK non viene ricevuto il device si considererà come disconnesso.

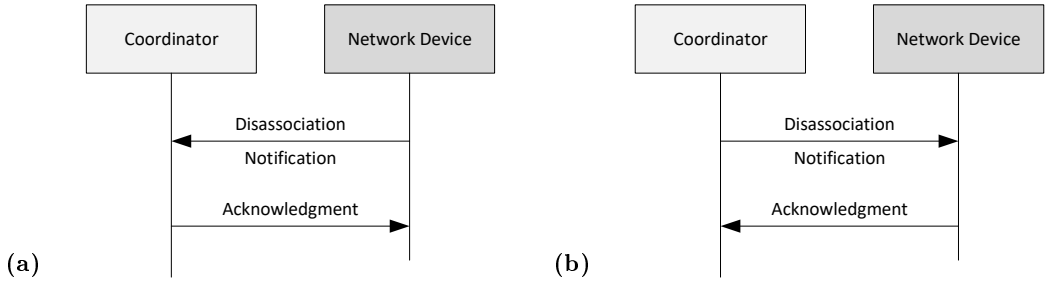

Figura 3.6: Dettaglio della procedura di disconnessione di un nodo: (a) il coordinatore disconnette il nodo, (b) il device lascia la PAN.

3.2 Layer fisico (PHY)

Il layer PHY è dedicato alle seguenti operazioni:

- Attivazione/disattivazione del transceiver radio
- Energy detection (ED) sul canale selezionato
- Link Quality Indicator (LQI) sui pacchetti ricevuti
- Clear Channel Assessment (CCA) per CSMA-CA
- Selezione del canale di trasmissione
- Trasmissione e ricezione dati

Le bande previste da questo protocollo per la trasmissione sono riportate in Tabella 3.1[1]. In particolare per la banda 2450 MHz si hanno le seguenti caratteristiche:

- Frequenze disponibili 2400 - 2483 MHz
- Modulazione O-QPSK
- Chip Rate 2000 kchip/s
- Symbol Rate 62.5 ksymbol/s
- Bit Rate 250 kbit/s

Ciascuna banda è suddivisa in più canali, in particolare per la banda 2450 MHz è presente la suddivisione in 16 canali, come illustrato in Figura 3.7, le cui frequenze centrali, in MHz, sono espresse secondo la formula

$$F_c = 2405 + 5\,(k - 11) \qquad k = 11,\, 12,\, \ldots, 26 \tag{3.1}$$

[1] aggiornato alla versione 2015

Band designation	Frequency Band (MHz)
169 MHz	169.400 – 169.475
433 MHz	433.05 – 434.79
450 MHz	450 – 470
470 MHz	470 – 510
780 MHz	779 – 787
863 MHz	863 – 870
868 MHz	868 – 868.6
896 MHz	896 – 901
901 MHz	901 – 902
915 MHz	902 – 928
917 MHz	917 – 923.5
920 MHz	920 – 928
928 MHz	928 – 960
1427 MHz	1427 – 1518
2380 MHz	2360 – 2400
2450 MHz	2400 – 2483.5
HRP UWB sub-gigahertz	250 – 750
HRP UWB low band	3244 – 4742
HRP UWB high band	5944 – 10234
LRP UWB	6289.6 – 9185.6

Tabella 3.1: Bande relative al protocollo 802.15.4 - 2015

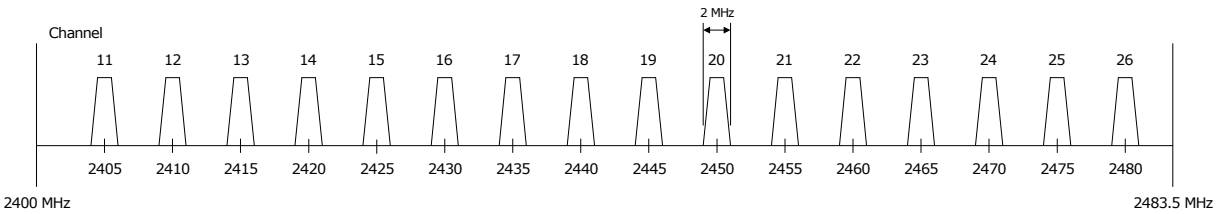

Figura 3.7: Canali disponibili nella banda ISM a 2.4 GHz.

3.2.1 Sublayer MAC

Le principali funzioni del sottolivello MAC comprendono:

- Delimitazione e riconoscimento dei frame
- Indirizzamento tra i nodi
- Trasferimento di PDU dal sottolivello superiore
- Protezione contro gli errori, generalmente generando e controllando CRC
- Controllo dell'accesso al mezzo fisico

Altre funzioni del sublayer MAC includono il controllo di flusso e l'inoltro dei frame verso i nodi vicini nel caso di trasmissione indiretta del pacchetto.

3.2.1.1 Struttura superframe

Un coordinatore su una PAN può organizzare le comunicazione utilizzando una struttura a superframe, delimitato dalla trasmissione di un frame beacon. Il coordinatore può entrare

in sleep durante la porzione inattiva.
La struttura del superframe è descritta dai valori di *macBeaconOrder* e *macSuperframeOrder*. L'attributo *macBeaconOrder* descrive l'intervallo al quale il coordinatore trasmette i beacon frames. Il Beacon Order (BO) ed il Beacon Interval (il periodo di beacon espresso in simboli, BI) sono legati dalla formula

$$BI = aBaseSuperframeDuration * 2^{BO} \qquad 0 \leq BO \leq 14 \tag{3.2}$$

Se $BO = 15$ il coordinatore non trasmette beacon automaticamente ma soltanto su esplicita richiesta da parte di un nodo (non esiste alcun superframe).
L'attributo *macSuperframeOrder* descrive la lunghezza della porzione attiva del superframe, che include il beacon frame. L'ordine del superframe (SO) e della durata del superframe (SD) sono legati dalla formula

$$SD = aBaseSuperframeDuration * 2^{SO} \qquad 0 \leq SO \leq BO \leq 14 \tag{3.3}$$

in cui se $SD = 15$ il superframe non rimane attivo dopo la trasmissione del beacon.
La porzione attiva di ogni superframe viene suddivisa in *aNumSuperframeSlots* equamente spaziati di durata

$$SlotDuration = aBaseSlotDuration * 2^{SO} \tag{3.4}$$

Il beacon viene trasmesso senza usare CSMA all'inizio dello slot 0, ed il CAP comincia immediatamente dopo. L'inizio dello slot 0 è definito come il punto al quale il primo simbolo del PPDU del beacon viene trasmesso. Il CFP, se presente, segue immediatamente il CAP e si estende fino alla fine della porzione attiva del superframe. Gli eventuali GTS sono localizzati in questa porzione.
Il sublayer MAC dovrebbe assicurare l'integrità del timing del superframe, ad esempio compensando l'errore di drift del clock.
PAN che non desiderano utilizzare la struttura a superframe (nonbeacon-enabled PAN) dovrebbero impostare *macBeaconOrder* sia *macSuperframeOrder* a 15. In questo caso tutte le trasmissioni, ad eccezione di ACK e di trasmissioni di dati che seguono l'ack di una richiesta di dati, devono usare un meccanismo Unslotted CSMA-CA per l'accesso al canale; inoltre i GTS non sono permessi.

3.2.1.1.1 Contention Access Period (CAP) Come già anticipato, il CAP dovrebbe iniziare immediatamente dopo la fine di trasmissione del beacon e completarsi prima dell'inizio del CFP al termine di uno slot del superframe. Se il CFP ha lunghezza pari a 0, il CAP termina alla fine della porzione attiva del superframe. Il CAP è lungo almeno *aMinCAPLength* simboli.
Tutti i frame ad eccezione degli acknowledgment trasmessi nel CAP dovrebbero usare Slotted CSMA-CA per effettuare l'accesso al canale.

3.2.1.1.2 Contention-Free Period (CFP) Il CFP dovrebbe cominciare sull'estremo di uno slot del superframe ed immediatamente dopo la fine del CAP. Se ogni GTS sono stati allocati dal coordinatore, dovrebbero essere localizzati in questa porzione del superframe e dovrebbero occupare slot contigui. All'interno del CFP, viste le caratteristiche di timeslot riservato, l'accesso al mezzo avviene senza CSMA-CA. Come per il CAP, il dispositivo che trasmette in questa porzione deve assicurarsi che le proprie trasmissioni abbiano termine almeno un IFS prima della fine del proprio GTS.

3.2.1.2 Accesso al canale – CSMA-CA

L'algoritmo CSMA-CA per l'accesso al canale è illustrato in Figura 3.8, e può essere Slotted o Unslotted. In entrambi i casi l'algoritmo è implementato sfruttando usando unità temporali chiamati backoff periods, in cui ognuno deve essere uguale a *aUnitBackoffPeriod* simboli. In Slotted CSMA-CA i limiti dei periodi di backoff di ogni device nella PAN devono essere allineati con quelli del coordinatore. Nella versione Unslotted i backoff di un device non sono legati temporalmente a quelli di nessun altro dispositivo nella PAN.
Per quanto riguarda l'accesso al canale sono tre le variabili da tenere in considerazione: NB, CW e BE. NB è il numero di volte che l'algoritmo trova il canale occupato, questo valore va inizializzato a 0 ogni volta che si tenta una nuova trasmissione. CW è la lunghezza della Contention Window, definendo il numero di periodi di backoff che è necessario siano liberi da qualsiasi attività sul canale prima dell'inizio di una nuova trasmissione. BE è legato al numero di periodi di backoff che un device deve aspettare prima di valutare un canale. In sistemi Unslotted, o Slotted a risparmio energetico, BE viene inizializzato al minimo tra 2 e *macMinBE*. Se *macMinBE* è impostato a zero, la collision avoidance è disattivata.

3.2.2 Pacchetti

Secondo lo standard 802.15.4 i dati che venono trasmessi da un dispositivo all'altro sono organizzati in pacchetti, che possono avere lunghezza variabile.
Un pacchetto è un'insieme di dati che viene trasmesso attraverso il mezzo fisico, che prende il nome di PPDU (PHY protocol data unit).
Al livello fisico esso prende il nome di *PHY packet* ed è composto da un header di sincronizzazione (SHR), da un header del layer (PHR) e dal relativo payload (PHY Payload).
L'SHR è composto da una sequenza di preambolo, lunga 4 ottetti, e da un delimitatore che indica l'inizio del frame, SFD (Start of Frame Delimiter), della lunghezza di 1 ottetto; il PHR indica la lunghezza del pacchetto (1 ottetto); il PSDU è il payload del livello fisico, che ha dimensione massima pari a 127 ottetti (il massimo valore contenuto nel PHR).
Il PSDU passa quindi al livello MAC, dove si suddivide in MHR, MSDU e MFR, rispettivamente header, payload e footer.
Nelle sottosezioni successive si può osservare la composizione dei diversi tipi di frame sia dal punto di vista del layer PHY, sia di quello del MAC.

I pacchetti possono essere distinti in 4 classi, che prendono il nome di frames, dipendentemente dal loro utilizzo. Possono essere suddivisi in

- *Beacon Frame*, usato da un coordinatore per trasmettere beacon
- *Data Frame*, per trasferire tutti i tipi di dati
- *Acknowledgment Frame*, per confermare una ricezione avvenuta con successo
- *MAC Command Frame*, per impartire comandi o comunicazioni di azioni particolari sulla rete

Nelle sezioni successive vengono riportati i campi principali dei pacchetti previsti dallo standard, omettendo quelli relativi al layer fisico, per il quale il transceiver opera in completa autonomia.

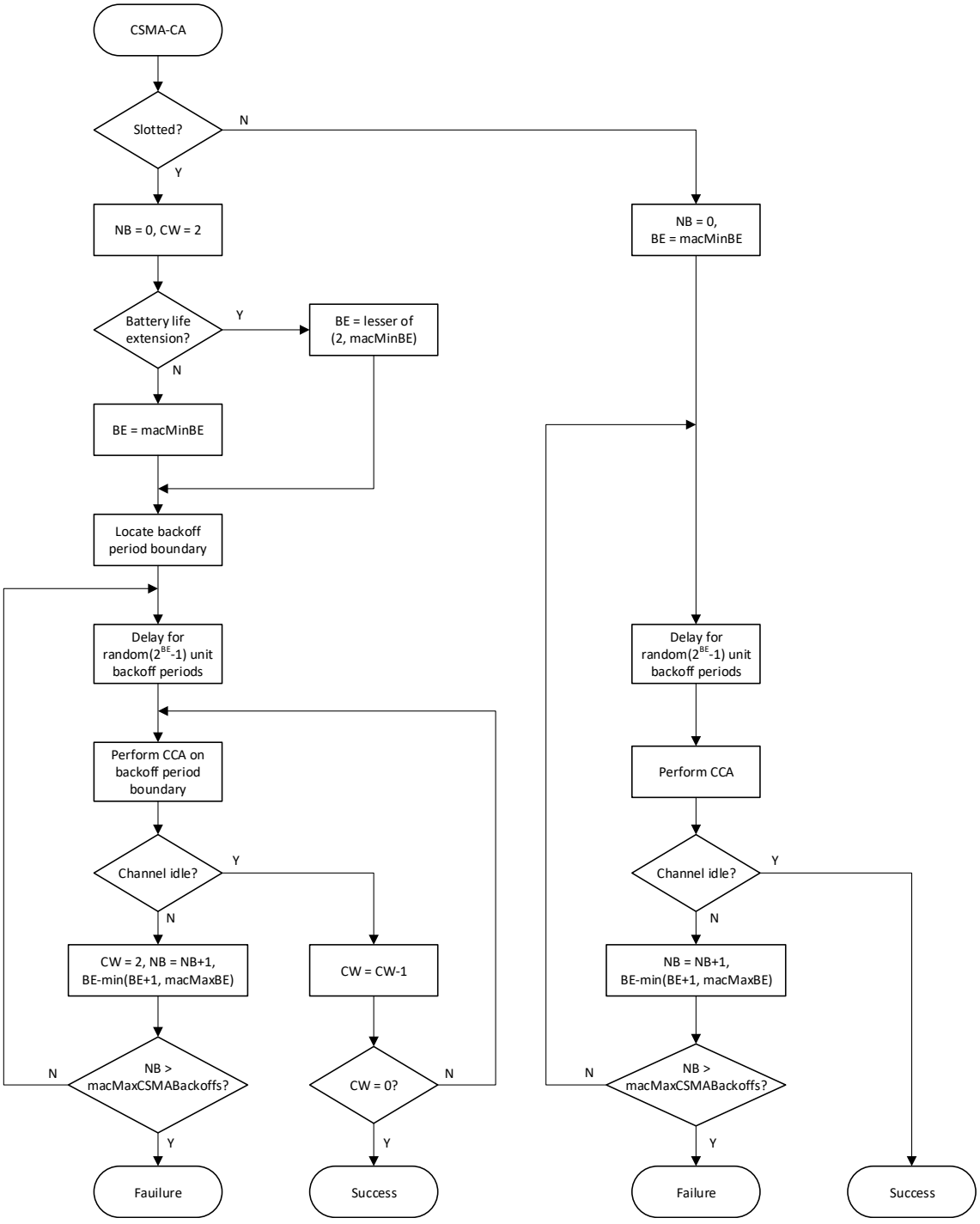

Figura 3.8: Algoritmo di accesso al canale CSMA-CA.

Octets: 2	1	0/2	0/2/8	0/2	0/2/8
Frame Control	Sequence Number	Destination PAN Identifier	Destination Address	Source PAN Identifier	Source Address
		Addressing fields			

Figura 3.9: Composizione del MAC Header.

3.2.2.1 MAC Header

L'header del sublayer MAC è schematizzato in Figura 3.9 ed è composto dai campi

- Frame Control, della dimensione di 2 ottetti, contiene i parametri che caratterizzano il pacchetto. È riportato brevemente in Figura 3.10.
- Sequence Number, della lunghezza di 1 byte, è l'indice del pacchetto in cui è contenuto.
- Addressing Fields, contiene gli indirizzi del mittente e della destinazione del pacchetto, sotto forma di indirizzo e di PAN ID di un nodo
- Auxiliary Secondary Header, che contiene informazioni riguardanti la sicurezza del frame

Bits: 0–2	3	4	5	6	7–9	10–11	12–13	14–15
Frame Type	Security Enabled	Frame Pending	Ack. Request	PAN ID Compression	Reserved	Dest. Addressing Mode	Frame Version	Source Addressing Mode

Figura 3.10: Composizione del campo Frame Control

3.2.2.2 Data Frame

È un frame generico, utilizzato per tutti i trasferimenti di dati, raffigurato in Figura 3.11. I dati che scambiati tra due nodi sono contenuti nel MAC Payload (MSDU).

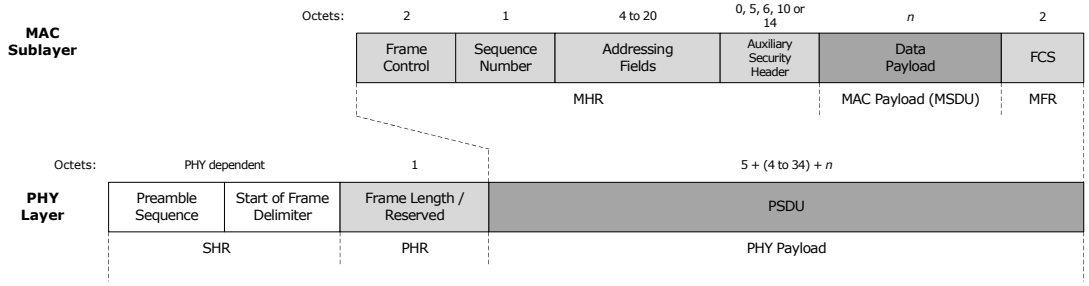

Figura 3.11: Composizione del Data Frame.

3.2.2.3 ACK Frame

Il frame di acknowledgment viene inviato per notificare l'avvenuta ricezione da parte del destinatario del pacchetto precedente. Viene inviato al nodo sorgente se la ricezione del pacchetto è avvenuta con successo. Il *Sequence Number* è lo stesso di quello del frame ricevuto.

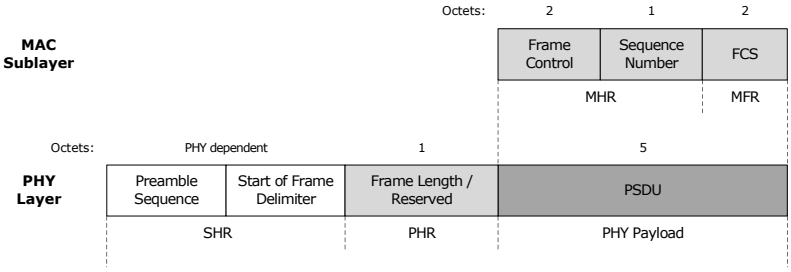

Figura 3.12: Composizione dell'ACK Frame.

3.2.2.4 Beacon Frame

Il beacon frame viene inviato dal coordinatore agli end-devices per delimitare il superframe, e informazioni come la durata del frame, la presenza di GTS e la disponibilità da parte del coordinatore di accettare connessioni.

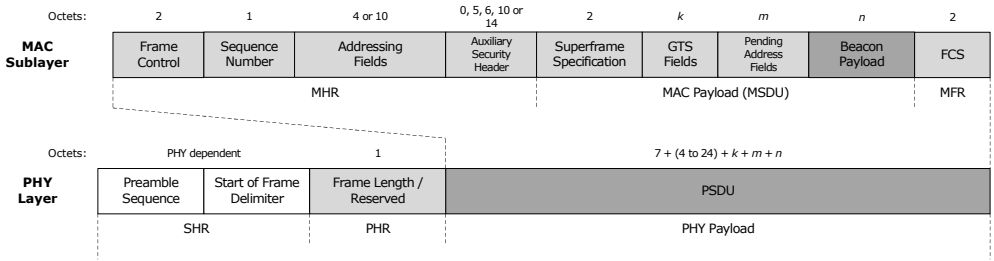

Figura 3.13: Composizione del Beacon Frame.

3.2.2.5 MAC Command Frame

Il MAC Command Frame serve per effettuare tutte le comunicazioni che riguardano la rete e la sua gestione, quali richieste di associazione, richieste di dati o di GTS. Il campo

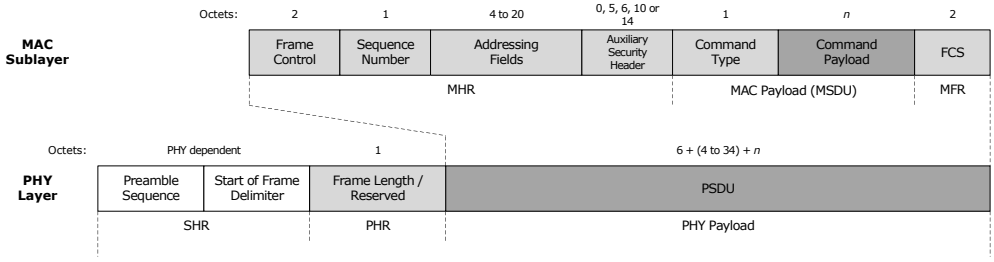

Figura 3.14: Composizione del MAC Command Frame.

Command Type identifica la funzione di tale frame, una lista dei campi è riportata in Tabella 3.2.

Command Frame Identifier	Command Name
0x01	Association request
0x02	Association response
0x03	Disassociation notification
0x04	Data request
0x05	PAN ID conflict notification
0x06	Orphan notification
0x07	Beacon request
0x08	Coordinator realignment
0x09	GTS request
0x0a–0xff	Reserved

Tabella 3.2: Valori per il bit di identificazione del comandi in un MAC Command Frame.

Capitolo 4

Descrizione dell'apparato

Per lo sviluppo del sistema viene utilizzata la scheda di sviluppo Microchip PICDEMZ, che monta un microcontrollore PIC18LF4620. L'interfaccia radio viene sviluppata utilizzando la daughter board MRF24J40MA, la cui interfaccia hardware è pensata appositamente per la main board precedentemente citata.
Per il debugging del codice e delle periferiche viene utilizzato un osilloscopio e per la verifica della comunicazione wireless si utilizza lo sniffer 802.15.4 Microchip ZENA Network Analyzer.

4.1 Scheda di sviluppo

La scheda di sviluppo utilizzata è una Microchip PICDEMZ, che contiene numerose funzionalità, utili per il testing e lo sviluppo di soluzioni wireless. Questa scheda consente un'interfaccia diretta con la daughter card MRF24J40MA tramite un connettore a 12 pin. Le principali caratteristiche sono [2]:

- Socket DIP per microcontrollori a 28 e 40 pin, che consentono l'utilizzo di svariati microcontrollori Microchip PIC. Di default queste schede montano un microcontrollore PIC18LF4620 con un quarzo esterno da 4 MHz.
- Sensore di temperatura TC77, comunicante con in microcontrollore tramite il bus SPI.
- Due led D1 e D2 collegati a RA0 e RA1.
- Due pulsanti S2 e S3 connessi ai pin RB4 e RB5, rispettivamente. Non ci sono pull-up collegati esternamente, quindi sarà necessario abilitare quelli interni.
- Pulsante collegato al pin MCLR del microcontrollore.
- Interfaccia RJ-11 per la programmazione in-circuit (ICSP) tramite debugger MPLAB ICD 3.
- Connettore RS-232 a 9 pin per connessione seriale con PC.
- Connettore Samtec P/N LST-106-07-F-D a 12 pin per la daughter card.
- Alimentazione tramite jack coassiale da 2.5 mm oppure tramite. batteria (9V). Sulla scheda è montato un regolatore di tensione LP2981.
- Shunt per la misura di corrente totale assorbita dalla scheda.
- Zona di prototipazione per lo sviluppo circuitale.

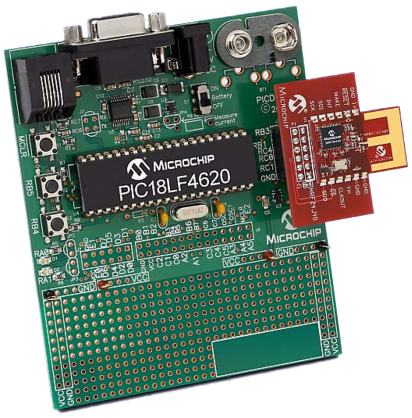

Figura 4.1: Scheda di sviluppo PICDEM Z con la daughter board MRF24J40MA.

4.1.1 Microcontrollore

Il microcontrollore utilizzato è un PIC18LF4620, della famiglia PIC18 a consumo ridotto, con architettura ad 8 bit. È costituito, oltre che da un'unità di elaborazione centrale, anche da svariate periferiche, che lo rendono un ottimo candidato per lo sviluppo di sistemi modulari, come nel nostro caso.
Le caratteristiche principali sono [3]:

- Clock fino a 40MHz
- Memoria programma di tipo flash da 64kB
- 3968 byte SRAM
- 1024 byte di memoria non volatile EEPROM
- 3 interrupt esterni
- Fino a 13 pin con funzionalità ADC a 10 bit con auto-acquisizione
- Modulo di comunicazione MSSP (SPI e I^2C)
- Modulo di comunicazione EUSART
- 3 moduli timer a 16 bit ed 1 a 8 bit
- Livelli di priorità per gli interrupt
- In-circuit Serial Programming tramite due pin

Nelle sezioni successive vengono presentate alcune delle caratteristiche che è bene considerare nell'utilizzo di a questo dispositivo. Per la lista completa delle funzionalità e delle caratteristiche si rimanda al relativo datasheet [4].

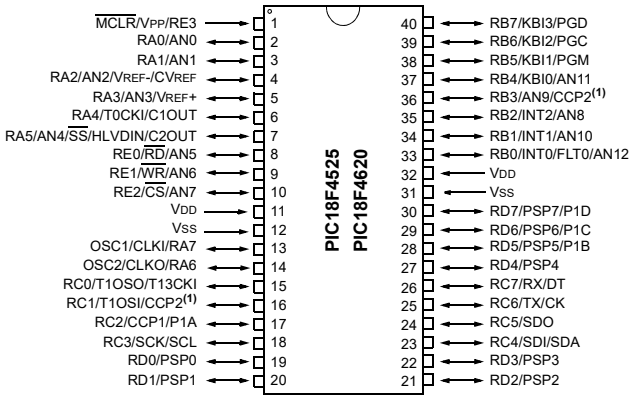

Figura 4.2: Pinout del microcontrollore PIC18LF4620.

4.1.1.1 Memoria

Nei microcontrollori della serie PIC18 sono disponibili tre aree di memoria

- Program Memory
- Data RAM
- Data EEPROM

La famiglia PIC18 implementa un program counter a 21 bit, capace di sfruttare fino a 2MB di memoria di programma che, com'è possibile vedere in Figura 4.3, è suddivisa in più sezioni. Per il dispositivo utilizzato, come precedentemente riportato, è limitata a 64kB, che consente l'utilizzo fino a 32768 istruzioni single-word.

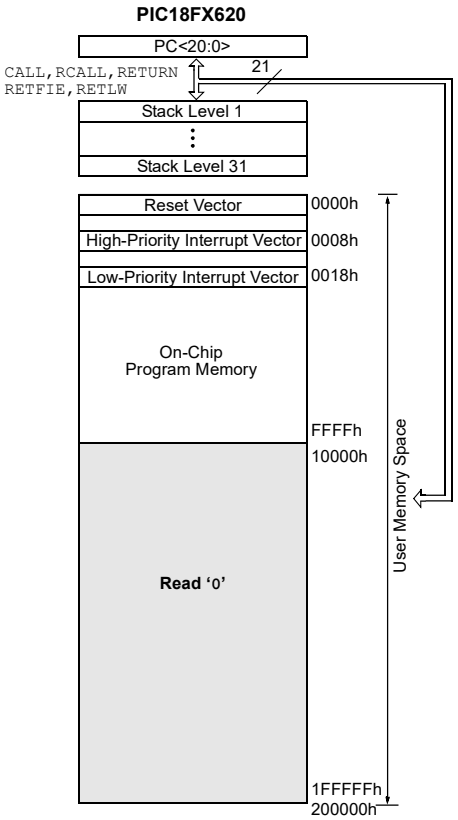

Figura 4.3: Program memory map per PIC18LF4620

La memoria dati è implementata come static RAM, in cui ogni locazione è identificata da un indirizzo a 12 bit, per una dimensione massima di 4096 byte. In questo dispositivo lo spazio è suddiviso in 16 banchi della dimensione di 256 byte ciascuno. Questa memoria contiene sia gli Special Function Registers (SFRs) sia i General Purpose Registers (GPRs). Gli SFR sono utilizzati dalla CPU per controllare il comportamento del device. Questi registri sono ubicati nel banco 15 occupando gli indirizzi tra 0xF80 e 0xFFF, e sono elencati in Figura 4.4. Possono essere classificati in due gruppi: quelli associati con il core del dispositivo (ALU, interrupts, resets) e quelli relativi alle periferiche.

Address	Name	Address	Name	Address	Name	Address	Name
FFFh	TOSU	FDFh	INDF2(1)	FBFh	CCPR1H	F9Fh	IPR1
FFEh	TOSH	FDEh	POSTINC2(1)	FBEh	CCPR1L	F9Eh	PIR1
FFDh	TOSL	FDDh	POSTDEC2(1)	FBDh	CCP1CON	F9Dh	PIE1
FFCh	STKPTR	FDCh	PREINC2(1)	FBCh	CCPR2H	F9Ch	—(2)
FFBh	PCLATU	FDBh	PLUSW2(1)	FBBh	CCPR2L	F9Bh	OSCTUNE
FFAh	PCLATH	FDAh	FSR2H	FBAh	CCP2CON	F9Ah	—(2)
FF9h	PCL	FD9h	FSR2L	FB9h	—(2)	F99h	—(2)
FF8h	TBLPTRU	FD8h	STATUS	FB8h	BAUDCON	F98h	—(2)
FF7h	TBLPTRH	FD7h	TMR0H	FB7h	PWM1CON(3)	F97h	—(2)
FF6h	TBLPTRL	FD6h	TMR0L	FB6h	ECCP1AS(3)	F96h	TRISE(3)
FF5h	TABLAT	FD5h	T0CON	FB5h	CVRCON	F95h	TRISD(3)
FF4h	PRODH	FD4h	—(2)	FB4h	CMCON	F94h	TRISC
FF3h	PRODL	FD3h	OSCCON	FB3h	TMR3H	F93h	TRISB
FF2h	INTCON	FD2h	HLVDCON	FB2h	TMR3L	F92h	TRISA
FF1h	INTCON2	FD1h	WDTCON	FB1h	T3CON	F91h	—(2)
FF0h	INTCON3	FD0h	RCON	FB0h	SPBRGH	F90h	—(2)
FEFh	INDF0(1)	FCFh	TMR1H	FAFh	SPBRG	F8Fh	—(2)
FEEh	POSTINC0(1)	FCEh	TMR1L	FAEh	RCREG	F8Eh	—(2)
FEDh	POSTDEC0(1)	FCDh	T1CON	FADh	TXREG	F8Dh	LATE(3)
FECh	PREINC0(1)	FCCh	TMR2	FACh	TXSTA	F8Ch	LATD(3)
FEBh	PLUSW0(1)	FCBh	PR2	FABh	RCSTA	F8Bh	LATC
FEAh	FSR0H	FCAh	T2CON	FAAh	EEADRH	F8Ah	LATB
FE9h	FSR0L	FC9h	SSPBUF	FA9h	EEADR	F89h	LATA
FE8h	WREG	FC8h	SSPADD	FA8h	EEDATA	F88h	—(2)
FE7h	INDF1(1)	FC7h	SSPSTAT	FA7h	EECON2(1)	F87h	—(2)
FE6h	POSTINC1(1)	FC6h	SSPCON1	FA6h	EECON1	F86h	—(2)
FE5h	POSTDEC1(1)	FC5h	SSPCON2	FA5h	—(2)	F85h	—(2)
FE4h	PREINC1(1)	FC4h	ADRESH	FA4h	—(2)	F84h	PORTE(3)
FE3h	PLUSW1(1)	FC3h	ADRESL	FA3h	—(2)	F83h	PORTD(3)
FE2h	FSR1H	FC2h	ADCON0	FA2h	IPR2	F82h	PORTC
FE1h	FSR1L	FC1h	ADCON1	FA1h	PIR2	F81h	PORTB
FE0h	BSR	FC0h	ADCON2	FA0h	PIE2	F80h	PORTA

Note 1: This is not a physical register.
2: Unimplemented registers are read as '0'.
3: This register is not available on 28-pin devices.

Figura 4.4: Special Function Registers (SFRs) per PIC18.

Per poter effettuare un rapido accesso alle locazioni di memoria viene utilizzato un Bank Select Register (BSR), un registro di selezione a 4 bit per i banchi di memoria. Sfruttando ciò l'accesso a dati contigui sullo stesso banco viene risolto in un singolo ciclo di clock, essendo necessari soltanto 8 bit di indirizzamento.
Per rendere un accesso rapido alle porzioni di memoria più comuni è stato implementato anche un Access Bank, che mappa tramite un indirizzo ad 8 bit gli indirizzi dei primi 128 byte del banco 0 e degli ultimi 128 del banco 15 (SFR).

La EEPROM, infine, è un array di memoria non volatile separato dalla RAM e dalla program memory, che consente lo storage di dati a lungo termine.

Le operazioni su questo spazio vengono effettuate tramite l'utilizzo di opportuni registri. In particolare l'accesso alla porzione di memoria è dato dalla coppia di registri `EEADRH:EEADRL`, in cui `EEADRH` contiene soltanto i due MSb dell'indirizzo e la scrittura/lettura avviene tramite il registro `EEDATA`. Con questa architettura è possibile accedere ad un range di memoria di 1024 byte.

4.1.1.2 Interrupts

Questo microcontrollore è provvisto di numerose sorgenti di interrupt ed è dotato di una funzione che consente di organizzarli in modo prioritario (alta o bassa priorità). In generale una sorgente di interrupt è caratterizzata da tre bit:

- *Flag bit*, per indicare se è avvenuto un evento di interrupt.
- *Enable bit*, per abilitare l'interrupt per un determinato modulo.
- *Priority bit*, per selezionare alta o bassa priorità.

L'opzione di priorità degli interrupt viene selezionata tramite il bit `RCON.IPEN` ed è possibile abilitare in modo separato gli interrupt ad alta o bassa priorità, rispettivamente tramite i bit `INTCON.GIEH` e `INTCON.GIEL`.

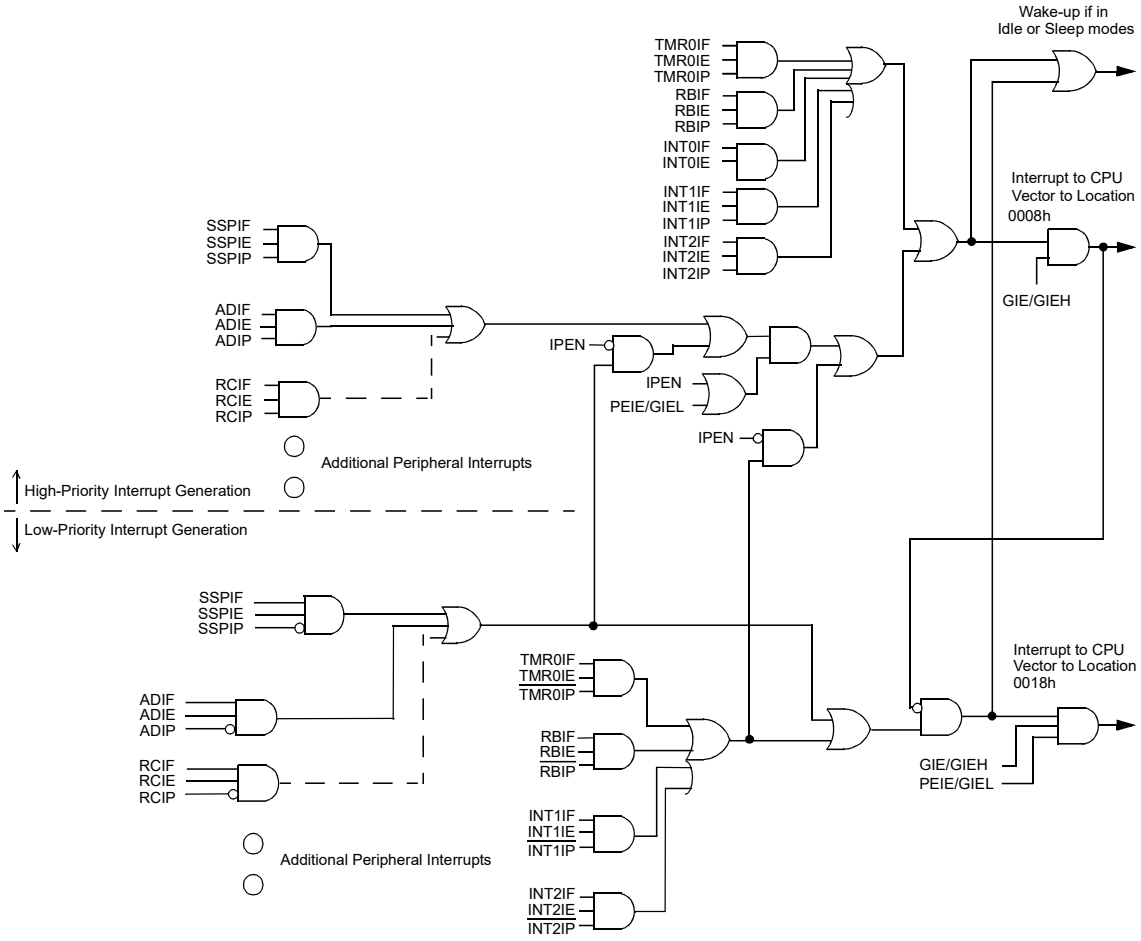

Figura 4.5: Logica di interrupt per PIC18.

Quando viene rilevato un interrupt, per evitare sovrapposizioni, vengono disabilitati tutti quelli con priorità minore o uguale. In particolare gli interrupt ad alta priorità possono bloccare quelli a precedenza più bassa. Il program counter viene immediatamente caricato

con l'indirizzo 0x0008 o 0x0018, dipendentemente dalla priorità dell'interrupt. Una volta all'interno della routine di interrupt, la sorgente può essere identificata valutando i flag dei moduli i cui interrupt sono abilitati.
I flag non sono legati all'abilitazione del corrispondente interrupt, ma sono implementati via hardware, e commutano ogniqualvolta si verifichi l'evento corrispondente. Ciò li rende utili in fase di attesa facendo polling su di essi, ad esempio, se è necessario che nessun'altra istruzione venga eseguita nel frattempo.
I registri relativi alle operazioni di interrupt sono riportati in Figura 4.6.

INTCON: INTERRUPT CONTROL REGISTER

GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF

INTCON2: INTERRUPT CONTROL REGISTER 2

$\overline{\text{RBPU}}$	INTEDG0	INTEDG1	INTEDG2	-	TMR0IP	-	RBIP

INTCON3: INTERRUPT CONTROL REGISTER 3

INT2IP	INT1IP	-	INT2IE	INT1IE	-	INT2IF	INT1IF

PIR1: PERIPHERAL INTERRUPT REQUEST (FLAG) REGISTER 1

PSPIF	ADIF	RCIF	TXIF	SSPIF	CCPIF	TMR2IF	TMR1IF

PIR2: PERIPHERAL INTERRUPT REQUEST (FLAG) REGISTER 2

OSCFIF	CMIF	-	EEIF	BCLIF	HLVDIF	TMR3IF	CCP2IF

PIE1: PERIPHERAL INTERRUPT ENABLE REGISTER 1

PSPIE	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE

PIE2: PERIPHERAL INTERRUPT ENABLE REGISTER 2

OSCFIE	CMIE	-	EEIE	BCLIE	HLVDIE	TMR3IE	CCP2IE

IPR1: PERIPHERAL INTERRUPT PRIORITY REGISTER 1

PSPIP	ADIP	RCIP	TXIP	SSPIP	CCP1IP	TMR2IP	TMR1IP

IPR2: PERIPHERAL INTERRUPT PRIORITY REGISTER 2

OSCFIP	CCMIP	-	EEIP	BCLIP	HLVDIP	TMR3IP	CCP2IP

RCON: RESET CONTROL REGISTER

IPEN	SBOREN	-	$\overline{\text{RI}}$	$\overline{\text{TO}}$	$\overline{\text{PD}}$	$\overline{\text{POR}}$	$\overline{\text{BOR}}$

Figura 4.6: Registri di interrupt per il microcontrollore.

4.1.2 Transceiver MRF24J40MA

Il transceiver utilizzato è un modulo Microchip MRF24J40MA [5], completo di tutti i componenti necessari per il funzionamento e basato sull'integrato MRF24J40, il cui clock è derivato da un oscillatore al quarzo da 20 MHz, con tolleranza di ±10 ppm a 25°C, per essere conforme alla tolleranza di ±40 ppm sul symbol rate imposta dallo standard.
La scheda integra un circuito di matching ed un'antenna pcb per operare nella banda ISM, tra 2.405 e 2.48 GHz, con una potenza massima di uscita di 0 dBm.
È conforme allo standard IEEE 802.15.4 e supporta vari protocolli, tra cui ZigBee (curato da ZigBee Alliance) e MiWi (gestito da Microchip).
L'interfaccia con il microcontrollore viene è gestita dalla comunicazione SPI a 4 fili e dai pin Wake, Reset ed Interrupt.

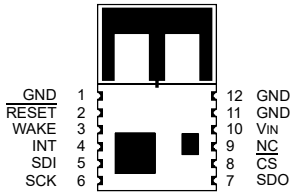

Figura 4.7: Pinout della scheda.

Il transceiver utilizzato è montato su una scheda di supporto che si interfaccia con la scheda PICDEMZ tramite un connettore a 12 pin Samtec P/N LST-106-07-F-D con un pinout compatibile con quello della motherboard riportato in Figura 4.8.

Microcontroller	Signal	Pin	Pin	Signal	Microcontroller
RB3		12	11		RB2
RB1		10	9	SCK	RC3
RC4	MISO	8	7	MOSI	RC5
RC0	$\overline{CS}$	6	5	INT	RB0
RC1	WAKE	4	3	$\overline{RESET}$	RC2
	GND	2	1	+3.3V	

Figura 4.8: Pinout del connettore J2 per la daughter board sulla scheda PICDEM Z (top view).

Lo schema a blocchi della scheda utilizzata è riportato in Figura 4.9, in cui si sono anche evidenziate le connessioni con la MCU.

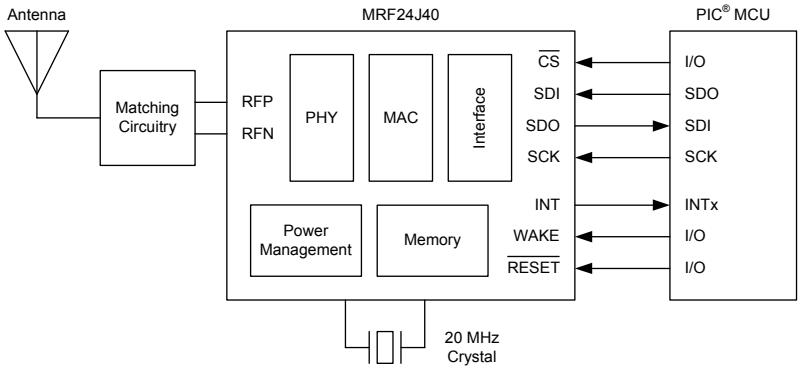

Figura 4.9: Schema a blocchi della daughter board.

4.1.2.1 MRF24J40

L'integrato Microchip MRF24J40 è il vero e proprio transceiver [4], che gestisce il sistema secondo le specifiche dettate dallo standard.
In particolare si occupa della gestione delle funzionalità di livello fisico e MAC per la creazione di una rete a basso costo e a ridotto consumo energetico.
Questo componente fornisce supporto per:

- Energy Detection
- Carrier Sense
- Tre modalità CCA
- Algoritmo CSMA-CA
- Ritrasmissione automatica
- Invio e gestione automatica degli Acknowledgment
- Buffer indipendenti per Beacon, Dati e slot GTS
- Supporto per cifratura e decodifica per il sublayer MAC

Grazie a queste caratteristiche si riesce ad alleggerire il carico computazionale e per la creazione del sistema non sono necessari componenti ad alte prestazioni.
In particolare in fase di ricezione è possibile filtrare i pacchetti non validi o destinati ad altri nodi. Le funzionalità di accesso al canale, filtraggio o la parte relativa alla sicurezza sono implementate in hardware.

4.1.2.1.1 Memoria La memoria nel MRF24J40 è implementata come RAM statica, i cui dati contenuti hanno la dimensione di 1 byte, ed accessibile attraverso la porta SPI. È suddivisa funzionalmente in registri di controllo e buffer dati (FIFO), come illustrato in Figura 4.10.

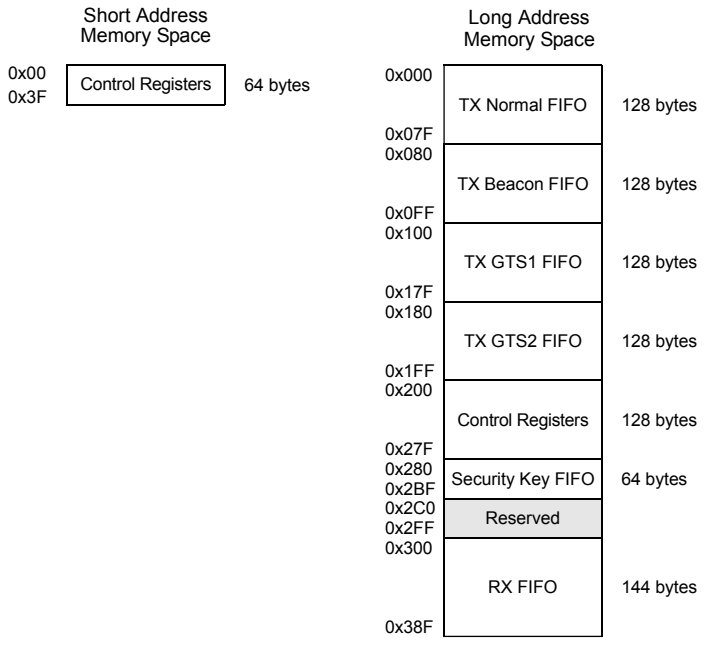

Figura 4.10: Memory map per il transceiver MRF24J40.

I registri di controllo definiscono i parametri operativi e stato del transceiver, mentre le FIFO vengono usate come buffer temporanei per la trasmissione, ricezione e per le chiavi di sicurezza.
Come si vede sono presenti due zone distinte di memoria, una contenente esclusivamente registri di controllo e l'altra, molto più grande, contenente anche i buffer FIFO. Queste due aree di memoria sono dette rispettivamente *short address memory* e *long address memory*. Per accedere ad un determinato elemento della memoria tramite la porta SPI deve essere trasmesso l'indirizzo con una ben determinata formattazione seguito dal dato che si intende trasmettere. Il modulo SPI del transceiver è impostato per un'operazione di tipo slave, per cui dal punto di vista dell'unità centrale sarà necessario, per ottenere un dato dalla daughter board, fornire un segnale di clock sulla linea SCK.

Per l'accesso alla short address memory si trasmette via SPI un primo byte composto, a partire dal MSb, da uno '0' seguito dall'indirizzo a 6 bit a cui viene accodato uno '0' o un '1' a seconda che si voglia rispettivamente leggere (Figura 4.11a) o scrivere (Figura 4.11b). Il secondo byte trasmesso conterrà quindi il valore da scrivere nella locazione selezionata.

Per la long address memory, i cui indirizzi hanno una lunghezza di 12 bit, sarà necessario inviare 3 byte. Il primo è composto da un '1' seguito dai primi 7 bit più significativi dell'indirizzo; il secondo byte è composto dai restanti 5 bit seguiti da uno '0' o da un '1', a seconda dell'operazione che si vuole eseguire, e dagli ultimi due bit, i cui valori verranno ignorati dal transceiver. Il terzo byte inviato conterrà infine il valore da scrivere nel registro. Queste operazioni sono riportate nelle figure 4.11c e 4.11d.
In entrambi i casi durante le operazioni di lettura il byte relativo ai dati viene ignorato; la risposta del transceiver all'interrogazione sarà presente sulla linea SDO del transceiver (SDI del microcontrollore).

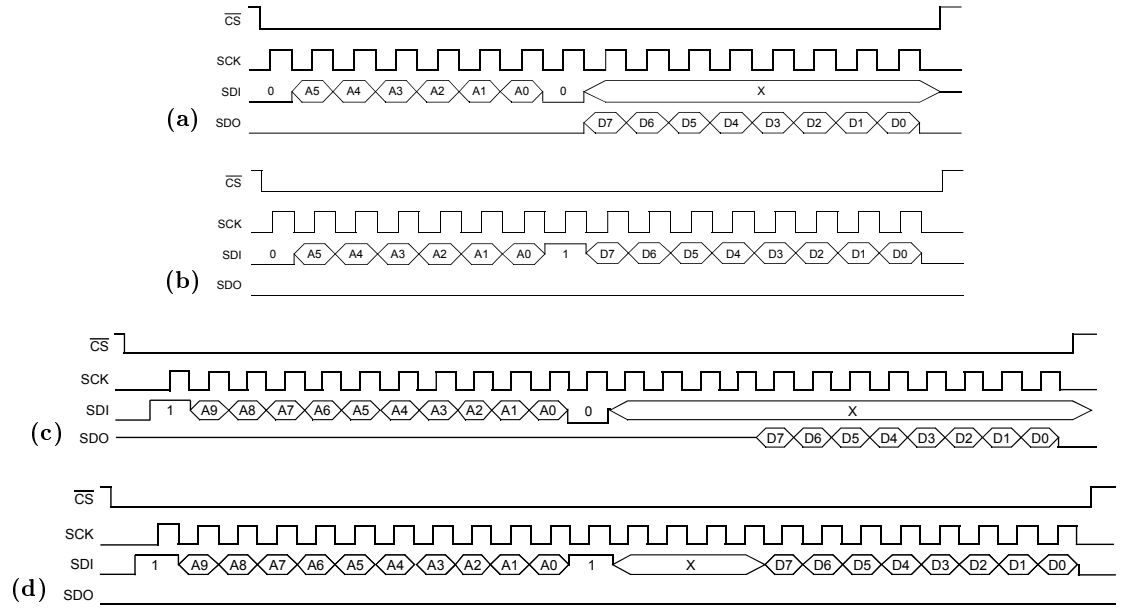

Figura 4.11: Operazioni sui registri di memoria dal punto di vista del transceiver: (a) short address read, (b) short address write, (c) long address read, (d) long address write.

In Figura 4.12 e in Figura 4.13 vengono riportati i registri di controllo nella short address memory e nella long address memory, rispettivamente.

0x00	RXMCR	0x10	ORDER	0x20	ESLOTG67	0x30	RXSR
0x01	PANIDL	0x11	TXMCR	0x21	TXPEND	0x31	INTSTAT
0x02	PANIDH	0x12	ACKTMOUT	0x22	WAKECON	0x32	INTCON
0x03	SADRL	0x13	ESLOTG1	0x23	FRMOFFSET	0x33	GPIO
0x04	SADRH	0x14	SYMTICKL	0x24	TXSTAT	0x34	TRISGPIO
0x05	EADR0	0x15	SYMTICKH	0x25	TXBCON1	0x35	SLPACK
0x06	EADR1	0x16	PACON0	0x26	GATECLK	0x36	RFCTL
0x07	EADR2	0x17	PACON1	0x27	TXTIME	0x37	SECCR2
0x08	EADR3	0x18	PACON2	0x28	HSYMTMRL	0x38	BBREG0
0x09	EADR4	0x19	Reserved	0x29	HSYMTMRH	0x39	BBREG1
0x0A	EADR5	0x1A	TXBCON0	0x2A	SOFTRST	0x3A	BBREG2
0x0B	EADR6	0x1B	TXNCON	0x2B	Reserved	0x3B	BBREG3
0x0C	EADR7	0x1C	TXG1CON	0x2C	SECCON0	0x3C	BBREG4
0x0D	RXFLUSH	0x1D	TXG2CON	0x2D	SECCON1	0x3D	Reserved
0x0E	Reserved	0x1E	ESLOTG23	0x2E	TXSTBL	0x3E	BBREG6
0x0F	Reserved	0x1F	ESLOTG45	0x2F	Reserved	0x3F	CCAEDTH

Figura 4.12: Registri di controllo nella Short Address Memory.

0x200	RFCON0	0x210	RSSI	0x220	SLPCON1	ox230	ASSOEADR0	0x240	UPNONCE0
0x201	RFCON1	0x211	SLPCON0	0x221	Reserved	0x231	ASSOEADR1	0x241	UPNONCE1
0x202	RFCON2	0x212	Reserved	0x222	WAKETIMEL	0x232	ASSOEADR2	0x242	UPNONCE2
0x203	RFCON3	0x213	Reserved	0x223	WAKETIMEH	0x233	ASSOEADR3	0x243	UPNONCE3
0x204	Reserved	0x214	Reserved	0x224	REMCNTL	0x234	ASSOEADR4	0x244	UPNONCE4
0x205	RFCON5	0x215	Reserved	0x225	REMCNTH	0x235	ASSOEADR5	0x245	UPNONCE5
0x206	RFCON6	0x216	Reserved	0x226	MAINCNT0	0x236	ASSOEADR6	0x246	UPNONCE6
0x207	RFCON7	0x217	Reserved	0x227	MAINCNT1	0x237	ASSOEADR7	0x247	UPNONCE7
0x208	RFCON8	0x218	Reserved	0x228	MAINCNT2	0x238	ASSOSADR0	0x248	UPNONCE8
0x209	SLPCAL0	0x219	Reserved	0x229	MAINCNT3	0x239	ASSOSADR1	0x249	UPNONCE9
0x20A	SLPCAL1	0x21A	Reserved	0x22A	Reserved	0x23A	Reserved	0x24A	UPNONCE10
0x20B	SLPCAL2	0x21B	Reserved	0x22B	Reserved	0x23B	Reserved	0x24B	UPNONCE11
0x20C	Reserved	0x21C	Reserved	0x22C	Reserved	0x23C	Unimplemented	0x24C	UPNONCE12
0x20D	Reserved	0x21D	Reserved	0x22D	Reserved	0x23D	Unimplemented		
0x20E	Reserved	0x21E	Reserved	0x22E	Reserved	0x23E	Unimplemented		
0x20F	RFSTATE	0x21F	Reserved	0x22F	TESTMODE	0x23F	Unimplemented		

Figura 4.13: Registri di controllo nella Long Address Memory.

4.1.2.1.2 Interrupts Il transceiver MRF24J40 è dotato un pin che notifica l'arrivo di uno tra otto eventi di interrupt (come riportato in Figura 4.14). Questi vengono abilitati tramite il registro `INTCON` ed i flag sono contenuti nel registro `INTSTAT`, che viene resettato dopo essere stato letto. Il pin di interrupt, riportato anche nella piedinatura della daughter card, continua a segnalare un interrupt finché non vengono resettati i flag. La polarità del pin `INT` è configurato tramite il bit `SLPCON0.INTEDGE`.

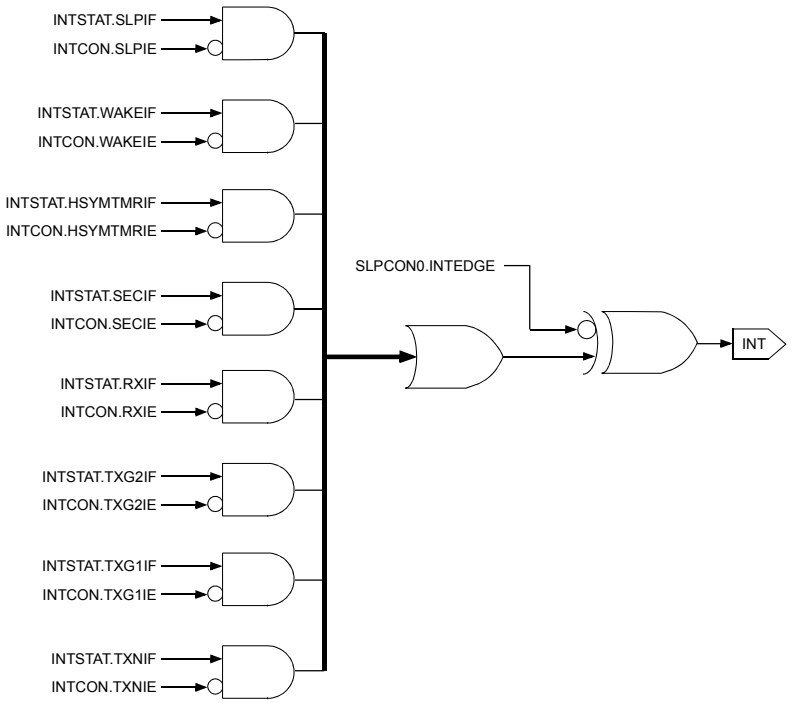

Figura 4.14: Logica di interrupt per l'MRF24J40.

Capitolo 5

Firmware

5.1 Descrizione delle funzionalità del codice

Il firmware è stato organizzato modularmente sotto forma di librerie, per cui ogni modulo è pensato e configurato singolarmente, per consentirne il riutilizzo al di fuori della specifica applicazione.

All'interno del progetto finale i file consistono generalmente in un header (`*.h`), contente le dichiarazioni delle funzioni, ed un file di implementazione (`*.c`), in cui sono riportate le relative definizioni, i due hanno la stessa radice.
Il progetto è così composto (se sono presenti sia file `.h` sia `.c` viene riportata l'estensione come `.*`):

- File di configurazione

  `config.h` : configurazione dei parametri principali del sistema

  `hardware_config.h` : contiene la configurazione dell'hardware utilizzato, quali la mappatura delle porte, la frequenza dell'oscillatore, ...

  `p18_init.*` : inizializzazione del microcontrollore

- Gestione delle periferiche di I/O

  `SPI_handler.*` : gestisce l'interfaccia SPI con la scheda del transceiver, contiene tutte le funzioni di lettura/scrittura

  `UART_handler.*` : gestisce la comunicazione tramite la porta seriale

- Gestione del transceiver

  `mrf24j40.h` : definisce gli indirizzi dei registri del transceiver e alcune funzioni elementari

  `shared_vars.h` : variabili condivise tra più moduli

- Definizione dei pacchetti

  `frame_ctrl_defs.h` : definisce le strutture base per la costruzione dei frame

  `data_frame.h` : contiene le definizioni per il data frame

  `beacon_frame.h` : contiene le definizioni per il beacon frame

  `MAC_cmd_frame.h` : contiene le definizioni per il mac command frame

- Gestione dei pacchetti

  `beacon_handler.*` : creazione e della decodifica di frame beacon

`command_handler.*` : generazione e decodifica frame di tipo MAC command

`data_handler.*` : genera e decodifica frame data

- Gestione del nodo

`buffer_handler.*` : contiene le definizioni dei buffer

`node_handler.*` : si occupa della gestione del nodo, e contiene le istruzioni di connessione/disconnessione

- Programma principale

`main.c` : è il programma principale, contiene il loop in cui vengono eseguite le istruzioni e la funzione `ISR`

Il codice è organizzato per fornire delle funzioni per la gestione della rete e dei nodi, incluse quelle di configurazione dei nodi, coordinatore o device, di gestione delle connessioni, di richiesta e fornitura dati.

Oltre al buffer FIFO contenuto nel transceiver sono stati previsti anche dei buffer locali, allo scopo di occupare per il minor tempo possibile la daughter card. Durante le operazioni di lettura è necessario spegnere l'interfaccia radio per evitare l'accesso contemporaneo al buffer da parte di MCU e transceiver; la probabilità di perdere dei pacchetti, inoltre, è tanto più alta quanto più lungo è l'intervallo di spegnimento del ricevitore, allo scopo di inibire la ricezione di pacchetti.

Considerando l'hardware in uso, bisogna prestare attenzione alla programmazione, vista l'esigua quantità di memoria disponibile. Viene evitato l'utilizzo di variabili costanti (`const`) in favore delle macro del preprocessore (`#define`). Nella dichiarazione delle variabili si sfruttano le definizioni presenti nella libreria `stdint.h`, che consentono un controllo diretto sulla dimensione.

Per accedere a gruppi di bit all'interno delle variabili si ricorre ai cosiddetti *bit-fields*, tipici della programmazione C/C++. L'utilizzo di questi costrutti viene utilizzato nel codice semplicemente per renderlo più leggibile; bisogna prestare attenzione perché queste strutture sono fortemente dipendenti dal compilatore. Nel nostro caso (Microchip XC8) l'assegnazione dei bit fields viene organizzata a partire dal bit meno significativo [6], quindi deve essere riportato per primo il campo meno significativo.

Nella definizione delle funzioni vengono utilizzati in modo consistente i passaggi per puntatore i quali, vista l'architettura del microcontrollore, risultano convenienti per variabili dalla dimensione strettamente maggiore di un byte.

Per quanto riguarda le singole periferiche, in fase di sviluppo sono stati realizzati singolarmente dei programmi di test per tracciarne le caratteristiche, al fine di ottenere il comportamento desiderato.
In particolare sono stati implementati programmi i cui obiettivi principali sono

- Comunicazione SPI
- Comunicazione RS232
- Scrittura nei registri del transceiver
- Interrupt esterni del microcontrollore
- Conversione ADC/DAC
- Costruzione del pacchetto
- Ping di due nodi con connessione diretta

5.2 Configurazione generale

La configurazione generale dei parametri della rete è contenuta nei due file `config.h` e `hardware_config.h`.
Il primo contiene i campi di indirizzo relativi al nodo

```
// PAN CONFIG
#define PAN_COORDINATOR
#define NONBEACON_MODE

#define PAN_ID              0xDE1A
#define EXTENDED_ADDRESS_H  0xAA00DE1A
#define EXTENDED_ADDRESS_L  0x00000000
#ifdef PAN_COORDINATOR
  #define SHORT_ADDRESS       0xDA00
#endif

#define MRF_TX_POWER_RANGE  0x00       // select transmission power - DS39776C-page 64
#define MRF_TX_POWER_FINE   0x00
```

`hardware_config.h` include le definizioni relative all'hardware in uso

```
// DEBUG VERSION
#define VERSION_DEBUG

// Specify quartz oscillation frequency (hertz)
#define XTAL_FREQ   4000000

// Specify clock type (uncomment to use non-PLL clock)
#define CLOCK_TYPE_PLL

// Initialize clock frequency
#ifdef CLOCK_TYPE_PLL
  #define _XTAL_FREQ (4*XTAL_FREQ)
#else
  #define _XTAL_FREQ  XTAL_FREQ
#endif

// SPI pin configuration
// define slave select pin
#define _CS PORTCbits.RC0

#define BUTTON_PRESSED 0
#define BUTTON_NOT_PRESSED 1
```

In particolare il parametro `_XTAL_FREQ` [6] è un parametro necessario per poter utilizzare la funzione `__delay_us` contenuta nelle definizioni del compilatore (`xc.h`).

5.3 Moduli del microcontrollore

Questa sezione si occupa di presentare le funzioni per le inizializzazioni, per l'interfacciamento con i moduli interni al microcontrollore, e di alcune tecniche utilizzate per realizzarle.

5.3.1 Inizializzazione del microcontrollore

Per definire il comportamento del microcontrollore la prima operazione da fare è la programmazione dei cosiddetti *configuration bits*, che definiscono i parametri basilari del sistema, come la sorgente di clock o la presenza di un watchdog.
Questa configurazione è inclusa nel file `p18_init.h`, i significati dei singoli bit sono contenuti nel datasheet [1]:

```
// CONFIG1H
#pragma config OSC = HSPLL      // Oscillator Selection bits
#pragma config FCMEN = OFF      // Fail-Safe Clock Monitor Enable bit
#pragma config IESO = OFF       // Internal/External Oscillator Switchover bit

// CONFIG2L
#pragma config PWRT = OFF       // Power-up Timer Enable bit
#pragma config BOREN = SBORDIS  // Brown-out Reset Enable bits
#pragma config BORV = 3         // Brown Out Reset Voltage bits

// CONFIG2H
#pragma config WDT = OFF        // Watchdog Timer Enable bit
#pragma config WDTPS = 256      // Watchdog Timer Postscale Select bits

// CONFIG3H
#pragma config CCP2MX = PORTC   // CCP2 MUX bit
#pragma config PBADEN = ON      // PORTB A/D Enable bit
#pragma config LPT1OSC = OFF    // Low-Power Timer1 Oscillator Enable bit
#pragma config MCLRE = ON       // MCLR Pin Enable bit

// CONFIG4L
#pragma config STVREN = ON      // Stack Full/Underflow Reset Enable bit
#pragma config LVP = OFF        // Single-Supply ICSP Enable bit
#pragma config XINST = OFF      // Extended Instruction Set Enable bit

// CONFIG5L
#pragma config CP0 = OFF        // Code Protection bit
#pragma config CP1 = OFF        // Code Protection bit
#pragma config CP2 = OFF        // Code Protection bit
#pragma config CP3 = OFF        // Code Protection bit

// CONFIG5H
#pragma config CPB = OFF        // Boot Block Code Protection bit
#pragma config CPD = OFF        // Data EEPROM Code Protection bit

// CONFIG6L
#pragma config WRT0 = OFF       // Write Protection bit
#pragma config WRT1 = OFF       // Write Protection bit
#pragma config WRT2 = OFF       // Write Protection bit
#pragma config WRT3 = OFF       // Write Protection bit
```

[1] PIC18F2525/2620/4525/4620 Data Sheet, DS39626E, pagg. 250 – 256

```
// CONFIG6H
#pragma config WRTC = OFF       // Configuration Register Write Protection bit
#pragma config WRTB = OFF       // Boot Block Write Protection bit
#pragma config WRTD = OFF       // Data EEPROM Write Protection bit

// CONFIG7L
#pragma config EBTR0 = OFF      // Table Read Protection bit
#pragma config EBTR1 = OFF      // Table Read Protection bit
#pragma config EBTR2 = OFF      // Table Read Protection bit
#pragma config EBTR3 = OFF      // Table Read Protection bit

// CONFIG7H
#pragma config EBTRB = OFF      // Boot Block Table Read Protection bit
```

5.3.2 Interrupt Service Routine

Per la gestione degli interrupt all'interno del programma principale è necessario dichiarare, oltre al main, una funzione che viene richiamata ogniqualvolta ne si presenti uno. La sintassi corretta utilizzata per la definizione della routine di interrupt è definita nel manuale operativo del compilatore XC8 [2].
Nel caso in cui vengano utilizzati gli interrupt prioritari[3] e si vogliano suddividere le due tipologie di eventi, la sintassi rispettivamente per bassa ed alta priorità diventerebbe:

```
void interrupt low_priority ISR_lp(void)
{
  ...
}

void interrupt high_priority ISR_hp(void)
{
  ...
}
```

Nel caso l'indicatore di priorità venga omesso, la routine di interrupt sarà quella relativa agli eventi ad alta priorità.
Contrariamente a quanto avviene per le normali funzioni, quelle di interrupt non devono venir chiamate nel codice principale e devono essere presenti al massimo una per ogni grado di priorità (per sistemi che sfruttano questo compilatore). Per tale motivo al loro interno è necessario un controllo dei flag di tutte le sorgenti di interrupt abilitate per poterne tracciare la provenienza.
Ad esempio se si considerano gli eventi generati da `TMR0`:

```
void interrupt ISR(void)
{
  ...
  if(INTCONbits.T0IE && INTCONbits.TMR0IF)
  {
    INTCONbits.TMR0IF = 0;
    ...
  }
  ...
}
```

[2] MPLAB XC8 C Compiler User's Guide, DS50002053F, pag. 202.
[3] inserire riferimento

Una volta identificata la sorgente è necessario azzerare il flag corrispondente.
Per quanto già riportato nel paragrafo[4] è buona pratica che l'interrupt service routine sia più snella possibile, limitandosi ad operazioni brevi. Nel progetto finale la funzione `ISR` viene implementata in `main.c`.

5.3.3 Interfacce di comunicazione

Il microcontrollore utilizza due moduli di comunicazione seriali: MSSP (Master Synchronous Serial Port), che può essere utilizzato in modalita SPI o I^2C, ed EUSART (Enhanced Universal Synchronous Asynchronous Receiver Transmitter), che gestisce la comune comunicazione seriale usata anche dall'interfaccia RS-232.

5.3.3.1 Interfaccia SPI

L'interfaccia SPI è un'interfaccia master-slave, tipicamente implementata con 3 o 4 connessioni, fra cui due linee dati (MOSI, *Master Output Slave Input* e MISO, *Master Input Slave Output*) anche denominate SDO e SDI, da una linea di clock SCK e da una di *Slave Select*, SS (o *Chip Select*, CS) opzionale.

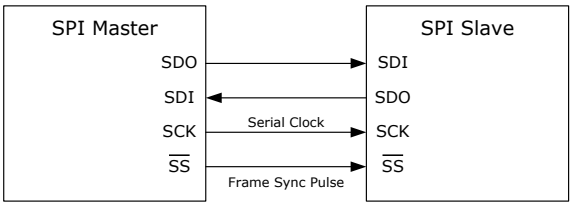

Figura 5.1: Schema di connessione dei dispositivi tramite SPI.

Questo sistema consente di realizzare una comunicazione full duplex temporizzata, essendo di tipo sincrono, dai fronti dell'onda trasmessa dal master presenti sulla linea SCK.
La modalità operativa dell'SPI può essere rappresentata da due parametri, rappresentati in Tabella 5.1.

SPI Mode	Clock Idle Polarity	Clock Edge
0,0	L	↑
0,1	L	↓
1,0	H	↑
1,1	H	↓

Tabella 5.1: Modalità operativa dell'SPI.

In cui clock edge è il fronte di campionamento del dato sulla linea di ingresso (il dato in uscita è sincronizzato sul fronte precedente). Un esempio di comunicazione in modalità (0,0) è illustrata in Figura 5.2.
Nel microcontrollore i registri contenenti i parametri di configurazione dedicati all'interfaccia SPI sono contenuti nei registri `SSPSTAT` e `SSPCON1`, a cui si aggiunge il buffer ad 8 bit `SSPBUF`. Lo schema funzionale del modulo MSSP contenuto nella MCU è rappresentato in Figura 5.3.

Si sottolinea in particolare che lo shift register `SSPREG`, che si occupa di portare in uscita il dato sulla linea SDO, non è accessibile via software ma si interfaccia con l'utente tramite un secondo registro, `SSPBUF`, su cui vengono scritti e ricevuti i valori via software.

[4] inserire riferimento

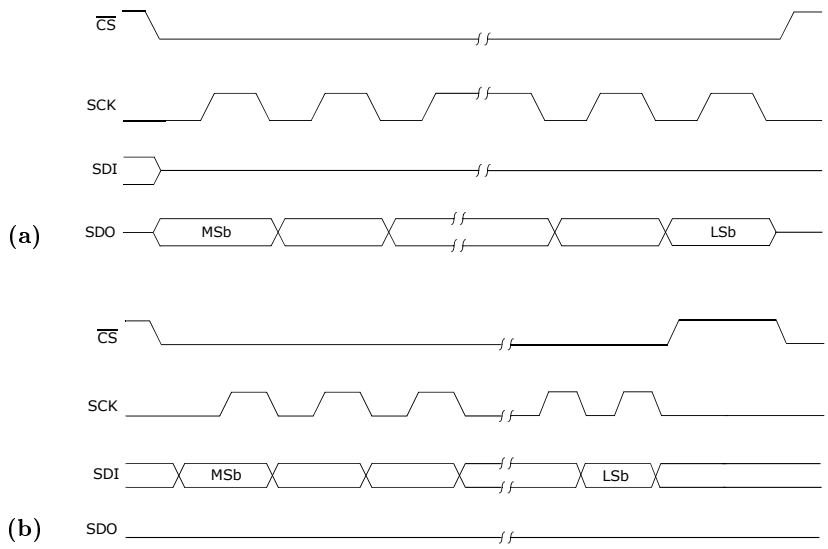

Figura 5.2: Metodo di comunicazione SPI modalità (0,0). (a) SPI output, (b) SPI input.

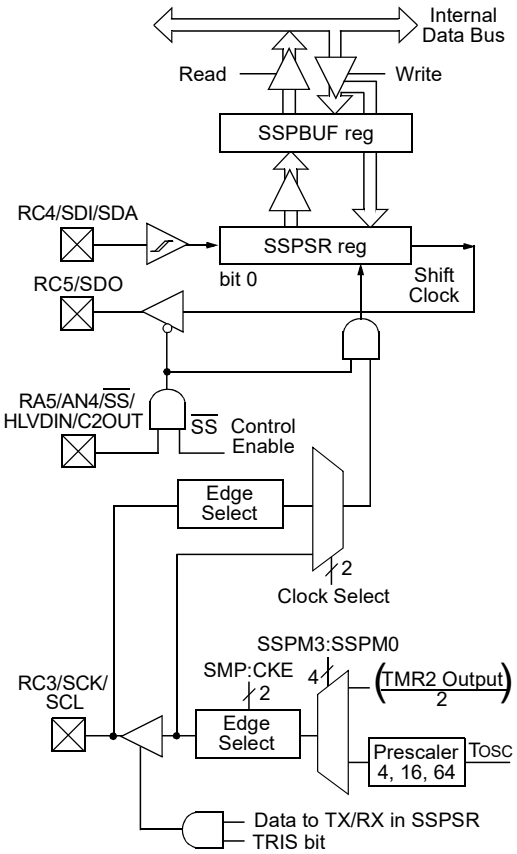

Figura 5.3: Diagramma a blocchi del modulo MSSP ad uso SPI. I pin di I/O indicati sono relativi al microcontrollore, vedi Figura 4.2.

Nel nostro caso la comunicazione avviene con la modalità (0,0) e deve inviare gruppi di 2 o 3 byte a seconda dell'area di memoria a cui si fa riferimento. Nella funzione `SPI_init`, implementata nel file `SPI_handler.c`, vengono impostati i parametri per il funzionamento desiderato.

```
1  void SPI_init()
2  {
3    TRISCbits.TRISC3 = 0;
4    SSPSTATbits.SMP = 1;                // 1->master, 0->slave
5
6    // setup spi mode 0
7    SSPCON1bits.CKP = 0;                // clock edge
8    SSPSTATbits.CKE = 1;                // clock polarity
9
10   SSPCON1bits.SSPM = 0x0;             // 0000 = SPI Master, clock = FOSC/4
11
12   SSPCON1bits.SSPEN = 1;
13
14   PIE1bits.SSPIE = 0;
15   IPR1bits.SSPIP = 0;
16   PIR1bits.SSPIF = 0;
17
18   _CS = 1;
19   return;
20 }
```

dove alle righe 14 – 16 sono settati i parametri relativi ai registri di interrupt.

In particolare per dialogare con la daughter board è necessario configurare il microcontrollore come master e per ottenere un dialogo sufficientemente veloce tra le due schede si impone la massima velocità possibile sulla linea SCK ($F_{OSC}/4$), considerata la lunghezza molto ridotta del bus, tramite la scelta di `SSPM`.
In seguito ad un processo di ottimizzazione si è arrivati alla configurazione attuale per la temporizzazione del modulo SPI: l'interrupt corrispondente viene disabilitato e si monitora il flag di interrupt in modo continuo (polling), che commuta una volta terminata la trasmissione del byte trasmesso. In fase di inizializzazione si porta la linea di chip select al livello logico alto (riga 18), il cui simbolo (`_CS`) viene definito nel file `hardware_config.h` associandolo al corrispondente pin del microcontrollore.

Per inviare un byte basta semplicemente abbassare la linea di chip select e caricarlo in `SSPBUF`. Al termine della scrittura comincia la trasmissione secondo le specifiche riportate nei registri di configurazione. Al termine dell'operazione il flag `SSPIF`, contenuto nel registro `PIR1`, commuta e si porta a `1`. Il registro `SSPBUF` è implementato utilizzando uno shift register, quindi al termine dell'operazione conterrà un eventuale dato ricevuto sulla linea in ingresso SDI, come riportato nella sezione [5] .
Il codice corrispondente è il seguente:

```
_CS = 0;
PIR1bits.SSPIF = 0;
uint8_t data;
SSPBUF = data;
while(SSPSTATbits.BF==0);
data = SSPBUF;
_CS = 1;
```

Per chiudere la comunicazione si riporta la linea di chip select al livello logico alto ed al termine delle istruzioni in `data` sarà contenuto il byte trasmesso dallo slave.

[5] inserire riferimento

Per quanto riguarda la ricezione, quando la trasmissione termina viene alzato il flag `SSPIF` ma il dato non è ancora accessibile poiché necessita di essere spostato nel buffer `SSPBUF`. Il dato è pronto per la lettura soltanto quando `SSPSTAT.BF=1`, che notifica l'avvenuto trasferimento. Poiché non è sempre necessario ricevere un dato, per l'invio è sufficiente l'attesa del solo flag di interrupt.

Poiché alla daughter board si deve comunicare un indirizzo formattato nel modo riportato alla sezione[6] , per diminuire la latenza sull'invio del comando si definiscono i comandi direttamente con la sintassi utilizzata per la scrittura sui registri del transceiver.
Ad esempio il registro `RFSTATE`, di indirizzo `0x20F`, viene definito nel file `mrf24j40.h` come:

```
#define RFSTATE       0xC1F0
```

Poiché l'operazione più comune sui registri di controllo è quella di scrittura, tali indirizzi sono preformattati per questo tipo di operazione. Per leggerne il dato contenuto la funzione di lettura deve mascherare, azzerandolo, il bit relativo al comando di scrittura sul registro, come visto in Figura 4.11c.

Per le operazioni su registri con long address è stata scritta funzione `SPI_read_long`, mentre `SPI_write_short` è quella relativa alla stessa azione su un registro con short address.

```
1  inline void SPI_write_long (uint16_t long_address,
2                              uint8_t data)
3  {
4  // long_address = 1AAAAAAA AAA10000 DDDDDDDD
5    _CS = 0;
6    PIR1bits.SSPIF = 0;
7
8    SSPBUF = long_address >> 8;
9    while(PIR1bits.SSPIF == 0);
10   PIR1bits.SSPIF = 0;
11
12   SSPBUF = (uint8_t)(long_address);
13   while(PIR1bits.SSPIF == 0);
14   PIR1bits.SSPIF = 0;
15
16   SSPBUF = data;
17   while(PIR1bits.SSPIF == 0);
18   PIR1bits.SSPIF = 0;
19
20   _CS = 1;
21   return;
22 }
```

```
1  inline uint8_t SPI_read_short(uint8_t short_address)
2  {
3    _CS = 0;
4    PIR1bits.SSPIF = 0;
5
6    SSPBUF = (short_address << 1) & 0x7E;
7    while(PIR1bits.SSPIF == 0);
8    PIR1bits.SSPIF = 0;
9
10   SSPBUF = 0x00;
11   uint8_t data;
```

[6] inserire riferimento

```
12    while(SSPSTATbits.BF==0);
13    data = SSPBUF;
14
15    _CS = 1;
16    return data;
17  }
```

Un ulteriore problema è quello del riempimento del buffer per la trasmissione dati, temporalmente dispendioso utilizzando le funzioni appena introdotte. Per limitare il problema si è creata una funzione che tramite un unico ciclo copia tutto il buffer nel transceiver fino alla dimensione specificata, risparmiando tempo sulle chiamate a funzione. Questo effetto verrà analizzato meglio nella sezione [7] . L'incremento dell'indirizzo è `0x20`, poiché viene usato l'indirizzo preformattato, come nei registri di controllo.

```
1   void fifoTX_put(uint8_t* data, uint8_t n_bytes,
2                   uint16_t write_fifo_first_address)
3   {
4     uint8_t byte_idx;
5     for(byte_idx = 0; byte_idx < n_bytes;
6           byte_idx++, write_fifo_first_address += 0x20)
7     {
8     // long_address = 1AAAAAAA AAA10000 DDDDDDDD
9       PIR1bits.SSPIF = 0;
10      _CS = 0;
11
12      SSPBUF = write_fifo_first_address >> 8;
13      while(PIR1bits.SSPIF == 0);
14      PIR1bits.SSPIF = 0;
15
16      SSPBUF = (uint8_t)(write_fifo_first_address);
17      while(PIR1bits.SSPIF == 0);
18      PIR1bits.SSPIF = 0;
19
20      // data transmission
21      SSPBUF = *(data+byte_idx);
22      while(PIR1bits.SSPIF == 0);
23      PIR1bits.SSPIF = 0;
24      _CS = 1;
25    }
26    return;
27  }
```

Allo stesso modo, per leggere il buffer viene utilizzata una funzione perfettamente analoga in cui l'unica differenza sta, come detto prima, di attendere il dato sfruttando il flag `SSPSTAT.BF`.

[7] inserire riferimento

Le funzioni disponibili per l'utilizzo dell'SPI sono contenute 1in `SPI_handler.h`.

```
#define _CS PORTCbits.RC0

void SPI_init();

uint8_t SPI_read_short(uint8_t short_address);
void SPI_write_short(uint8_t short_address, uint8_t data);

uint8_t SPI_read_long(uint16_t long_address);
inline void SPI_write_long(uint16_t address, uint8_t data);

void SPI_write_16(uint16_t data);
```

5.3.3.2 Interfaccia seriale (USART)

L'interfaccia USART viene utilizzata nella comunicazione tra il nodo centrale ed un computer tramite l'interfaccia RS-232 allo scopo di raccogliere i dati e poterli processare in un secondo momento. Questa periferica si rivela molto utile in fase di debug, soprattutto se associata ad un semplice terminale di tipo ASCII.
Essendo la comunicazione asincrona si deve inizialmente decidere il baud rate, corrispondente al bitrate in una comunicazione binaria, il che equivale alla frequenza di campionamento/generazione del byte scambiato.

All'interno delle librerie utilizzate per sviluppare il software per l'acquisizione dati (Qt) la massima velocità di comunicazione della porta seriale è pari a 115.2 kbaud/s, che viene quindi adottata come velocità di trasmissione vista anche la breve distanza tra scheda e computer.
Il tempo di trasmissione del singolo byte è dato da:

$$T_B = \frac{8}{BR} = 69.4\,\mu s \tag{5.1}$$

in cui con BR si indica il baud rate.
Supponendo di voler trasferire un pacchetto composto da un timestamp a 32 bit, da un dato a 16 bit e da un'informazione sul nodo di altri 32 bit, il tempo di trasmissione sfruttando questa configurazione richiederebbe circa $700\,\mu s$. Ciò pone un primo limite allo streaming di dati su computer tramite questo tipo di interfaccia, pari a 1.4 kHz.

Per poter configurare il modulo EUSART è necessario settare, oltre che i registri riguardanti la tipologia di trasmissione, anche il Baud Rate Generator (BRG), necessario per la corretta interpretazione e trasmissione dei dati.
Il BRG consiste in un timer impostabile a 8 o 16 bit (con il bit `BRG16`), che può essere settato in modalità low o high baud rate tramite il `BRGH`, relativi al registro `TXSTA`. Per utilizzare un determinato baud rate è necessario impostare il relativo contatore, contenuto nei registri `SPBRGH` e `SPBRGL`.
Le formule per il contatore associato al baud rate generator è contenuta in Tabella 5.2, al variare dei bit `BRG16` e `BRGH`, dalle quali si ricava la formula per il baud rate generator

$$n = [\texttt{SPBRGH} : \texttt{SPBRGL}] = round\left(\frac{F_{OSC}}{k\,BR} - 1\right) \tag{5.2}$$

in cui $k = 4, 16, 64$.
Utilizzando un Baud Rate di 115200, ed avendo $F_{OSC} = 4\,\text{MHz}$, si sfrutta il timer a 16 bit e la modalità ad alto baud rate. Si ottiene

$$n = 8$$

associato ad un errore percentuale di

$$\epsilon = 3.55\%$$

Configuration Bits		BRG/EUSART Mode	Baud Rate Formula
BRG16	BRGH		
0	0	8-bit/Asynchronous	$F_{OSC} / [64\,(n+1)]$
0	1	8-bit/Asynchronous	$F_{OSC} / [16\,(n+1)]$
1	0	8-bit/Asynchronous	$F_{OSC} / [16\,(n+1)]$
1	1	8-bit/Asynchronous	$F_{OSC} / [4\,(n+1)]$

Tabella 5.2: Formule per il baud rate, F_{OSC} è la frequenza dell'oscillatore ed n il valore di `BRG`.

In `UART_handler.h` sono dichiarate le funzioni relative a questo modulo:

```
#define UART_BAUD_RATE 115200

void UART_init();
void UART_send_data(uint8_t* data, uint8_t data_length);
void UART_send_hex(uint8_t ascii_char);
void UART_send_data_16(uint16_t* data, uint16_t data_length);
void UART_send_dec(uint8_t send_number);
void UART_send_number(uint8_t single_char);
void UART_send_string(uint8_t* string);
void UART_send_char(uint8_t ascii_char);
```

L'inizializzazione del modulo EUSART viene fatta sfruttando il seguente codice:

```
1  void UART_init()
2  {
3    TRISCbits.TRISC6 = 1;
4    TRISCbits.TRISC7 = 1;
5
6    TXSTAbits.TX9 = 0;  // 8-bit transmission
7    TXSTAbits.TXEN = 1;
8    TXSTAbits.TX9D = 0; // Unused 9th bit
9
10   RCSTAbits.SPEN = 1;
11   RCSTAbits.RX9 = 0;  // 8-bit reception
12   RCSTAbits.CREN = 1; // enable receiver
13
14   BAUDCONbits.RXDTP = 0;
15   BAUDCONbits.TXCKP = 0;  // Idle low
16
17   // Baud rate settings
18   TXSTAbits.SYNC = 0;
19   TXSTAbits.BRGH = 1; // High speed baud rate (verify)
20   BAUDCONbits.BRG16 = 1;  // 8-bit baud rate generator
21
22   // Desired baud rate
23   SPBRG = round(_XTAL_FREQ/(4.*UART_BAUD_RATE)-1);
24
25   // Sync break transmission completed
```

```
26    TXSTAbits.SENDB = 0;
27    INTCONbits.GIE  = 1;
28    PIE1bits.RCIE = 1;
29    PIR1bits.RCIF = 0;
30
31    IPR1bits.RCIP = 1;
32    IPR1bits.TXIP = 1;
33  }
```

Mentre la trasmissione del singolo byte (non formattato) viene eseguita tramite la funzione `UART_send_char`:

```
1   void UART_send_char(uint8_t ascii_char)
2   {
3     TXREG = ascii_char;
4     while( !TXSTAbits.TRMT );
5
6     return;
7   }
```

Nelle versioni di debugging è utile inviare il dato preformattato per essere visualizzato su un terminale ascii

```
1   void UART_send_hex(uint8_t hex_num)
2   {
3     TXSTAbits.TXEN = 1;
4
5     TXREG = '0';
6     while( !TXSTAbits.TRMT );
7     TXREG = 'x';
8     while( !TXSTAbits.TRMT );
9
10    uint8_t single_char = hex_num >> 4;
11    UART_send_number(single_char);
12
13    single_char = hex_num & 0x0f;
14    UART_send_number(single_char);
15
16    TXREG = ' ';
17    while( !TXSTAbits.TRMT );
18
19    return;
20  }
```

5.3.4 Dispositivi di I/O

Oltre alle porte di I/O digitale è presente anche un modulo analogico di input, utilizzato per la creazione di un campione da trasferire attraverso la rete. Non essendo importante il dato in sè, visto che ha soltanto una funzione indicativa per la trasmissione dati, non si è caratterizzato il sistema di acquisizione né quello di riproduzione.

5.3.4.1 Convertitore A/D

Il convertitore A/D contenuto all'interno del microcontrollore è di tipo SAR a 10 bit e consiste in 13 canali con multiplexer.
Di seguito viene riportato il codice relativo all'inizializzazione del modulo di conversione A/D. Innanzitutto si impostano i riferimenti tramite i campi `VCFG1` e `VCFG0` del registro

`ADCON1`, quindi si configurano i pin di I/O analogico, clock e tempo di acquisizione.
Il clock del campionamento viene derivato dall'oscillatore principale e fornisce il valore del periodo T_{AD}; il tempo di campionamento viene poi definito come un numero intero di T_{AD}.

Per cominciare l'acquisizione si seleziona il canale che si vuole campionare tramite il campo `ADCON0.CSH` e si avvia la conversione impostando `ADCON0.GO_DONE = 1`. La fine del campionamento viene notificata commutando il flag di interrupt `ADIF`. È importante evidenziare che il campionamento non avviene ciclicamente in modo automatico, ma deve essere

```
//  Initialize client ADC (using AN6, RE1)
  ADCON1bits.VCFG1 = 0;
  ADCON1bits.VCFG0 = 0;
  ADCON2bits.ADCS = 0b001;    // AD conversion clock
  ADCON2bits.ACQT = 0b100;    // AD acquisition time
  ADCON1bits.PCFG = 0b1000;   // Enable AN6 to AN0 to be analog pins

  ADCON0bits.CHS = 0x06;      // channel select

  // enable adc module
  ADCON0bits.ADON = 1;
  ADCON2bits.ADFM = 1;

  // enable adc interrupts
  IPR1bits.ADIP = 1;
  PIR1bits.ADIF = 0;
  PIE1bits.ADIE = 1;
```

Nella routine di interrupt `ISR`, definita in `main.c`, verrà quindi azzerato il flag e letto il dato ricevuto. Quando si vuole ripetere l'acquisizione basterà impostare ancora `ADCON0.GO_DONE=1`. Le fasi di conversione sono riportate in Figura 5.4.

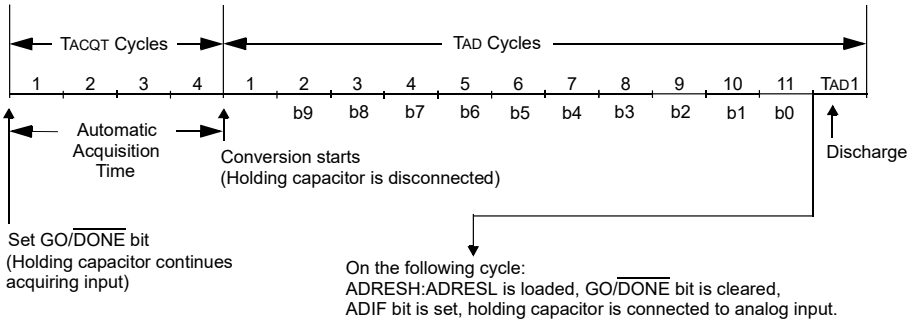

Figura 5.4: Temporizzazione della conversione per il modulo ADC.

5.3.4.2 Convertitore D/A

L'operazione di ricostruzione, inversa al campionamento, viene svolta utilizzando un convertitore D/A esterno. Nello specifico viene usato un MCP4812, DAC a 10 bit comunicante tramite interfaccia SPI.
Il dato da convertire viene ricevuto, assieme ad alcune impostazioni per la riproduzione, tramite un comando di lunghezza pari 16 bit, come rappresentato in Figura 5.5.

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

Figura 5.5: Comando di scrittura per MCP4812.

Per avviare la riproduzione del dato appena ricevuto bisogna pilotare il pin $\overline{LDAC}$: una volta finito di trasmettere il comando il fronte negativo applicato a questo pin fa cominciare la conversione, come riportato in Figura 5.6. La durata minima dell'impulso su $\overline{LDAC}$ è di 100 ns.

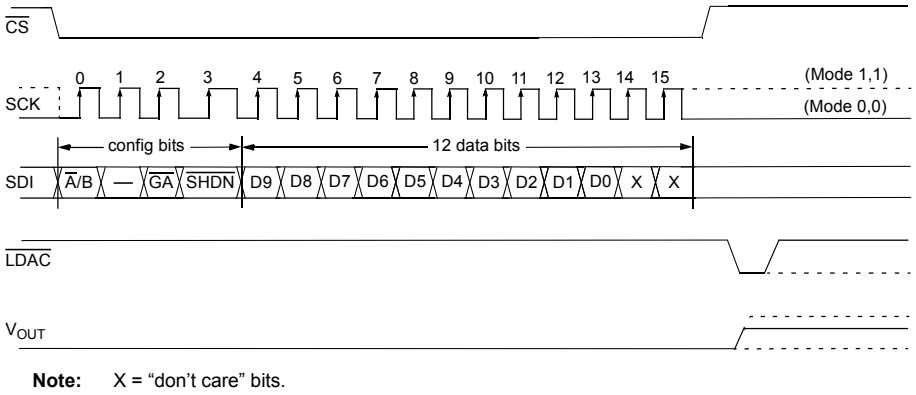

Figura 5.6: Timing della riproduzione di un campione per l'MCP4812.

Poiché si sfrutta la comunicazione SPI, sempre in modalità (0,0), oltre all'inizializzazione dell'interfaccia l'unica altra operazione è quella di configurare il pacchetto che viene inviato alla periferica.
Con riferimento al datasheet del convertitore[8] , si scelgono determinate condizioni operative, nei bit 12 – 15, a cui viene fatto seguire il dato a 10 bit. Le operazioni sulle linee di controllo $\overline{LDAC}$ e $\overline{CS}$ gestite dai bit `LATD.LATD1` e `LATD.LATD0`.
Ciò viene svolto in poche righe direttamente nel main del programma relativo al coordinatore con le seguenti istruzioni.

```
data_dac = ((data << 2) & 0x0FFC) | 0x3000;
LATDbits.LATD0 = 0
SPI_write_16(data_dac);
LATDbits.LATD0 = 1
LATDbits.LATD1 = 0
LATDbits.LATD1 = 1
```

In cui `SPI_write_16` è una funzione (contenuta in `SPI_handler.h`) in cui viene trasmesso via SPI il numero a 16 bit che si intende inviare, senza alcun tipo di formattazione.

[8] inserire riferimento

5.3.5 Temporizzazione

Per la temporizzazione di alcuni eventi eventi viene utilizzato un timer, nello specifico il modulo Timer0. Uno schema riassuntivo, come si trova sul datasheet, è riportato in Figura 5.7.

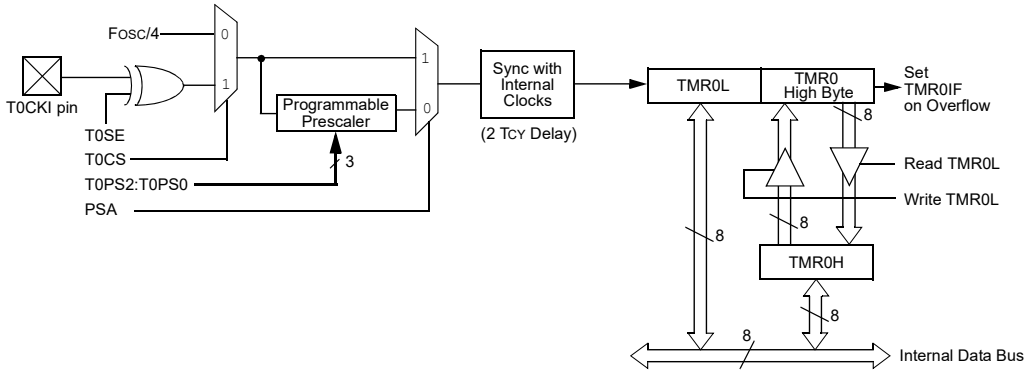

Figura 5.7: Diagramma a blocchi per Timer0 in modalità a 16 bit.

La sorgente di clock è selezionabile tra l'oscillatore a $F_{OSC}/4$ oppure da un pin esterno, T0CKI. È dotato di un prescaler per rallentare il conteggio, dividendone la frequenza per un valore tra 2 e 256.
Ad ogni impulso di clock in uscita dal prescaler viene incrementato il registro `TMR0`. Quando si verifica un overflow, da 0xFF a 0x00 o da 0xFFFF a 0x0000 se il timer è impostato rispettivamente ad 8 o a 16 bit, viene settato il flag `TMR0IF`. Se è abilitato anche il corrispondente flag `T0IE` si entrerà eventualmente nella routine di interrupt, in cui si dovrà ripristinare il valore iniziale da cui far cominciare il conteggio, `TMR0`, e azzerare il flag.
L'intervallo temporale conteggiato dal timer, in questo caso a 8 bit, è dato da

$$T_{tmr} = (\texttt{0xFF} - \texttt{TMR0} + 1) \cdot \frac{\texttt{PSC}}{(F_{OSC}/4)} \tag{5.3}$$

in cui `PSC` è il valore del prescaler.
Questo modulo viene utilizzato per temporizzare l'inizio dell'operazione di campionamento, che si ricorda funzionare soltanto in seguito ad un trigger.

5.4 Modulo MRF24J40

In questa sezione vengono riportate le principali funzioni sviluppate per la configurazione ed il dialogo con il transceiver. Queste sono state dichiarate nel file `mrf24j40.h`:

```
void MRF_init( void );
void MRF_poweron( void );
void MRF_hard_reset( void );
void MRF_soft_reset( void );
void MRF_RF_state_machine_reset( void );
void MRF_set_channel(uint8_t channel);

// see DS39776C-page 64 for details. Parameters are hex numbers
void MRF_set_tx_power(uint8_t tx_range, uint8_t tx_fine);
// get Received Signal Strength Indicator
uint8_t MRF_get_RSSI( void );

void MRF_trigger_TX( void );
```

L'inizializzazione del transceiver è stata suddivisa in più parti: nella funzione `mrf_init` vengono configurati alcuni parametri, non legati alla struttura della rete nè al ruolo nel nodo all'interno di essa, come la potenza di trasmissione, mentre le impostazioni relative ai parametri della rete vengono configurate nelle funzioni di inizializzazione del nodo.

Innanzitutto è necessario un soft reset per azzerare il contenuto dei registri di controllo, quindi si impostano un certo numero di parametri, il cui significato è brevemente illustrato nel codice riportato, in cui si elenca il contenuto della funzione `MRF_init`, definita nel file `MRF24J40.c`.
In particolare alla riga 21 viene configurato il canale da utilizzare per la comunicazione, e a partire dalla riga 44 vengono assegnati al transceiver indirizzi e PAN ID (di default) per poter ricevere correttamente i pacchetti.
Per la valutazione del canale e della qualità della comunicazione tra due nodi viene richiesto il valore di RSSI alla fine del pacchetto ricevuto (riga 40).
Alcuni parametri vengono impostati con i valori raccomandati nel datasheet, per interi registri o per parti di essi, ad esempio come alla riga 14.

```
1  void MRF_init()
2  {
3    // MRF24J40 initialization routine
4
5    SPI_write_short(SOFTRST, 0x07);     // soft reset
6    // flush RX FIFO, enable reception of all packet types (no filtering)
7    SPI_write_short(RXFLUSH, 0x01);
8    SPI_write_short(PACON0, 0x29);      // Enable FIFO
9    SPI_write_short(PACON1, 0x02);
10   SPI_write_short(PACON2, 0x98);
11   // VCO stabilization periods & MSIFS , minSIFS=RFSTBL+MSIFS ()
12   SPI_write_short(TXSTBL, 0x93);
13
14   SPI_write_short(TXPEND, 0x7C);      // recommended value
15   SPI_write_short(FRMOFFSET, 0x15);   // recommended value
16   SPI_write_short(TXTIME, 0x30);      // recommended value
17
18   // set channel & initilize RFOPT
19   uint8_t channel_reg = 0x00;
20   channel_reg = ( ( (MRF_CHANNEL_MIN-11) << 4) & 0xF0) | 0x03;
21   SPI_write_long (RFCON0, channel_reg|0x03);
```

```
22
23    SPI_write_long (RFCON1, 0x02);
24    // enable PLL for RF TX or RX
25    SPI_write_long (RFCON2, 0x80);
26    // set tx power
27    SPI_write_long (RFCON3, 0x00);      // max power
28    SPI_write_long (RFCON6, 0x90);      // (recommended)
29    // Sleep clock selection - internal, 100kHz
30    SPI_write_long (RFCON7, 0x80);
31    // RF VCO
32    SPI_write_long (RFCON8, 0x10);
33    // remove clockout (discontinued) & SLPCLKDIV=1
34    SPI_write_long (SLPCON1, 0x21);
35
36    // Channel access specifications
37    SPI_write_short(BBREG0 , 0x00);     // Set standard mode (no turbo).
38    SPI_write_short(BBREG2 , 0xB8);     // Set CCA mode to ED.
39    SPI_write_short(CCAEDTH, 0x60);     // Set CCA ED threshold.
40    SPI_write_short(BBREG6 , 0x40);     // Set appended RSSI value to RXFIFO
41    SPI_write_short(ACKTMOUT, 0x39);
42
43    //program long & short mac address & pan id
44    SPI_write_short(PANIDL,  SourcePanID & 0x00ff );
45    SPI_write_short(PANIDH, (SourcePanID & 0xff00) >> 8);
46    SPI_write_short(SADRL,  SourceShortAddress & 0x00ff );
47    SPI_write_short(SADRH, (SourceShortAddress & 0xff00) >> 8);
48    SPI_write_short(EADR0, (SourceExtendedAddress_L & 0x000000ff) );
49    SPI_write_short(EADR1, (SourceExtendedAddress_L & 0x0000ff00) >> 8);
50    SPI_write_short(EADR2, (SourceExtendedAddress_L & 0x00ff0000) >> 16);
51    SPI_write_short(EADR3, (SourceExtendedAddress_L & 0xff000000) >> 24);
52    SPI_write_short(EADR4, (SourceExtendedAddress_H & 0x000000ff) );
53    SPI_write_short(EADR5, (SourceExtendedAddress_H & 0x0000ff00) >> 8);
54    SPI_write_short(EADR6, (SourceExtendedAddress_H & 0x00ff0000) >> 16);
55    SPI_write_short(EADR7, (SourceExtendedAddress_H & 0xff000000) >> 24);
56
57    // Enable interrupts
58    SPI_write_short(MRF_INTCON, MRF_INTERRUPTS);
59
60    // reset RF state machine
61    SPI_write_short(RFCTL, 0x04);
62    SPI_write_short(RFCTL, 0x00);
63
64    __delay_us(500);                    // delay at least 192us
65    return;
66  }
```

Una seconda funzione definita in questi file è quella per l'avvio della trasmissione dei dati contenuti nella TX Normal FIFO, `MRF_trigger_TX`.

```
1  void MRF_trigger_TX()
2  {
3  // trigger transmission
4    SPI_write_short(TXNCON, 0x01);
5    SPI_read_short(MRF_INTSTAT);
6    return;
7  }
```

5.5 Gestione della rete

Le funzioni relative alla gestione della rete si occupano ad un livello più alto delle connessioni tra i nodi, operando sui buffer per la creazione di pacchetti e la configurazione della rete.

5.5.1 Definizione dei pacchetti

Prima di elencare le funzioni relative alla gestione dei nodi è lecito riportare la struttura dei campi utilizzati per la creazione dei pacchetti. Tali definizioni sono contenute nei file `frame_ctrl_defs.h`, `beacon_frame.h`, `data_frame.h` e `MAC_cmd_frame.h`, in cui i campi hanno lo stesso significato degli omonimi riportati nello standard.
Nelle sezioni successive si riportano tali strutture dati e le parti più significative dei relativi file.

5.5.1.1 `frame_ctrl_defs.h`

In questo file vengono definite le parole chiave e la struttura dati per la gestione del campo *Frame Control*, comune a tutti i tipi di pacchetto. Viene dichiarato anche il campo di indirizzo, che contiene *Short Address*, *Extended Address* e *PAN Identifier*.

```
// Frame control fields
#define FRAME_TYPE_BEACON            0x0
#define FRAME_TYPE_DATA              0x1
#define FRAME_TYPE_ACK               0x2
#define FRAME_TYPE_MAC_COMM          0x3

#define PAN_ID_COMP_NO               0x0
#define PAN_ID_COMP                  0x1

#define ADDR_MODE_NONE               0x0
#define ADDR_MODE_SHORT              0x2
#define ADDR_MODE_EXTENDED           0x3

#define FRAME_VERSION_2003           0x0
#define FRAME_VERSION_2006           0x1

typedef union
{
  uint16_t frame_control;
  struct  // LSB first
  {
    uint16_t FrameType          :3;
    uint16_t SecurityEnabled    :1;
    uint16_t FramePending       :1;
    uint16_t ACKRequest         :1;
    uint16_t PanIDCompression   :1;
    uint16_t                    :1;
    uint16_t                    :2;
    uint16_t DstAddrMode        :2;
    uint16_t FrameVersion       :2;
    uint16_t SrcAddrMode        :2;
  };
} FrameControl;

typedef struct
```

```
{
  uint32_t extendedAddrL;  // extended address, LOW BYTES (long)
  uint32_t extendedAddrH;  // extended address, HIGH BYTES (long)
  uint16_t shortAddr;
  uint16_t PANID;          // PAN identifier
} AddressField;
```

5.5.1.2 beacon_frame.h

```
typedef struct
{
  FrameControl Frame;     // frame control
  uint8_t SeqNumber;      // packet sequential number
  AddressField CoordAddr;
} MACBeaconHeader;

typedef struct
{
  union
  {
    uint16_t superframe_spec;
    struct                       // LSB first
    {
    uint16_t BEACON_ORDER    :4;
    uint16_t SF_ORDER        :4;
    uint16_t FINAL_CAP_SLOT  :4;
    uint16_t BATT_LIFE_EXT   :1;
    uint16_t                 :1;
    uint16_t PAN_COORD       :1;
    uint16_t ASS_PERMIT      :1;
    };
  };
} superframe_specification;

typedef struct
{
  union
  {
    uint8_t GTS_spec;
    struct
    {
      uint8_t GTS_PERMIT       :1;
      uint8_t                  :4;
      uint8_t GTS_DESCR_COUNT  :3;
    };
  };
} GTS_spec;

typedef struct
{
  union
  {
    uint8_t GTS_directions;
    struct
    {
      uint8_t                  :1;
```

```
      uint8_t GTS_DIR_MASK     :7;
    };
  };
} GTS_directions;

typedef struct
{
  uint16_t DevShortAddr;
  union
  {
    uint8_t GTS_slot_specs;
    struct
    {
      uint8_t GTS_LENGTH       :4;
      uint8_t GTS_START_SLOT   :4;
    };
  };
} GTS_list;

typedef struct
{
  union
  {
    uint8_t pend_addr_specs;
    struct
    {
      uint8_t                  :1;
      uint8_t N_LONG_ADDR_PEND :3;
      uint8_t                  :1;
      uint8_t N_SHORT_ADDR_PEND:3;
    };
  };
} pend_addr_specs;

typedef struct
{
  pend_addr_specs PendingAddrSpecs;
  uint8_t* AddrListPend;
} pend_addr_fields;

typedef struct
{
  GTS_spec       GTSSpecification;
  GTS_directions GTSDirection;
  GTS_list*      GTSListPtr;
} GTS_fields;


typedef struct
{
  MACBeaconHeader macHeader;
  superframe_specification superframeSpec;
  GTS_fields GTSFields;
  pend_addr_fields PendingAddrFields;
  uint8_t* BeaconPayloadPtr;
  uint8_t* BeaconPayloadLen;
} BeaconFrame;
```

5.5.1.3 data_frame.h

```
typedef struct
{
  FrameControl Frame;     // frame control
  uint8_t SeqNumber;      // packet sequential number

  AddressField srcAddr;
  AddressField dstAddr;
} MAC_DATA_HEADER;


typedef struct
{
  MAC_DATA_HEADER macHeader;
  uint8_t  data_frame_counter;
  uint8_t* data_payload;
  uint8_t  payload_length;
} DataFrame;
```

5.5.1.4 MAC_cmd_frame.h

```
// MAC command frames
#define MAC_FRAME_ASS_REQUEST                   0x01
#define MAC_FRAME_ASS_RESPONSE                  0x02
#define MAC_FRAME_DISASS_NOTIFICATION           0x03
#define MAC_FRAME_DATA_REQUEST                  0x04
#define MAC_FRAME_PANID_CONFLICT_NOTIFICATION   0x05
#define MAC_FRAME_ORPHAN_NOTIFICATION           0x06
#define MAC_FRAME_BEACON_REQUEST                0x07
#define MAC_FRAME_COORD_REALIGN                 0x08
#define MAC_FRAME_GTS_REQUEST                   0x09

// Association response
#define MAC_ASS_RES_ASSOCIATION_SUCCESSFUL  0x00
#define MAC_ASS_RES_PAN_AT_CAPACITY         0x01
#define MAC_ASS_RES_PAN_ACCESS_DENIED       0x02

// Disassociation Notification
#define MAC_DISASS_RSN_KICK         0X01
#define MAC_DISASS_RSN_SELF_LEAVE   0X02

// Device type
#define DEVICE_TYPE_FFD 1
#define DEVICE_TYPE_RFD 0

typedef struct
{
  FrameControl Frame;         // frame controls
  uint8_t SeqNumber;      // packet sequential number

  AddressField srcAddr;
  AddressField dstAddr;
} MACcmdHeader;

// Association request
typedef struct
```

```
{
  union
  {
    uint8_t capability_information;
    struct
    {
      uint8_t alternate_PAN_coord :1;
      uint8_t device_type         :1;
      uint8_t power_source        :1;
      uint8_t rec_on_when_idle    :1;
      uint8_t                     :2;
      uint8_t security_capability :1;
      uint8_t allocate_address    :1;
    };
  };
} AssociationRequest;

// Association response
typedef struct
{
  uint16_t short_address;
  uint8_t  association_status;
} AssociationResponse;

// Disassociation Notification
typedef uint8_t DisassociationNotification;

// Coordinator Realignment
typedef struct
{
  uint16_t PAN_identifier;
  uint16_t coord_short_addr;
  uint8_t  channel;
  uint16_t short_address;
  uint8_t  channel_page;
} CoordinatorRealignment;

// GTS request
typedef struct
{
  union
  {
    uint8_t GTS_characteristics;
    struct
    {
    uint8_t GTS_length              :4;
    uint8_t GTS_direction           :1;
    uint8_t characteristics_type    :1;
    uint8_t                         :2;
    };
  };
} GTSRequest;

typedef struct
{
  MACcmdHeader macHeader;
  uint8_t  CmdFrameID;
```

```
    union
    {
      AssociationRequest   association_request;
      AssociationResponse  association_response;
      DisassociationNotification disassociation_notification_reason;
      CoordinatorRealignment coordinator_realignment;
      GTSRequest GTS_request;
    };
  } MACCmdFrame;
```

5.5.2 Definizione della rete

Le definizioni per le proprietà delle connessioni, utilizzate nella gestione dei nodi, sono contenute invece nel file `shared_vars.h`. Si riportano le definizioni relative alle connessioni del nodo, contenute nel file in esame.

```
// NODE PROPERTIES
uint8_t deviceType;
uint8_t powerSource;
uint8_t radio_channel;

// Addressing data
uint32_t SourceExtendedAddress_L;
uint32_t SourceExtendedAddress_H;
uint16_t SourceShortAddress;
// AND SOURCE PAN ID
uint16_t SourcePanID;

typedef union
{
  uint8_t device_properties;
  typedef struct
  {
    uint8_t associated  :1;     // 1-> device connected
    uint8_t deviceType  :1;     // 1-> FFD, 0-> RFD
    uint8_t deviceCoord :1;     // 1-> PAN coordinator
    uint8_t             :5;
  };
} DeviceProperties;

typedef struct
{
  DeviceProperties deviceProp;
  AddressField deviceAddr;
} neighborDevice;

uint8_t n_connections;              // number of associated devices
neighborDevice deviceList[DEVICE_MAX_CONNECTIONS];
uint16_t last_ShortAddr;
```

La lista delle connessioni `deviceList`, limitata a 16 dispositivi, mantiene in memoria tutti i nodi associati col dispositivo. Questa struttura contiene extended address, short address e tipologia di tali nodi.
Siccome l'allocazione dinamica per l'utilizzo di liste di dimensione variabile non è possibile, si dichiara staticamente tutto l'array, che avrà un'occupazione in memoria pari a 208 byte, ricordando che il tipo `AddressField`, definito nella sezione 5.5.1.1, è composto da 12 byte.

Sempre a causa dell'assenza dell'allocazione dinamica, in fase di richiesta di connessione da parte di un nodo bisogna effettuare una ricerca all'interno della lista per evitare di assegnare indirizzi già usati da altri nodi e per verificare se un dispositivo è già connesso al nodo in esame. In fase di disconnessione un determinato dispositivo viene rimosso dalla lista, che verrà riordinata per velocizzare le operazioni di inserimento dei dati per eventuali nuove associazioni.

Se un nodo non è associato con il coordinatore (o con un generico nodo di destinazione con cui intende comunicare) i pacchetti ricevuti da quest'ultimo e provenienti dal nodo non connesso verranno semplicemente ignorati e rappresenteranno, dal punto di vista del traffico, soltanto un'occupazione del canale di comunicazione.

5.5.3 Configurazione di avvio

All'avvio i parametri del nodo vengono configurati, prima della configurazione del transceiver e della relativa inizializzazione, chiamando la funzione `init_node`, contenuta nel file `node_handler.c`.
Le operazioni da compiere sono quelle di assegnazione degli indirizzi di default, eventualmente ottenuti dalla EEPROM contenuta nella MCU, e dall'inizializzazione della lista di connessioni relative al nodo.

```
1  void init_node()
2  {
3    n_connections = 0;
4
5    packet_cnt = 0;
6    data_packet_cnt = 0;
7
8    SourceExtendedAddress_L = EXTENDED_ADDRESS_L;
9    SourceExtendedAddress_H = EXTENDED_ADDRESS_H;
10 #ifdef PAN_COORDINATOR
11   SourceShortAddress = SHORT_ADDRESS;
12   SourcePanID = PAN_ID;
13 #else
14   SourceShortAddress = 0xffff;
15   SourcePanID = 0xffff;
16 #endif
17   last_ShortAddr = SHORT_ADDRESS;
18
19   uint8_t idx;
20   for(idx = 0; idx < DEVICE_MAX_CONNECTIONS; idx++)
21   {
22     deviceList[idx].deviceProp.associated = 0;
23     deviceList[idx].deviceProp.deviceType = DEVICE_TYPE_FFD;
24     deviceList[idx].deviceProp.deviceCoord = 0;
25     deviceList[idx].deviceAddr.extendedAddrH = 0x00000000;
26     deviceList[idx].deviceAddr.extendedAddrL = 0x00000000;
27   }
28
29   return;
30 }
```

In fase operativa i parametri relativi a device non connessi (osservando il campo utilizzato alla riga 22) vengono ignorati, quindi i valori degli altri parametri diventano di secondaria importanza.

5.5.4 Costruzione della rete p2p

In seguito all'inizializzazione generica del transceiver si passa a quella specifica del nodo. La costruzione della rete avviene di fatto soltanto in modo formale, informando il coordinatore che si intende comunicare e trasmettere pacchetti di dati.
Una volta verificata la lista delle connessioni, il coordinatore assegnerà un indirizzo di tipo short al nodo che intende associarsi e soltanto in seguito i pacchetti provenienti da questo potranno essere accettati ed inoltrati al sistema di storage, ovvero il computer connesso tramite porta seriale. Il transceiver viene configurato per filtrare in automatico eventuali pacchetti con errori CRC o non diretti verso il nodo in ricezione tramite la configurazione del registro `RXMCR`.
La funzione di inizializzazione per nodo peer-to-peer (coordinatore o device) è la seguente `p2p_initialize`, definita in `node_handler.c`.

```
1  void p2p_initialize()
2  {
3    SPI_write_short(TXMCR, 0x3C);
4
5  #ifdef PAN_COORDINATOR
6    SPI_write_short(RXMCR, 0x0C);
7  #else
8    SPI_write_short(RXMCR, 0x00);
9  #endif
10
11   SPI_write_short(ORDER, 0xff);
12   SPI_read_short(INTSTAT);
13
14   return;
15 }
```

Una rete peer-to-peer non è soggetto alla struttura a superframe, quindi si devono configurare i campi relativi a *Beacon Order* e *Superframe Order*, contenuti nel registro `ORDER`, ponendoli entrambi a `0xf` (riga 11).

5.5.5 Costruzione della rete beacon-enabled

Con riferimento alla funzione `beacon_initialize`, riportata a fine sezione, per quanto riguarda la rete beacon-enabled, come per il caso peer-to-peer, si configura il ruolo del nodo nella rete (righe 7 e 9) e si definisce la struttura della rete, in particolare la presenza di GTS (righe 16 – 19).
Altra operazione importante per questo tipo di rete è la temporizzazione relativa all'invio e alla ricezione del beacon. Per generarla il transceiver utilizza una coppia di timer, usati in sequenza: uno di questi (MAINCLK) è derivato direttamente dall'oscillatore al quarzo esterno da 20 MHz, l'altro (SLPCLK) è basato su una sorgente scelta tra l'oscillatore interno da 100 kHz ed uno esterno a 32 kHz.
I due timer sono *main counter*, il cui contatore a 26 bit è indicato come MAINCNT e la cui sorgente di clock è SLPCLK, e *remain counter*, con un contatore a 16 bit (REMCNT) associato a MAINCLK.
Il clock più lento, come si vede in figura, può essere scelto tra due diverse sorgenti tramite il bit `RFCON7.SLPCLKSEL`. Sulla daughter card utilizzata la sorgente esterna non è presente, quindi si deve per forza la sorgente interna da 100 kHz. Questo oscillatore, prima di poter essere utilizzato deve essere calibrato. Per compiere questa operazione si sfrutta il clock principale associato ad un contatore a 20 bit per la stima della durata di 16 cicli di

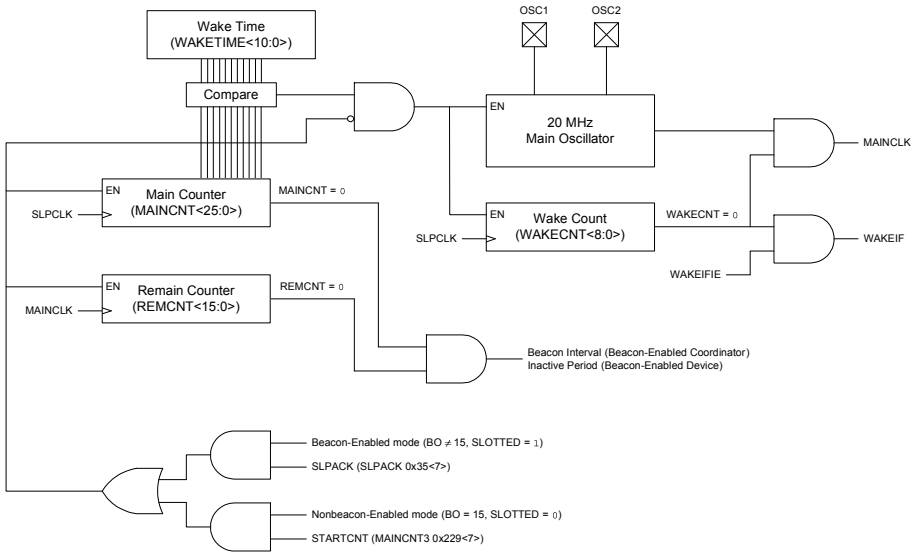

Figura 5.8: Schema a blocchi dei due counter.

SLPCLK. Al termine della calibrazione tale valore sarà contenuto nel registro `SLPCAL`.
Il periodo di SLPCLK viene quindi calcolato come:

$$T_{SLPCLK} = \text{SLPCAL} \cdot \frac{50\,ns}{16} \tag{5.4}$$

questa operazione viene eseguita alle righe 30 – 33 della funzione `beacon_initialize`.

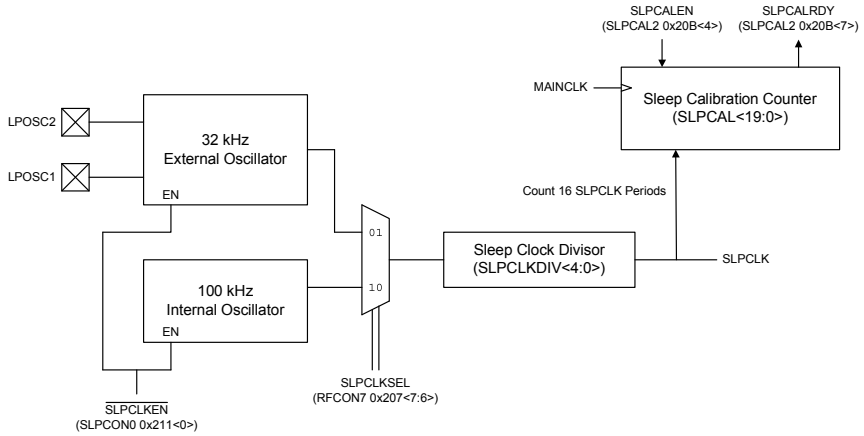

Figura 5.9: Generazione di SLPCLK.

Per il coordinatore di una rete beacon-enabled la somma di questi tempi corrisponde al Beacon Interval (BI) del superframe, come mostrato nella (5.5).

$$\text{Beacon Interval} = (\text{MAINCNT} \cdot T_{\text{SLPCLK}}) + \text{REMCNT} \cdot 50\,ns \tag{5.5}$$

Questo calcolo viene svolto in automatico dal microcontrollore alle righe 45 – 50, in cui si sfruttano le operazioni tra interi per calcolare i valori da assegnare i registri `MAINCNT` e `REMCNT`.
Per un device di una rete beacon-enabled la grandezza espressa dalla (5.5) non rappresenta più il Beacon Interval ma la porzione inattiva del superframe. In Figura 5.10 è riportata l'operazione combinata dei due timer per il coordinatore della rete, in questo caso è presente

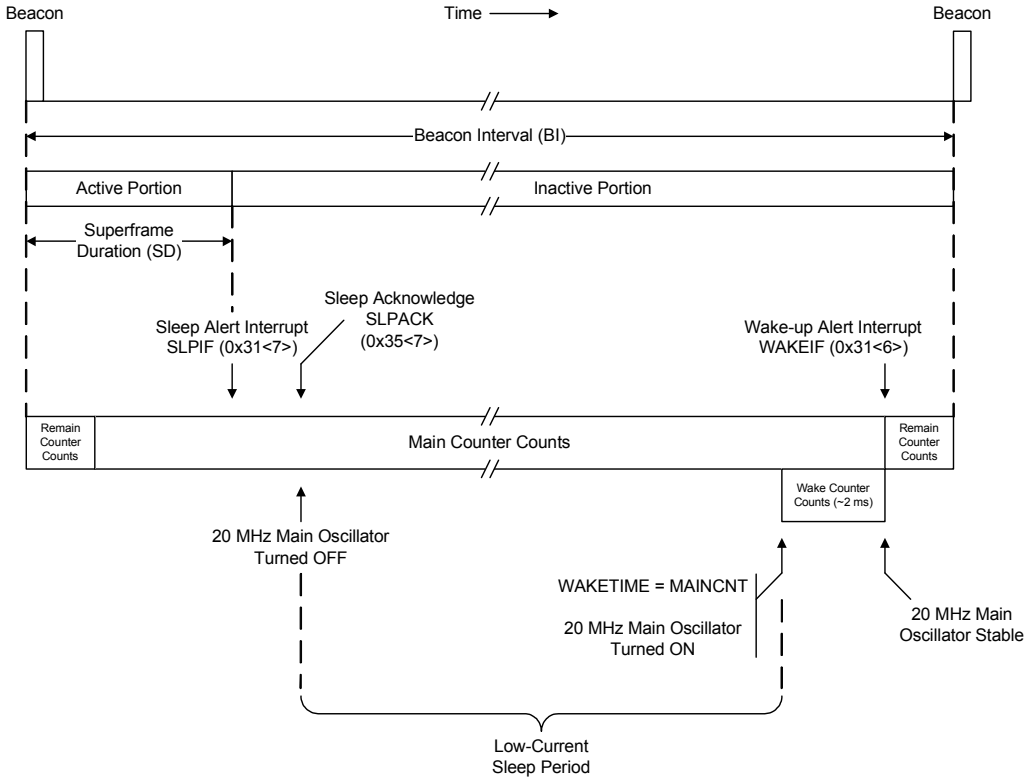

Figura 5.10: Timeline per il coordinatore di una rete beacon-enabled.

anche un periodo inattivo.

```
1  void beacon_initialize()
2  {
3    // reconfigure the transceiver for beacon enabled network
4    SPI_write_short(TXMCR, 0b00100100);
5
6  #ifdef PAN_COORDINATOR
7    SPI_write_short(RXMCR, 0x0C);
8  #else
9    SPI_write_short(RXMCR, 0x00);
10   SPI_write_short(FRMOFFSET, 0x15);
11 #endif
12
13   SPI_write_short(TXBCON1, 0x30);
14   SPI_write_short(WAKECON, 0x03);     // disable immediate wakeup
15
16   SPI_write_short(GATECLK, 0x00);
17   SPI_write_short(ESLOTG1, 0xFF);
18   SPI_write_short(ESLOTG23, 0xFF);
19   SPI_write_short(ESLOTG45, 0xFF);
20   SPI_write_short(ESLOTG67, 0x0F);
21
22   SPI_write_short(SLPACK, 0x5F);
23
24   SPI_write_long(RFCON7, 0x80);
25
```

```
26    // use range counter SLPCLK
27    SPI_write_long(SLPCON0, 0x00);  // INTEDGE=0
28    SPI_write_long(SLPCON1, 0x21);
29
30    //   SLPCLK calibration
31    SPI_write_long(SLPCAL2, 0x10);  // trigger calibration
32    do
33      __delay_ms(1);
34    while( (SPI_read_long(SLPCAL2)&0x80) == 0x00 );   // wait for calibration
35
36    #define WAKETIME  0x0D2
37    #define WAKECNT   0x0C8
38
39    SPI_write_long(WAKETIMEL, WAKETIME & 0x0FF);
40    SPI_write_long(WAKETIMEH, WAKETIME >> 8);
41    SPI_write_short(SLPACK, (WAKECNT & 0x7F));
42    SPI_write_short(RFCTL, (WAKECNT >> 4)& 0x18);
43
44  #ifdef PAN_COORDINATOR
45    // PROGRAM MAINCNT & REMCNT (BO=2, SO=2)
46    uint32_t TBO = 4915200 * 4;
47    uint32_t SLPCAL = ((uint32_t)(slpcal_value[2])<<16)
48                    + ((uint32_t)(slpcal_value[1])<<8)
49                    | slpcal_value[0];
50    uint32_t MAINCNT = TBO/SLPCAL;
51    uint32_t REMCNT = (TBO - MAINCNT*SLPCAL)/16;
52
53    SPI_write_long(MAINCNT0, MAINCNT & 0x000000FF);
54    SPI_write_long(MAINCNT1, (MAINCNT >> 8) & 0x000000FF);
55    SPI_write_long(MAINCNT2, (MAINCNT >> 16) & 0x000000FF);
56    SPI_write_long(MAINCNT3, (MAINCNT >> 24) & 0x000000FF);
57
58    SPI_write_long(REMCNTL, REMCNT & 0x000000FF);
59    SPI_write_long(REMCNTH, (REMCNT >> 8) & 0x000000FF);
60
61    // enable wakeif to read beacon transmission
62    uint8_t tmp_reg = SPI_read_short(MRF_INTCON);
63    SPI_write_short(MRF_INTCON, tmp_reg & 0x3f);
64  #else
65    SPI_write_long(MAINCNT0, MAINCNT & 0x00000000);
66    SPI_write_long(MAINCNT1, (MAINCNT >> 8) & 0x00000000);
67    SPI_write_long(MAINCNT2, (MAINCNT >> 16) & 0x00000000);
68    SPI_write_long(MAINCNT3, (MAINCNT >> 24) & 0x00000000);
69
70    SPI_write_long(REMCNTL, REMCNT & 0x00000000);
71    SPI_write_long(REMCNTH, (REMCNT >> 8) & 0x00000000);
72  #endif
73
74    SPI_read_short(MRF_INTSTAT);
75
76    SPI_write_short(ORDER, 0x22);
77
78    // reset RF state machine
79    SPI_write_short(RFCTL, 0x04);
80    SPI_write_short(RFCTL, 0x00);
81
82    // delay at least 192us
```

```
83    __delay_us(300);
84
85    beacon_cnt = 0;
86    return;
87  }
```

Alla riga 12 viene abilitato l'interrupt di trasmissione del beacon. Visto che quest'ultimo viene inviato automaticamente in base ai periodi impostati, è necessario per notificare alla MCU l'inizio del superframe e aggiornare di conseguenza il contenuto della FIFO `TXBFIFO` dell'MRF24J40, eventualmente anche soltanto per il Sequence Number del beacon frame.

5.5.6 Trasmissione e ricezione dei pacchetti

Per quanto riguarda la trasmissione dei pacchetti, una volta caricati in un'area di memoria (FIFO) all'interno del transceiver, questi vengono inviati in seguito alla ricezione da parte del transceiver di un trigger di trasmissione, ovvero in seguito alla scrittura del flag `TXNTRIG` (per Normal FIFO) nel registro `TXNCON`. Vengono inoltre resettati tutti i flag di interrupt del transceiver leggendo il registro `INTSTAT`.

```
SPI_write_short(TXNCON, 0x01);
SPI_read_short(INTSTAT);
```

Un esempio di creazione di un pacchetto, in questo caso si tratta di un frame di tipo dati, è quella contenuta in `data_handler.c` nella funzione `send_data`, che scrive il pacchetto dati nel buffer locale in attesa dell'invio. In questo caso non viene richiesto un acknowledgment e si sfrutta l'indirizzamento tramite short address.
Il pacchetto viene costruito fino alla riga 17, poi viene inviato al buffer interno.

```
1   void send_data(uint16_t* destShortAddr,
2           uint8_t* data_ptr, uint8_t data_len)
3   {
4   //  create data packet
5     data_frame.macHeader.Frame.frame_control = FRAME_CONTROL_DATA |
6   //          FRAME_CONTROL_ACK_REQUEST |
7             FRAME_CONTROL_SRC_ADDR_SHORT |
8             FRAME_CONTROL_DST_ADDR_SHORT | FRAME_CONTROL_VERSION_2006;
9
10    data_frame.macHeader.SeqNumber = ++packet_cnt;
11
12    data_frame.macHeader.srcAddr.shortAddr = SourceShortAddress;
13    data_frame.macHeader.dstAddr.shortAddr = *destShortAddr;
14    data_frame.macHeader.srcAddr.PANID = SourcePanID;
15    data_frame.macHeader.dstAddr.PANID = SourcePanID;
16
17    data_frame.data_frame_counter = ++data_packet_cnt;
18
19    // WRITE PACKET INTO THE BUFFER
20    uint8_t HeaderLength;
21    uint8_t FrameLength;
22
23    bufferPut((uint8_t*)(&data_frame.macHeader.Frame.frame_control),
24               2, ADDR_FRAME);
25    bufferPut((uint8_t*)(&data_frame.macHeader.SeqNumber),
26               1, ADDR_DATA_SEQNUM);
27
28    TX_buffer_size = ADDR_DATA_ADDRFIELDS;
29    // ADDRESSING FIELDS
```

```
30    bufferPut((uint8_t*)(&data_frame.macHeader.dstAddr.PANID),
31               2, ADDR_DATA_ADDRFIELDS);
32    TX_buffer_size += 2;
33
34    if( data_frame.macHeader.Frame.DstAddrMode == ADDR_MODE_SHORT )
35    {
36      bufferPut((uint8_t*)(&data_frame.macHeader.dstAddr.shortAddr),
37                 2, TX_buffer_size);
38      TX_buffer_size += 2;
39    }
40    else if( data_frame.macHeader.Frame.DstAddrMode == ADDR_MODE_EXTENDED)
41    {
42      bufferPut((uint8_t*)(&data_frame.macHeader.dstAddr.extendedAddrL),
43                 4, TX_buffer_size);
44      bufferPut((uint8_t*)(&data_frame.macHeader.dstAddr.extendedAddrH),
45                 4, TX_buffer_size+4);
46      TX_buffer_size += 8;
47    }
48    // source panID and address
49    if( data_frame.macHeader.Frame.PanIDCompression == PAN_ID_COMP_NO )
50    {
51      bufferPut((uint8_t*)(&data_frame.macHeader.srcAddr.PANID),
52                 2, TX_buffer_size);
53      TX_buffer_size += 2;
54    }
55    if( data_frame.macHeader.Frame.SrcAddrMode == ADDR_MODE_SHORT )
56    {
57      bufferPut((uint8_t*)(&data_frame.macHeader.srcAddr.shortAddr),
58                 2, TX_buffer_size);
59      TX_buffer_size += 2;
60    }
61    else if( data_frame.macHeader.Frame.SrcAddrMode == ADDR_MODE_EXTENDED )
62    {
63      bufferPut((uint8_t*)(&data_frame.macHeader.srcAddr.extendedAddrL),
64                 4, TX_buffer_size);
65      bufferPut((uint8_t*)(&data_frame.macHeader.srcAddr.extendedAddrH),
66                 4, TX_buffer_size+4);
67      TX_buffer_size += 8;
68    }
69    // HeaderLength and FrameLength buffer bytes are not included in the count
70    HeaderLength = TX_buffer_size-2;
71
72    // write Payload
73    data_frame.data_payload = data_ptr;
74    data_frame.payload_length = data_len;
75
76  //  bufferPut( &data_frame.data_frame_counter, 1, TX_buffer_size);
77  //  TX_buffer_size += 1;
78    bufferPut( data_frame.data_payload, data_frame.payload_length,
79               TX_buffer_size);
80    TX_buffer_size += data_frame.payload_length;
81
82    FrameLength =  TX_buffer_size-2;
83
84    bufferPut(&HeaderLength, 1, ADDR_HDRLENGTH);    //header length
85    bufferPut(&FrameLength , 1, ADDR_FRAMELENGTH);  //frame length
86
```

```
87    return;
88  }
```

Per quanto riguarda la ricezione di un pacchetto, questa viene notificata dal transceiver al microcontrollore tramite il pin di interrupt. Una volta identificato il flag di ricezione la MCU disabilita l'interfaccia radio, per evitare che altri eventuali dati in ricezione vadano ad interferire con quelli appena ricevuti, e si trasferisce il contenuto del buffer `RXFIFO` nel buffer interno.

```
SPI_write_short(BBREG1, 0x04);  // disable packet reception
SPI_read_RXFIFO();
SPI_read_short(INTSTAT);
SPI_write_short(BBREG1, 0x00);  // enable packet reception
```

In cui il trasferimento dei dati dal transceiver al buffer interno è dato dalla seguente funzione

```
void SPI_read_RXFIFO()
{
  RX_buffer_size = SPI_read_long( RX_FIFO_FRAME_LENGTH ) + 2;
  uint8_t* RX_dest_buffer = RX_buffer;

  uint16_t i;
  uint16_t addr;
  for(i = 0, addr = RX_FIFO_FIRST_ADDRESS; i < RX_buffer_size;
       i++, addr += 0x20)
    *(RX_dest_buffer++) = SPI_read_long( addr );

  return;
}
```

Una volta ricevuto il pacchetto non resta che decodificarlo ed estrarne le informazioni. Ciò viene fatto chiamando la funzione `read_frame`, contenuta in `node_handler.*`, che identifica il tipo di frame e la relativa provenienza, che si occupa in automatico soltanto della connessione/disconnessione dei nodi (ora omesse, si trovano alle righe 66, 69 e 72). Eventuali altri comandi con risposta automatica non sono stati implementati in quanto non utilizzati all'interno dell'applicazione sviluppata.

```
1   void read_frame()
2   {
3     // Read frame type from RX_buffer
4     FrameControl fc;
5     fc.frame_control = (RX_buffer[1] << 8) | RX_buffer[0];
6     uint8_t idx = 2;    // RX_buffer payload index
7     // Get addressing fields
8     AddressField add1;
9     AddressField add2;  // Source (packet sender)
10
11    // | Src Addr | Src Pan | Dst Addr | Dst Pan |
12    if( fc.DstAddrMode == ADDR_MODE_EXTENDED )
13    {
14      add1.extendedAddrH = RX_buffer[idx+3]<<24 | RX_buffer[idx+2]<<16
15                           | RX_buffer[idx+1]<<8 | RX_buffer[idx];
16      add1.extendedAddrL = RX_buffer[idx+7]<<24 | RX_buffer[idx+6]<<16
17                           | RX_buffer[idx+5]<<8 | RX_buffer[idx+4];
18      idx += 8;
19    }
20    else if ( fc.DstAddrMode == ADDR_MODE_SHORT )
```

```
21    {
22      add1.shortAddr = RX_buffer[idx+1]<<8 | RX_buffer[idx];
23      idx += 2;
24    }
25    if ( fc.PanIDCompression != PAN_ID_COMP )
26    {
27      add1.PANID = RX_buffer[idx+1]<<8 | RX_buffer[idx];
28      idx += 2;
29    }
30    if( fc.DstAddrMode == ADDR_MODE_EXTENDED )
31    {
32      add2.extendedAddrH = RX_buffer[idx+3]<<24 | RX_buffer[idx+2]<<16
33                           | RX_buffer[idx+1]<<8 | RX_buffer[idx];
34      add2.extendedAddrL = RX_buffer[idx+7]<<24 | RX_buffer[idx+6]<<16
35                           | RX_buffer[idx+5]<<8 | RX_buffer[idx+4];
36      idx += 8;
37    }
38    else if ( fc.DstAddrMode == ADDR_MODE_SHORT )
39    {
40      add2.shortAddr = RX_buffer[idx+1]<<8 | RX_buffer[idx];
41      idx += 2;
42    }
43    add2.PANID = RX_buffer[idx+1]<<8 | RX_buffer[idx];
44    idx += 2;
45
46    // NO SECURITY HEADER
47
48    switch(fc.FrameType)
49    {
50      case FRAME_CONTROL_DATA:
51        // Copy data payload
52        frame_payload.srcShortAddr = add2.shortAddr;
53        uint8_t i;
54        for(i=0; i<RX_buffer_size-idx; i++)
55          frame_payload.dataVec[i] = RX_buffer[idx+i+1];
56
57        frame_payload.data_length = RX_buffer_size - idx;
58        break;
59
60      case FRAME_CONTROL_MAC_COMM:
61      {
62        MACCmdFrame cmd;
63        switch( RX_buffer[idx] )
64        {
65          case MAC_FRAME_ASS_REQUEST:
66               /* CONNECT NODE */
67            break;
68          case MAC_FRAME_ASS_RESPONSE:
69               /* SEND RESPONSE */
70            break;
71          case MAC_FRAME_DISASS_NOTIFICATION:
72               /* SEND NOTIFICATION */
73            break;
74          case MAC_FRAME_DATA_REQUEST:
75            data_TX_flag = 1;
76            break;
77          case MAC_FRAME_PANID_CONFLICT_NOTIFICATION:
```

```
78            // NOT IMPLEMENTED
79            break;
80          case MAC_FRAME_ORPHAN_NOTIFICATION:
81            // NOT IMPLEMENTED
82            break;
83          case MAC_FRAME_BEACON_REQUEST:
84            // NOT IMPLEMENTED
85            break;
86          case MAC_FRAME_COORD_REALIGN:
87            // NOT IMPLEMENTED
88            break;
89          case MAC_FRAME_GTS_REQUEST:
90            // NOT IMPLEMENTED
91            break;
92        }
93      }
94      default:
95        break;
96    }
97
98    return;
99  }
```

5.5.7 Connessione e disconnessione

La connessione e la disconnessione dei nodi viene gestita all'interno di `read_frame`, già citata nella sezione 5.5.6.
Per l'associazione, la cui parte di codice è stata omessa in precedenza, ciò che si fa è di verificare (dal punto di vista del coordinatore) la presenza del device che ha inviato la richiesta nella `deviceList`. Se risultasse assente si può decidere se rifiutare o meno il client, cosa che può essere fatta se la rete è riservata o se è stato raggiunto il limite di dispositivi associati, oppure accettarlo ed inserirlo nella lista delle connessioni. In entrambi i casi viene inviata una risposta tramite una *Association Response*.

```
if(n_connections < DEVICE_MAX_CONNECTIONS) // connect node
{
  uint8_t i;
  // check if the node is already connected
  for(i=0; i < DEVICE_MAX_CONNECTIONS; i++)
  {
    if( add2.extendedAddrH == deviceList[i].deviceAddr.extendedAddrH &&
          add2.extendedAddrL == deviceList[i].deviceAddr.extendedAddrL &&
          deviceList[i].deviceProp.associated )
    {
      i += DEVICE_MAX_CONNECTIONS+1;
      break;
    }
  }
  if(i <= DEVICE_MAX_CONNECTIONS) // not yet connected
  {
    n_connections++;
    // assign short address and insert device in deviceList
    // (last place, the array is ordered)
    last_ShortAddr++;

    deviceList[n_connections].deviceProp.associated = 1;
```

```
      deviceList[n_connections].deviceProp.deviceType = 1;
      deviceList[n_connections].deviceAddr.shortAddr = last_ShortAddr;

      send_association_response(&cmd, &add2.extendedAddrH,
              &add2.extendedAddrL, last_ShortAddr,
              MAC_ASS_RES_ASSOCIATION_SUCCESSFUL);
    }
    else  // already connected
    {
      send_association_response(&cmd, &add2.extendedAddrH,
              &add2.extendedAddrL,
              deviceList[i - DEVICE_MAX_CONNECTIONS-1].deviceAddr.shortAddr,
              MAC_ASS_RES_ASSOCIATION_SUCCESSFUL);
    }
  }
  else  // PAN at limit
  {
    send_association_response(&cmd, &add2.extendedAddrH, &add2.extendedAddrL,
                              0xffff, MAC_ASS_RES_PAN_AT_CAPACITY);
  }
```

Dal lato client, una volta ricevuta una *Association Response* è necessario memorizzare o meno l'indirizzo locale che è stato assegnato ed inserire il coordinatore in `deviceList`. Siccome il primo step per associarsi ad altri dispositivi è quello di entrare nella rete, la notifica sarà diretta al coordinatore, che sarà il primo elemento inserito in tale lista e che avrà indice `0`. Nel caso in cui dovesse essere rifiutata la connessione, nel transceiver verranno scritti i parametri generici per un nodo non connesso.

```
  // Verify association status field
  if(RX_buffer[idx+2] == MAC_ASS_RES_ASSOCIATION_SUCCESSFUL)
  {
  // Write the assigned short address in mrf registers
    SPI_write_short(PANIDL,  add2.PANID & 0x00ff );
    SPI_write_short(PANIDH, (add2.PANID & 0xff00) >> 8);
    SPI_write_short(SADRH, (RX_buffer[idx] & 0xff00) >> 8);
    SPI_write_short(SADRL,  RX_buffer[idx+1] & 0x00ff );
    deviceList[0].deviceAddr.extendedAddrH = add2.extendedAddrH;
    deviceList[0].deviceAddr.extendedAddrL = add2.extendedAddrL;
    deviceList[0].deviceProp.deviceCoord = 1;
    deviceList[0].deviceProp.associated = 1;
    deviceList[0].deviceProp.deviceType = 1;
    n_connections++;
  }
  else
  {
    SPI_write_short(PANIDL, 0xff);
    SPI_write_short(PANIDH, 0xff);
    SPI_write_short(SADRH,  0xff);
    SPI_write_short(SADRL,  0xff);
  }
```

Per la disconnessione, invece, si possono presentare due scenari diversi: uno in cui un nodo decide di lasciare la PAN, ed uno in cui la notifica di disconnessione viene inviata al nodo che si intende escludere dalla rete (dal coordinatore al device).
Nel primo caso è necessario cercare il nodo che si sta disconnettendo e rimuoverlo dalla lista delle connessioni, che dovendo essere priva di spazi vuoti verrà riordinata. Nel secondo caso invece il coordinatore ha già provveduto all'eliminazione del dispositivo dalla sua lista

di connessioni, e quindi il nodo dovrà scrivere le impostazioni generiche per *Short Address* e *PAN ID* nei registri del transceiver.

```
if(RX_buffer[idx] == MAC_DISASS_RSN_SELF_LEAVE) // received from device
{
  // search for the node, remove it from the list, and sort the list
  // REMOVE DEVICE FROM DEVICE LIST
  uint8_t i;
  for(i = 0; i <= DEVICE_MAX_CONNECTIONS; i++)
  {
    if( add2.extendedAddrH == deviceList[i].deviceAddr.extendedAddrH &&
          add2.extendedAddrL == deviceList[i].deviceAddr.extendedAddrL &&
          deviceList[i].deviceProp.associated )
    {
      deviceList[i].deviceProp.associated = 0;
      deviceList[i].deviceAddr.extendedAddrH = 0x00000000;
      deviceList[i].deviceAddr.extendedAddrL = 0x00000000;
      deviceList[i].deviceAddr.PANID = 0xffff;
      deviceList[i].deviceAddr.shortAddr = 0xffff;
      n_connections--;
      break;
    }
  }
  // rearrange the list
  uint8_t sorted;
  do {
    sorted = 1;
    for( i = 0; i < DEVICE_MAX_CONNECTIONS-1; i++ )
    {
      if( !deviceList[i].deviceProp.associated )
      {
        deviceList[i] = deviceList[i+1];
        sorted = 0;
      }
    }
  } while(!sorted);
}
else  // kick, sent by coordinator to the device
{
  SPI_write_short(PANIDL, 0xff);
  SPI_write_short(PANIDH, 0xff);
  SPI_write_short(SADRH,  0xff);
  SPI_write_short(SADRL,  0xff);
  // remove coordinator from the connections
  deviceList[0].associated = 0;
}
```

5.6 Esempio applicativo del codice

Le funzioni viste finora sono utilizzate nel file `main.c` per lo sviluppo dell'applicazione. Viene ora riportato il codice relativo all'implementazione del nodo in una rete peer-to-peer in uno scenario semplificato in cui i due nodi sono già intrinsecamente connessi.

```
extern void SYS_init(void);
extern void TMR0_init(void);
void board_init();
```

```
void set_sampling_time_ms(float time);

void interrupt ISR(void);

volatile uint8_t tx_flag;
volatile uint8_t data_ready;
volatile uint16_t wait_for_sampling;
volatile uint16_t tmr0_count;

customPayload Data;

int main(void)
{
// Initialize modules
  SYS_init();   // Initialize system
  board_init();
  SPI_init();   // Spi
  UART_init();  //  UART

// External interrupt on portb0
  TRISBbits.TRISB0 = 1;
  INTCON2bits.INTEDG0 = 0;
  INTCONbits.INT0IF = 0;
  INTCONbits.INT0IE = 0;

// Timer0
  TMR0_init();
  T0CONbits.TMR0ON = 0;
  INTCONbits.TMR0IE = 0;
  INTCONbits.TMR0IF = 0;

//  Initialize coordinator node
  init_node();
// INITIALIZE MRF24J40MA (on channel 11)
  MRF_init();

//  Initialize client ADC (using AN6, RE1)
  ADCON1bits.VCFG1 = 0;
  ADCON1bits.VCFG0 = 0;
  ADCON2bits.ACQT = 0b011;    // AD acquisition time
  ADCON2bits.ADCS = 0b001;    // AD conversion clock (Fosc/8)
  ADCON1bits.PCFG = 0b1000;   // Enable AN6 to AN0 to be analog pins

  ADCON0bits.CHS = 0x06;      // channel select

// enable adc module
  ADCON0bits.ADON = 1;
  ADCON2bits.ADFM = 1;

// enable adc interrupts
  IPR1bits.ADIP = 1;
  PIR1bits.ADIF = 0;
  PIE1bits.ADIE = 1;

  INTCONbits.INT0IE = 1;

  INTCONbits.GIE = 1;
```

```
  INTCONbits.PEIE = 1;

  uint8_t i = 0;
  tx_flag = 0;
  connected = 0;

  MACCmdFrame cmd;
  AddressField coordAddr;

  coordAddr.extendedAddrH = PAN_COORDINATOR_EXTENDED_ADDRESS_H;
  coordAddr.extendedAddrL = PAN_COORDINATOR_EXTENDED_ADDRESS_L;
  coordAddr.shortAddr = PAN_COORDINATOR_SHORT_ADDRESS;
  coordAddr.PANID = PAN_ID;

 [...]

  while(1)
  {
    while(wait_for_sampling);
    wait_for_sampling = 1;

        // start acquisition
    ADCON0bits.GO_DONE = 1;

    while(!data_ready);
    data_ready = 0;

    Data.data_index++;

        // build local buffer
    uint8_t local_buffer[10];
    write_on_payload_buffer(local_buffer, &Data);

    send_data_to_buffer(&coordAddr.shortAddr, local_buffer, sizeof(Data) );

        // send buffer to transceiver (SPI)
    send_buffer_TX();
  }
  return (EXIT_SUCCESS);
}


void interrupt ISR()
{

[...]

  if( INTCONbits.TMR0IF == 1 )
  {
    INTCONbits.TMR0IF = 0;
    TMR0H = (uint8_t)(tmr0_count >> 8);
    TMR0L = (uint8_t)(tmr0_count);

    LATDbits.LD1 = 1;
    __delay_us(2);
    LATDbits.LD1 = 0;
```

```
      wait_for_sampling = 0;
    }
    if(PIR1bits.ADIF == 1)
    {
      PIR1bits.ADIF = 0;

      Data.data1 = (ADRESH);
      Data.data1 = (Data.data1 << 8) | ADRESL;

      data_ready = 1;
    }
    return;
  }

  [...]
```

Capitolo 6

Test sul sistema

Le prove sul sistema si sono dapprima concentrate sul comportamento dei singoli componenti e quindi sul sistema complessivo.

6.1 Sviluppo e test dell'interfaccia SPI

Un primo test che si è rivelato molto importante per la realizzazione del sistema è stato quello della velocità di trasmissione della comunicazione SPI.
La differenza di implementazione della comunicazione con questo modulo ha un notevole effetto sulla velocità di trasmissione.
Per fare dei confronti tra vari metodi ci si concentra sulla funzione di scrittura di un byte nell'area di memoria con *long address* del transceiver. Si prende in considerazione la funzione `SPI_write_long()` definita in `SPI_handler.c`, visto anche che i buffer FIFO su cui vengono trasferiti i dati saranno associati alla stessa porzione di memoria.
Inizialmente la funzione era costituita in questo modo:

```
inline void SPI_write_long(uint16_t long_address, uint8_t data)
{
// long_address = 1AAAAAAA AAA10000 DDDDDDDD

  _CS = 0;
  PIR1bits.SSPIF = 0;\\
  long_address = (long_address << 5)|0x8010;

  SPI_end_tx = 0;
  SSPBUF = long_address >> 8;
  while(!SPI_end_tx);

  SPI_end_tx = 0;
  SSPBUF = (uint8_t)(long_address);
  while(!SPI_end_tx);

  SPI_end_tx = 0;
  SSPBUF = data;
  while(!SPI_end_tx);

  _CS = 1;
  return;
}
```

In cui una volta entrato nella sopracitata funzione si doveva attendere la commutazione della variabile `SPI_end_tx`, ad opera dell'evento di interrupt, operata nella routine `ISR`.

```
PIE1bits.SSPIE = 1;
[...]
void ISR()
{
  if( PIR1bits.SSPIF )
  {
    PIR1bits.SSPIF = 0;
    SPI_end_tx = true;
  }
  return;
}
```

Una seconda opzione è quella in cui si guarda direttamente il flag di interrupt dell'SPI, `PIR1.SSPIF`, per cui il codice inserito in `ISR` non viene più considerato. I tempi di scrittura sono

$$\begin{aligned} T_i &= 87\,\mu s \\ T_f &= 23.3\,\mu s \end{aligned}$$

Poiché l'uso che viene fatto di questo sistema è quello di un sistema di raccolta e trasmissione dati, non è un problema avere una funzione di attesa che blocca il codice, inoltre per massimizzare il flusso di dati è necessario avere una trasmissione più veloce possibile.
Un ulteriore step di miglioramento si ottiene considerando il trasferimento di tutto il buffer via SPI. La dimensione massima del pacchetto prevista dallo standard IEEE 802.15.4 è di 128 byte [1], quindi le prove seguenti vengono effettuate in un'ottica peggiorativa.
Dal momento che si fa riferimento ad un'area di memoria contigua (quella del buffer) si può utilizzare un puntatore per accedere a tutti gli elementi in maniera iterativa. Anziché utilizzare la funzione `SPI_write_long`, per accedere a tutte locazioni della TX Normal FIFO del transceiver ne viene implementata un'altra in cui il ciclo stesso è situato all'interno della funzione, ovvero `fifoTX_put` appartenente a `node_handler.c`.
Inoltre, a differenza del primo metodo, è stata introdotta un'ulteriore variazione: l'indirizzo viene inserito preformattato (ovvero già nella forma voluta dal transceiver), che consente di avere un risparmio temporale prima della trasmissione del primo byte.
Viene misurato il periodo di trasferimento del singolo byte di dati, i cui grafici sono riportati nelle Figure 6.1.

Si nota molto bene il grande vantaggio derivante dall'utilizzo di indirizzi già mascherati: in Figura 6.1a e 6.1b la funzione utilizzata per il trasferimento dei dati è la stessa, salvo il formato degli indirizzi.
Il raffronto tra le Figure 6.1b e 6.1c evidenzia il risparmio ulteriore di tempo ottenuto eliminando di fatto una chiamata a funzione; si può notare il leggero ritardo (di circa $2\mu s$) nell'invio del terzo byte tra questi due metodi a causa di una differenza di implementazione: per l'accesso al dato viene utilizzato un puntatore, che è necessario dereferenziare. Nonostante ciò il tempo complessivo di trasmissione con `fifoTX_put` risulta minore.
I tempi ottenuti per il singolo dato, relativi al periodo dell'onda sulla linea CS sono:

$$\begin{aligned} T_a &= 25.25\,\mu s \\ T_b &= 15.25\,\mu s \\ T_c &= 13.25\,\mu s \end{aligned} \tag{6.1}$$

Per il metodo più veloce, quindi, si ottiene un tempo complessivo pari a $2.33\,ms$, il cui grafico è riportato in Figura 6.2.

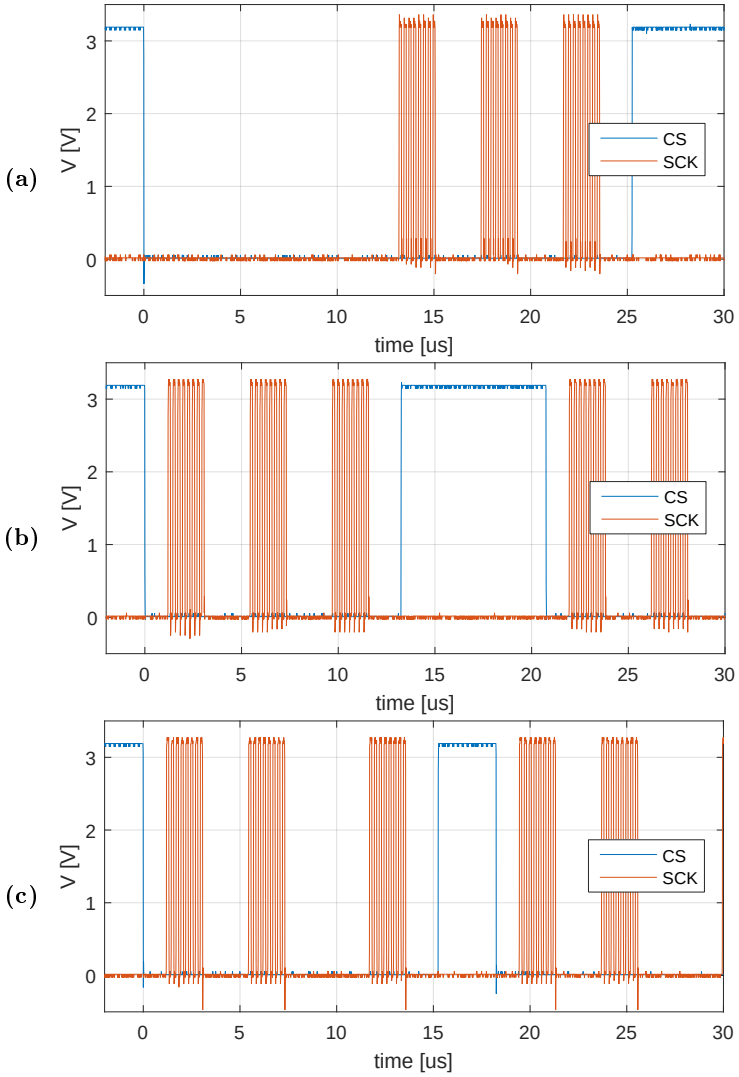

Figura 6.1: Trasmissione SPI per la scrittura del singolo byte: (a) utilizzando `SPI_write_long` senza preformattazione, (b) `SPI_write_long` con preformattazione degli indirizzi, (c) `fifoTX_put`.

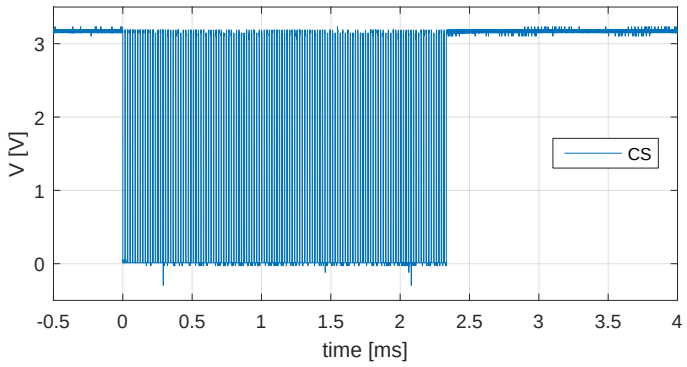

Figura 6.2: Trasmissione SPI per la scrittura dell'intero buffer.

6.2 Valutazione della latenza complessiva

Per la valutazione della latenza complessiva introdotto a dal sistema sono stati sfruttati i pin di I/O digitale, utilizzandoli come flag per l'oscilloscopio agli estremi di varie sezioni all'interno del programma.
Per valutare in modo migliore i tempi relativi alle operazioni svolte dal transceiver si è inserito uno shunt da 10 Ω alla scheda di sviluppo, per misurarne l'assorbimento di corrente. Il grafico in Figura 6.3 è derivato dalla combinazione delle due grandezze. La tensione sullo shunt viene riportata ingrandita di 40 volte per una maggiore comprensione.

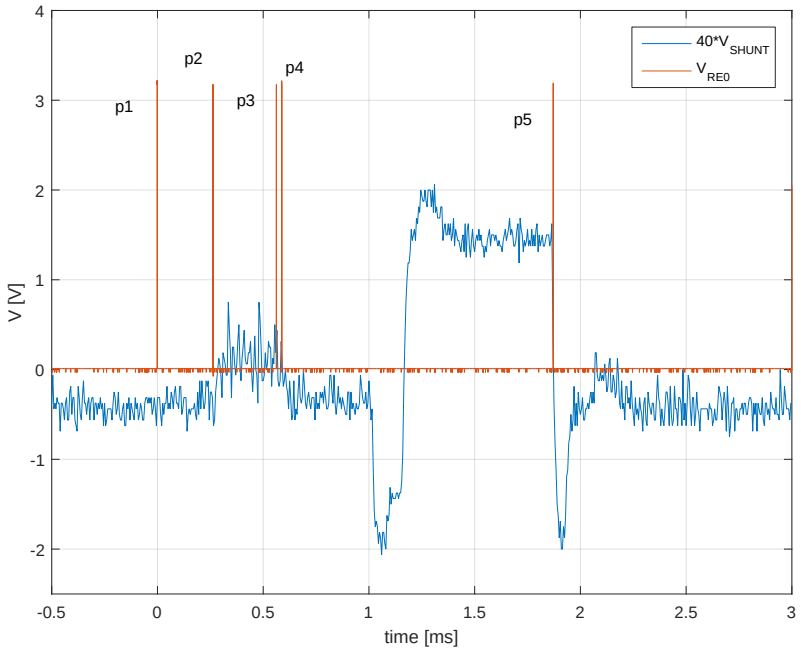

Figura 6.3: Assorbimento di potenza e principali eventi.

Facendo riferimento a questa figura, gli intervalli in essa contenuti rappresentano:

- p1: inizio del campionamento;
- p1-p2: creazione del pacchetto dati e memorizzazione nel buffer locale;
- p2-p3: caricamento del pacchetto nella memoria del transceiver;
- p3-p4: invio del trigger di trasmissione;
- p4-p5: trasmissione del pacchetto tramite interfaccia radio;
- p5: trasmissione conclusa.

Intervallo	Durata [μs]
p1–p2	260
p2–p3	300
p3–p4	26
p4–p5	1285

Tabella 6.1: Durata degli intervalli tra gli eventi relativi alla Figura 6.3.

Le prove sono state effettuate con un payload dati pari a 2 byte e incapsulati in un fame trasmesso utilizzando short address. Al momento dell'invio il transceiver aggiunge un

footer di 2 byte, un header di sincronizzazione di 5 byte ed un header fisico di un byte.
Il pacchetto fisico sarà quindi composto da 13+6+2 byte, che alla velocità di $250\,kb/s$ portano ad un intervallo temporale di

$$T_{pkt} = 672\,\mu s$$

valore del tutto compatibile con la durata della trasmissione ricavata dall'assorbimento di corrente, della durata di circa $690\,\mu s$.
Il fatto che il transceiver si attivi in ritardo di circa $500\,\mu s$ è dovuto al fatto che lo standard deve garantire un tempo minimo tra due frames (Interframe Spacing), che viene introdotto all'inizio della trasmissione.
Questo test è stato svolto sotto una forte limitazione, ovvero in una topologia peer-to-peer operante in un ambiente libero da interferenti esterne. Per valutare il limite applicativo l'accesso al canale di comunicazione è diretto, in quanto è stato disattivato l'algoritmo di CSMA-CA.
Con l'apparato sviluppato in questo modo ed utilizzando pacchetti di lunghezza ridotta, come si può notare, il limite superiore per il rate di pubblicazione di un pacchetto è dell'ordine di qualche centinaio di Hz.

Capitolo 7

Conclusioni

Nello svolgimento di questa tesi si è realizzato e validato sperimentalmente un sistema di test per la valutazione delle performance di reti senza fili in ambito biomedicale, afferenti alla famiglia delle cosiddette Wireless Body Sensor Networks. Si è ottimizzata e valutata la dinamica dell'interfaccia tra microcontrollore e transceiver, arrivando ad un punto accettabile per avere un servizio fruibile. Inoltre, è stata implementata una rete peer-to-peer per il trasferimento di dati ad un rate dell'ordine delle centinaia di Hz, sufficiente per la maggior parte delle applicazioni in ambito biomedicale.

Per quanto riguarda la topologia beacon-enabled, è possibile ottenere una comunicazione corretta, anche se si verificano problemi di stabilità nella pubblicazione del beacon. Per la generazione del *Beacon Interval* vengono utilizzati due timer, uno veloce ed uno lento (range timer). Quest'ultimo è derivato da un oscillatore RC, che è interno al transceiver stesso e la cui frequenza è nominalmente pari a 100 kHz. In realtà, l'oscillatore ha una frequenza molto inferiore, intorno a 60 kHz, ed è soggetto a derive termiche non trascurabili e rumore di fase (jitter), che rendono il beacon instabile.

Una possibile soluzione, ancora da testare, prevede che sia il microcontrollore stesso a forzare la pubblicazione del beacon, dal momento che dispone di un segnale di clock molto più stabile e indipendente da fattori esterni.

L'architettura proposta consente la riduzione dei costi di realizzazione, nonché un ridotto consumo energetico. Per questi motivi, tale architettura potrebbe essere applicata anche a sistemi cui sia richiesto una lunga vita operativa e la possibilità di integrare funzionalità di energy harvesting.

Tra i possibili ambiti applicativi, particolare interesse riveste l'ambiente biomedicale. In ambito ospedaliero, durante l'esecuzione di un esame, il numero elevato di sensori cablati possono diminuire il comfort percepito dal paziente, rischiando di falsare la diagnosi finale.

In tal senso, una possibile soluzione è la creazione di sistemi di misura distribuiti con interfaccia senza fili, in cui ogni nodo è connesso ad un'unica entità centrale in grado di raccogliere dati di origine e formato eterogenei. Inoltre, tali sistemi consentirebbero, laddove il monitoraggio non presenti particolari criticità dal punto di vista della sicurezza della persona, una certa libertà nei movimenti con conseguente incremento del benessere del paziente.

Bibliografia

[1] Ieee standard for information technology– local and metropolitan area networks– specific requirements– part 15.4: Wireless medium access control (mac) and physical layer (phy) specifications for low rate wireless personal area networks (wpans). *IEEE Std 802.15.4-2006 (Revision of IEEE Std 802.15.4-2003)*, pages 1–320, Sept 2006.

[2] Microchip Technology. *PICDEM^{TM} Z Demonstration Kit User's Guide.*

[3] Microchip Technology. *PIC18F4620 Data Sheet: 44-Pin Enhanced Flash Microcontrollers with 10-Bit A/D and nanoWatt Technology.*

[4] Microchip Technology. *MRF24J40 Data Sheet: IEEE 802.15.4^{TM} 2.4 GHz RF Transceiver.*

[5] Microchip Technology. *MRF24J40MA Data Sheet: 2.4 GHz IEEE Std. 802.15.4^{TM} RF Transceiver Module.*

[6] Microchip Technology. *MPLAB® XC8 C Compiler User's Guide.*