Università degli Studi di Padova

Facoltà di Ingegneria

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Elettronica

Tesi di Laurea

Realizzazione di un gateway TCP/IP – ZigBee, IEEE 1451 compliant, su sistema a microcontrollore a 16 bit

Relatore
Prof.ssa Giada Giorgi

Laureando
Roberto Guerra
n° matr. 562607/IL

Anno Accademico 2009/2010

UNIVERSITÀ DEGLI STUDI DI PADOVA

Corso di Laurea in Ingegneria Elettronica

Tesi di Laurea

Realizzazione di un gateway TCP/IP – ZigBee, IEEE 1451 compliant, su sistema a microcontrollore a 16 bit

Relatore
Prof.ssa Giada Giorgi

Laureando
Roberto Guerra
n° matr. 562607/IL

Padova, 21 Luglio 2010
Anno Accademico 2009/2010

Indice

Introduzione

Questo lavoro di tesi si colloca all'interno di un progetto più ampio riguardante la realizzazione di una rete di smart transducers compatibile con lo standard IEEE 1451. Una rete di trasduttori intelligenti integra funzionalità di trasduttori classici, unità di elaborazione ed interfaccia di comunicazione, al fine di collezionare, disseminare e processare informazioni relative all'ambiente fisico circostante.

Il sistema è implementato su microcontrollori a 8 e 16 bit di Microchip Technology Inc. per mezzo di strumenti hardware e sofware forniti dal medesimo produttore. La rete si compone di diversi nodi detti WTIM implementati sulle schede di sviluppo PICDEMZ. Queste schede comunicano con il nodo principale della rete, l'NCAP implementato sulla scheda di sviluppo Explorer 16, per mezzo del protocollo wireless ZigBee, formando così una rete a stella.

Il lavoro descritto in questa tesi è rivolto alla realizzazione del nodo NCAP, ovvero il nodo che si occupa di creare e coordinare la rete di sensori e di implementare un'interfaccia di comunicazione (TCP/IP) che permetta alla rete di interagire all'interno di una rete più ampia quale Internet. A tal fine, come verrà discusso in seguito, si sono dovuti individuare e risolvere i vari problemi di incompatibilità e coesistenza tra i due stack protocollari TCP/IP e ZigBee.

Cap. 1: Lo standard IEEE 1451

1.1 – Introduzione

L'interfacciamento di trasduttori, per garantire la loro interoperabilità a livello di rete per ogni nodo, richiede che vi sia un protocollo di comunicazione comune per gestire i dati provenienti dagli stessi, facilitandone le operazioni di controllo e di configurazione. Poiché la rete e i sensori espongono all'utente e agli altri nodi la loro interfaccia, qualsiasi modifica all'hardware dell'elemento sensibile o della piattaforma di rete, se non esistesse una direttiva standard, comporterebbe notevoli costi in termini economici e temporali. Queste considerazioni hanno dato vita alla famiglia di standard IEEE 1451, un insieme di documenti elaborati negli ultimi dieci anni dall'Institute of Electric and Electronic Engineers, che prevede la definizione di un'interfaccia standard per una rete di sensori intelligenti. Questa famiglia di standard definisce un'architettura di base della rete, che consente di modificarne la configurazione in modalità plug and play.

Una volta definite le caratteristiche principali delle reti 1451, l'IEEE ha definito in maniera più dettagliata le caratteristiche dei singoli nodi, classificandoli in due distinte categorie:

- TIM (Transducer Interface Module): dispositivi a cui spetta la gestione del sensore/trasduttore/attuatore, il condizionamento dell'informazione rilevata, la conversione di tali informazioni, la trasmissione delle stesse; sono considerati dei nodi terminali di rete e si appoggiano ai nodi della seconda categoria: gli NCAP.
- NCAP (Network Capable Application Processor): dispositivi ai quali spetta la gestione della rete, e l'inoltro di messaggi che circolano in essa. Tali dispositivi operano da Gateway tra gli utenti della rete ed i TIM, nel caso fosse previsto l'accesso da esterno.

Questa famiglia non definisce l'interfaccia fisica tra l'NCAP e la rete, ma i tipi di interfaccia che possono esistere tra l'NCAP e i TIM.

Lo standard IEEE 1451 è suddiviso in protocolli, ognuno dei quali ne definisce un aspetto. In questo capitolo verranno presentati ed analizzati i protocolli che sono stati adottati per la realizzazione pratica del progetto.

Per comprendere l'organizzazione dello standard e dei suoi diversi protocolli è utile osservare la figura 1.1.1. La trattazione riguarderà solamente gli aspetti più rilevanti di ogni direttiva, ovvero quelli concernenti il progetto, poiché si tratta di standard complessi che interessano diverse tipologie di interfacce fisiche ed offrono talvolta funzionalità molto avanzate, non sviluppate nel progetto.

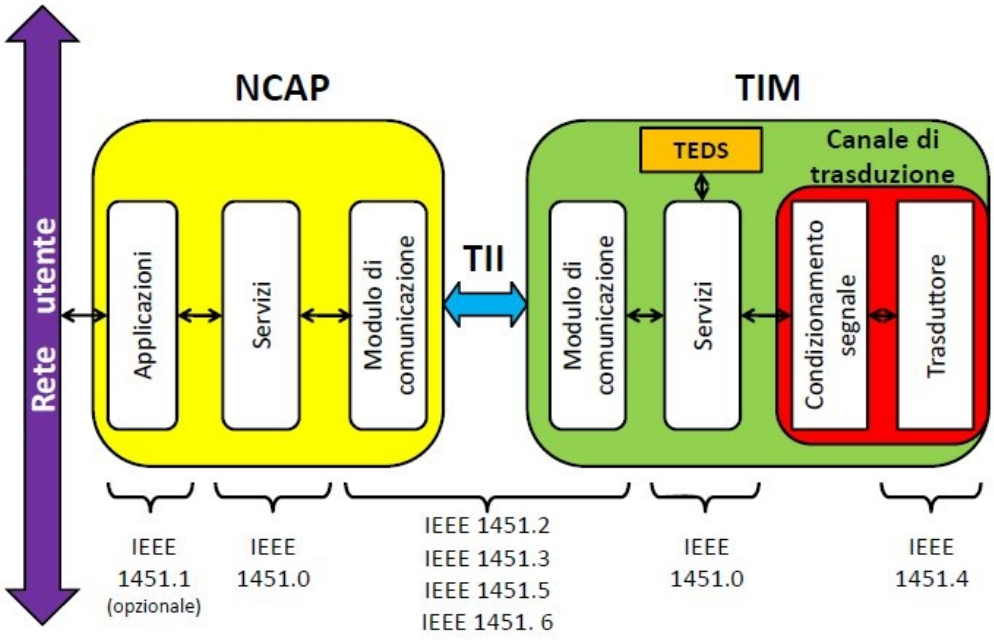

Fig. 1.1.1: Struttura di un sistema con un singolo TIM.

A partire da sinistra, ovvero dall'NCAP, si incontra prima l'applicazione lato utente, la quale si interfaccia al livello servizi definito dal 1451.0. Il livello servizi si interfaccia a sua volta con il livello di comunicazione

il quale ha il compito di interfacciare il nodo NCAP con i nodi TIM secondo un'interfaccia detta Transducer Independent Interface (TII) che implementa uno dei protocolli 1451.2, 1451.3, 1451.5 e 1451.6.

Proseguendo sempre verso destra, in maniera speculare, si incontrano il modulo di comunicazione lato TIM, i servizi lato TIM, i quali includono i TEDS, ed il canale di trasduzione, fisicamente connesso al trasduttore il quale può implementare il protocollo 1451.4.

Gli interfacciamenti tra i diversi protocolli all'interno di un singolo nodo avvengono tramite delle Application Process Interface (API). In particolare le interfacce tra livello applicazioni e servizi sono dette Application API (AAPI), mentre le interfacce tra livello servizi e modulo di comunicazione sono dette Communication API (CAPI).

1.2 – Il protocollo IEEE 1451.5

1.2.1 – Introduzione

A seconda del protocollo implementato dalla Transducer Independent Interface (TII) si definisce un tipo di comunicazione tra il nodo NCAP e il TIM, in accordo a quanto riportato in figura 1.2.1.

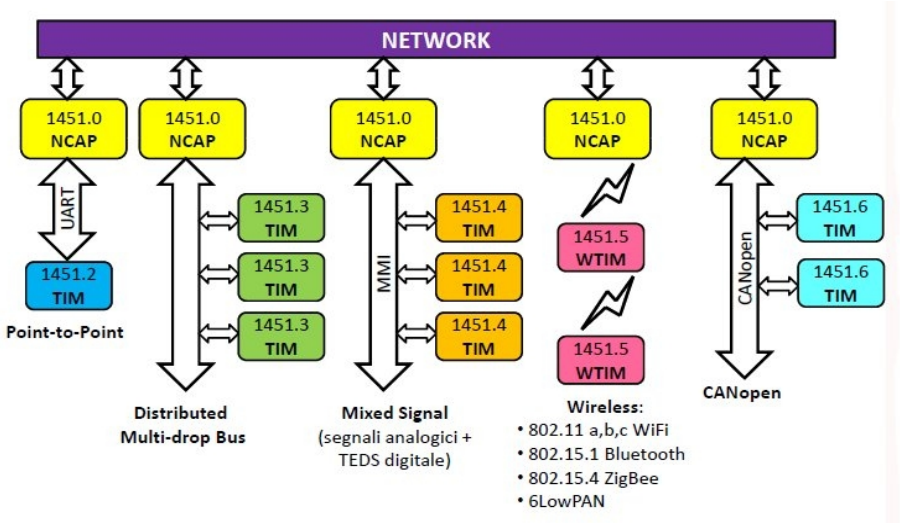

Fig. 1.2.1: Standard per l'interfaccia TII

Il protocollo implementato nel progetto in questione è il 1451.5, il quale introduce il concetto di Wireless Interface Transducer Module (WTIM) connesso ad un Network-Capable Application Processor (NCAP) Service Module attraverso un canale radio approvato tra IEEE 802.11, IEEE 802.15.4 e ZigBee, IEEE 802.15.1 e Bluetooth.

Secondo le direttive 1451.5, un WTIM è un dispositivo che comprende una Dot 5 Approved Radio (Dot5AR), il condizionamento del segnale, la conversione analogica/digitale e uno o più trasduttori (sensori/attuatori).

Poiché i WTIM possono avere interfacce Dot5AR diverse tra loro, l'NCAP dovrà avere almeno una Dot5AR per ogni tipo presente nei WTIM a cui deve essere associato.

Questo standard si focalizza sull'interfaccia di comunicazione tra WTIM ed NCAP attraverso i protocolli Dot5AR, stabilendo di fatto i metodi e i formati di dati necessari a governare i trasduttori, per le operazioni di rete e per i TEDS .

Lo standard in questione è basato unicamente su interfacce wireless, ma non specifica le caratteristiche fisiche e tecniche né dei trasduttori né del sistema wireless.

La nascita di questo protocollo è dovuta alla volontà di uniformare le specifiche ed accomunare più tecnologie sotto un unico standard aperto, riducendo al minimo il rischio di incompatibilità e i costi di produzione.

Le specifiche di IEEE 1451.5 riguardanti ZigBee indicano i requisiti che una rete di tale tipo deve avere affinché possa fungere da rete di trasporto per un sistema compatibile con IEEE 1451.

1.2.2 - Convergence Layer

Lo standard IEEE 1451.5 definisce il concetto di convergence layer (in italiano strato di convergenza) tra l'entità superiore (IEEE 1451.0) e la

rete di trasporto (in questo caso ZigBee, Fig. 1.2.2). Esso ha infatti il compito di tradurre i comandi provenienti dal livello superiore in comandi specifici comprensibili al livello inferiore e viceversa.

Nella documentazione del protocollo 1451.5 sono elencati solamente i metodi della Communication API tra 1451.5 e 1451.0 (appartenenti alla MCI del 1451.0). Per ciascuno di essi, assieme ad una descrizione sintetica che ne spiega l'utilizzo in ambito 1451.5, vengono elencati nome, tipi di dato e parametri; questi metodi costituiscono il confine superiore del convergence layer.

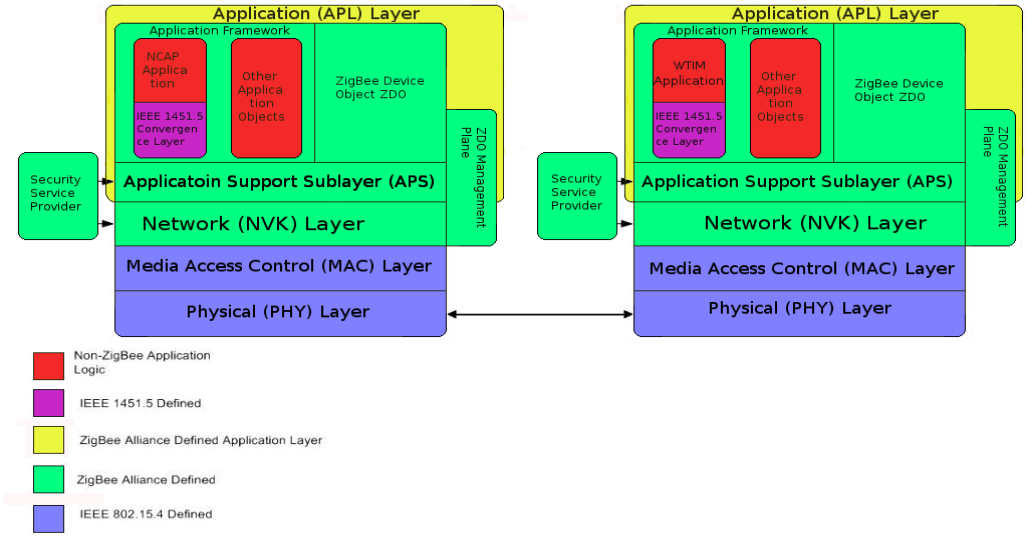

Fig. 1.2.2: Modello dello stack protocollare ZigBee.

1.2.3 – Regole generali

- Un NCAP può instradare dati sia verso una rete esterna, sia verso un trasduttore collegato ad un WTIM;
- un NCAP può avere più WTIM associati;
- un NCAP può avere più interfacce radio (anche diverse);
- l'interfaccia ZigBee è gestita dal protocollo IEEE 1451.5 sia per gli NCAP che per i WTIM;

- un WTIM può associarsi ad un solo NCAP;
- un WTIM può interfacciarsi a più trasduttori;
- è ammessa la comunicazione tra due WTIM.

Nota: Nel caso di questo progetto diversi di questi aspetti non sono approfonditi, poiché ci si occupa della comunicazione tra un NCAP e un singolo WTIM (Figura 1.2.3).

Fig. 1.2.3: Connessione di un NCAP e un WTIM

1.2.4 – Diagramma di stato del NCAP

Lo stato iniziale di un NCAP dopo l'accensione o dopo il reset è UNREGISTERED. Dopo un processo di registrazione del modulo 1451.5, l'NCAP passa allo stato DOT5REGISTERED per l'entità DOT5AR del protocollo 1451.5, e vi rimane finché non vi si associa alcun WTIM.

L'NCAP ha il compito di mantenere, separatamente, lo stato di ognuna delle sue entità 1451.5, le quali a loro volta, devono mantenere separatamente gli stati di tutti i loro WTIM associati.

Quando un'entità 1451.5 registra uno o più WTIM, l'NCAP passa allo

stato TIM-REGISTERED per tale modulo.

Prima di iniziare uno scambio dati con un WTIM, l'NCAP deve invocare un comando di tipo "open", con il quale passa appunto allo stato OPEN per tale WTIM e vi rimane fino al primo comando di tipo "close", attraverso il quale ritorna allo stato TIMREGISTERED.

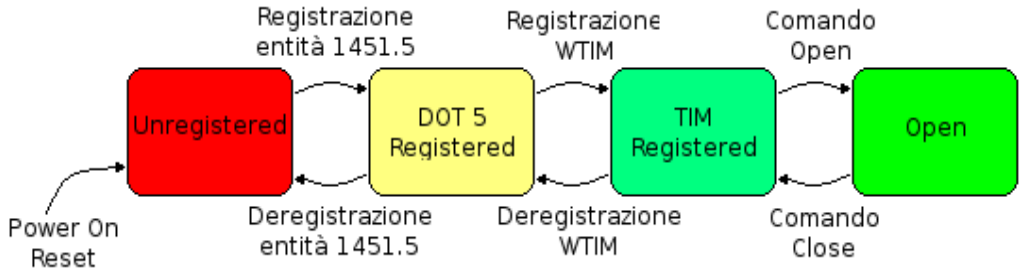

Fig. 1.2.4: Diagramma degli stati del nodo NCAP

Il vantaggio di mantenere separati gli stati per ogni entità 1451.5 e per ogni WTIM consiste nel fatto che esso può iniziare in qualsiasi momento la ricerca di altri dispositivi e l'attesa di nuove connessioni.

1.2.5 – Diagramma di stato del WTIM

Lo stato iniziale di un WTIM dopo l'accensione o dopo il reset è UNREGISTERED. Dopo il processo di registrazione con un NCAP, il WTIM passa allo stato REGISTERED ma non può comunicare fintantoché non invoca un comando di tipo "open", con il quale passa appunto allo stato OPEN e vi rimane fino al primo comando di tipo

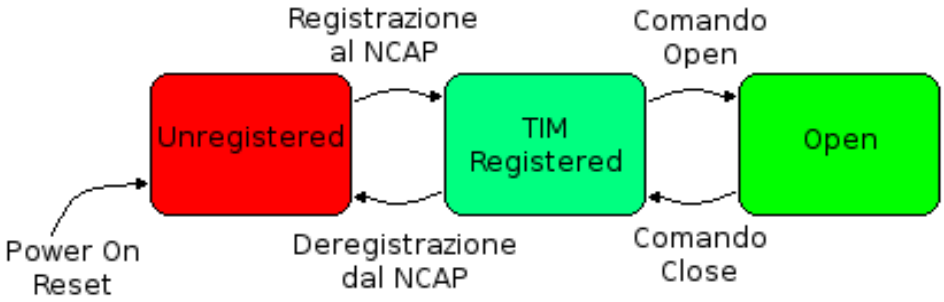

Fig. 1.2.5: Diagramma di stato del WTIM

"close", attraverso il quale ritorna allo stato REGISTERED.

1.3 – Il protocollo IEEE 1451.0

1.3.1 – Introduzione

Lo standard IEEE 1451.0 si posiziona, nello stack protocollare, tra l'applicazione ed il protocollo di comunicazione scelto in base all'interfaccia fisica utilizzata, fungendo quindi a sua volta da convergence layer tra l'applicazione e la rete.

Ovviamente anche questo protocollo riconosce i due dispositivi previsti dalla famiglia IEEE 1451 (NCAP e TIM) e si occupa di descriverne tutte le caratteristiche comuni, nonché le operazioni e le funzionalità che caratterizzano i dispositivi di una rete di sensori intelligenti, indipendentemente dal tipo di interfaccia fisica prescelta.

1.3.2 – Conformità

Per ottenere la dicitura plug-and-play, un dispositivo di questo tipo deve rispondere a determinate caratteristiche richieste a livello applicazione.

Esse sono tutte specificate nella documentazione dello standard, ma in particolare si ricorda che sia gli NCAP che i TIM funzionanti secondo lo standard IEEE 1451.0 devono supportare un protocollo di comunicazione e un mezzo fisico definiti nella stessa famiglia di standard IEEE 1451.

1.3.3 – Rete

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

1.3.4 – Indirizzamento

Esistono due tipi di indirizzamento definiti da questo standard. Il primo è di tipo fisico, del quale però se ne occupa appunto il physical layer, tramite il parametro destId (così riconosciuto dal Module Communication Interface). Il secondo parametro per l'indirizzamento viene chiamato TransducerChannelNumber.

Nota: nel progetto in questione, il TransducerChannelNumber non verrà considerato.

1.3.5 – Api

Lo standard definisce una application program interface (API) per tutte le applicazioni che provvedono alla comunicazione tra la rete e il layer IEEE 1451 e alla comunicazione tra il protocollo IEEE 1451.0 e i sottostanti layer fisici di comunicazione, di solito denominati in questo standard come layer IEEE 1451.X.

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

Questa API contiene i metodi per leggere e scrivere i TEDS, per gestire i TransducerChannels, per inviare comandi di configurazione e controllo ai TIM.

L'altra API, la Module Communication Interface, si colloca tra lo standard 1451.0 e un altro membro della famiglia 1451.

Essa è un'interfaccia simmetrica che va implementata sia sul NCAP che sul TIM.

Questa API contiene metodi che dovrebbero essere implementati dal layer IEEE 1451.X e chiamati per iniziare le operazioni di comunicazione.

Similarmente ci sono metodi che devono essere implementati dal 1451.0 e invocati dal 1451.X per consegnare le informazioni inviate.

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

Nota: in questo progetto l'interfaccia Transducer Service non verrà analizzata; verrà implementata solamente parte dell'interfaccia Module Communications, che costituirà il confine superiore del convergence layer, ossia l'interfaccia di comunicazione tra i livelli 1451.0 e 1451.5. Le funzioni di interfacciamento della Module Communication sono suddivise in tre gruppi (registration, communication, receive) a seconda dei compiti da esse svolti. Per ognuna delle categorie elencate esiste poi un'ulteriore suddivisione tra funzioni per reti generiche e funzioni per reti point-to-point; in questo progetto verranno utilizzate solamente le seconde.

Cap. 2: Lo Standard ZigBee

2.1 – Introduzione

ZigBee è il nome di un insieme di protocolli di comunicazione ad alto livello che utilizzano piccole antenne digitali a bassa potenza, basato sullo standard IEEE 802.15.4 per Wireless Personal Area Networks (WPAN).

ZigBee opera nelle frequenze radio assegnate per scopi industriali, scientifici e medici (ISM).

Questa tecnologia ha lo scopo di essere più semplice e più economica di altre WPAN come, ad esempio, Bluetooth.

2.2 – Hardware e Software

Il protocollo ZigBee è stato ideato per rendere facile la sua implementazione su microprocessori a basso costo che offrono solamente le caratteristiche minime ed essenziali. Il progetto dei radiotrasmettitori è stato ottimizzato per avere un basso costo unitario con produzioni su larga scala. Essi hanno poca circuiteria analogica e utilizzano il digitale ovunque sia possibile. Per esempio, si dice che un nodo ZigBee del tipo più complesso richieda solamente il 10% del codice necessario per un tipico nodo Bluetooth o Wi-Fi, mentre il più semplice dovrebbe richiederne intorno al 2%.

Anche se i radiotrasmettitori sono economici, il ZigBee Qualification Process comporta una validazione completa delle richieste del livello fisico. Questa minuziosa analisi del livello fisico ha molti vantaggi, poiché tutti i trasmettitori derivanti da uno stesso set di semiconduttori avranno le stesse caratteristiche RF. D'altra parte un livello fisico non certificato che presenta malfunzionamenti potrebbe influenzare negativamente le capacità di una rete ZigBee. In questo caso, infatti, i

vincoli ingegneristici sono necessariamente stretti, in termini di banda e di consumo di energia. Per tale motivo i trasmettitori sono testati secondo lo standard ISO-17025 e la Clause 6 dello standard IEEE 802.15.4-2003.

Attualmente gran parte dei produttori integra radiotrasmettitori e microcontrollori su un singolo chip.

2.3 – Tipi di Dispositivi

Le specifiche IEEE 802.15.4 definiscono due tipi di dispositivi:

- Full Function Device (FFD): offre piena funzionalità, è solitamente alimentato da rete e rimane acceso quando inattivo;
- Reduced Function Device (RFD): offre funzionalità limitata, è solitamente alimentato da batterie e rimane spento quando inattivo.

Le specifiche ZigBee definiscono invece tre tipi di dispositivi:

- ZigBee Coordinator (ZC): è il dispositivo più intelligente tra quelli disponibili, costituisce la radice di una rete ZigBee e può operare da ponte tra più reti. Ci può essere un solo Coordinator in ogni rete. Esso è inoltre in grado di memorizzare informazioni riguardanti la propria rete e può agire da deposito per le chiavi di sicurezza.
- ZigBee Router (ZR): questo dispositivo agisce come router intermedio passando i dati da e verso altri dispositivi.
- ZigBee End Device (ZED): include solo le funzionalità minime per dialogare con il suo nodo parente (Coordinator o Router), non può trasmettere dati provenienti da altri dispositivi; è il nodo che richiede il minor quantitativo di memoria e quindi risulta spesso più economico rispetto ai ZR o ai ZC.

Esiste inoltre una quarta categoria di dispositivi, che racchiude in sé alcune funzionalità miste delle altre tre categorie. Essi sono gli ZigBee Device Object (ZDO): dispositivi speciali che all'interno della rete eseguono funzionalità particolari quali, ad esempio, la scoperta di nuovi dispositivi e la gestione della sicurezza.

Basandosi sulle definizioni sopra illustrate, il protocollo ZigBee classifica così i suoi tre tipi di dispositivi:

- **Coordinator**: di tipo FFD, unico all'interno della rete; forma la rete e la gestisce, alloca e memorizza gli indirizzi dei nodi collegati.
- **Router**: di tipo FFD, opzionale, estende la rete dando la possibilità di aumentare il numero di nodi e di trasferire dati tra di essi; può esercitare funzioni di controllo e monitoraggio.
- **End Device**: tipo FFD o RFD, esercita solo funzioni di controllo e/o monitoraggio.

Nota: in questo specifico progetto l'NCAP, dovendo occuparsi della creazione, gestione e manutenzione della rete sarà quindi costituito da un nodo di tipo ZigBee Coordinator (ZC), mentre il WTIM sarà costituito da un nodo ZigBee End Device (ZED), limitato alla sola trasmissione dei dati provenienti dalla lettura del proprio sensore.

2.4 – Tipi di Rete

Una rete wireless ZigBee può assumere diverse configurazioni, ma in ogni rete, ci devono essere almeno due componenti fondamentali: un nodo Coordinator ed un nodo End Device. Il Router invece è presente solo in alcuni tipi di rete.

Le tre tipologie di reti realizzabili sono: rete a stella (star network), rete ad albero (cluster tree network) e rete a maglia (mesh network).

Una rete a stella è formata da un coordinatore e da uno o più nodi

terminali i quali non possono comunicare tra di loro, ma solo con il coordinatore.

Nota: la rete di tipo a stella è la più adatta al presente progetto (ci si occupa solo della comunicazione tra coordinator e un end device), pertanto gli altri tipi di rete non verranno considerati.

2.5 – Indirizzamento

Ogni nodo ZigBee ha, a livello di rete, due indirizzi: un MAC address a 64 bit (dei quali 24 bit identificano il produttore) ed un network address di 16 bit.

Per stabilire una connessione con una nuova rete, il nodo utilizza l'indirizzo MAC, dopodiché, una volta connesso, verrà identificato nella rete attraverso il suo network address e tramite esso potrà comunicare con gli altri dispositivi.

Per la messaggistica di tipo unicast è utilizzato il MAC address specifico, mentre per il broadcast (impiegato nelle operazioni di gestione della rete quali creazione, connessione ecc.) si utilizza il MAC address generico 0xFFFF.

Nota: Questo progetto prevede la sola comunicazione tra Coordinator ed End Device, pertanto la messaggistica è di tipo unicast (Figura 2.5.1).

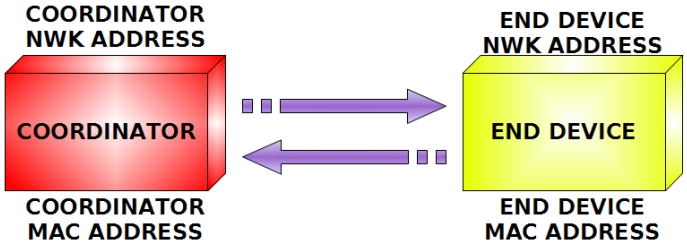

Fig. 2.5.1: Connessione unicast

2.6 – Sincronizzazione

In una rete di tipo non-beacon un dispositivo che vuole inviare un messaggio deve semplicemente attendere che il canale sia libero, dopodiché può iniziare la trasmissione.

Se il dispositivo di destinazione è di tipo FFD il suo ricetrasmettitore sarà sempre acceso e il messaggio verrà ricevuto immediatamente.

Diversamente un RFD potrà avere il ricetrasmettitore spento (in modalità risparmio energetico) al momento della ricezione, per cui, una volta acceso il ricetrasmettitore, dovrà appoggiarsi al suo FFD associato (genitore) richiedendo l'inoltro dei messaggi non ricevuti, il quale li avrà temporaneamente memorizzati in un buffer.

Tale caratteristica permette al RFD di risparmiare energia ma, al contempo, richiede al FFD una sufficiente quantità di memoria libera. Se dopo un certo tempo, chiamato macTransactionPersistenceTime, il RFD non ha richiesto al suo FFD genitore l'inoltro dei messaggi ad esso riservati, essi verranno irreversibilmente scartati.

Nota: Questo progetto prevede che la rete sia di tipo non-beacon enable, pertanto non si presterà particolare attenzione alla sincronizzazione.

2.7 – Profili

Un profilo ZigBee è una semplice descrizione dei componenti logici e delle loro interfacce. I dati da scambiare, per esempio le letture o le tarature dei sensori, sono chiamati attributi e ad ognuno di essi è associato un identificatore univoco.

Gli attributi sono raggruppati in cluster ai quali, a loro volta, è associato un altro identificatore univoco. Di conseguenza le interfacce sono specificate a livello di cluster. I profili inoltre possono specificare quali cluster sono obbligatori.

Ogni blocco funzionale che supporta uno o più cluster viene definito endpoint, pertanto dispositivi diversi possono comunicare tra loro facendo ricorso agli stessi endpoint (se implementati).

2.8 – Messaggistica

Il protocollo ZigBee definisce due formati per i frame: il formato Key–Value–Pair (KVP) e il formato Messagge (MSG). Entrambi i formati sono associati ad un Cluster ID, ma il KVP rispetta una struttura definita, a differenza del MSG che non è vincolato da alcuna struttura. Il profilo in uso a livello applicazione specifica il formato dei messaggi supportato, ma non è permesso ai cluster di utilizzare entrambi i formati.

2.8.1 – Struttura del frame KVP

- Transaction Count
- Frame Type
- Transaction
  - Transaction Sequence Number
  - Command Type and Attribute Data Type
  - Attribute ID
  - Error Code (optional)
  - Data (variable length)

2.8.2 – Struttura del frame MSG

- Transaction Count
- Frame Type
- Transaction

- Transaction Sequence Number
- Transaction Length
- Data (variable length)

2.9 – Binding

I dispositivi ZigBee possono comunicare tra di loro se conoscono i rispettivi indirizzi di rete (messaggistica diretta). La scoperta e il mantenimento di questi indirizzi comporta un notevole dispendio di risorse e un alto overhead. Il protocollo ZigBee, per ovviare a tale problema, offre una caratteristica chiamata binding: il coordinatore memorizza una tabella di corrispondenze tra dispositivi e cluster/endpoint relativi. Una volta creati tutti i bind necessari, i dispositivi possono comunicare tra di loro attraverso il coordinatore, che detiene le corrispondenze. Tale tipo di messaggistica prende il nome di messaggistica indiretta ed è utile nelle reti con più end device.

Nota: per lo sviluppo di questo progetto il binding è non influente ed, oltretutto, richiederebbe una gestione efficiente dell'allocazione dinamica di memoria; pertanto non verrà implementato.

2.10 – Routing

I protocolli di routing si basano sugli algoritmi di tipo ad-hoc On-demand Distance Vector che puntano a costruire delle reti ad-hoc di nodi a bassa velocità.

I profili correnti derivati dai protocolli ZigBee supportano sia reti beacon enabled che reti non-beacon enabled.

Nelle reti non-beacon enabled (quelle il cui beacon order è 15), viene utilizzato un meccanismo di accesso al canale di tipo CSMA/CA.

In questo tipo di reti gli ZigBee Router e gli ZigBee Coordinator solitamente tengono i loro ricevitori sempre attivi, il ché provoca un

considerevole consumo di energia; in pratica queste reti sono miste: alcuni dispositivi sono costantemente pronti a ricevere, mentre altri si limitano a trasmettere in presenza di uno stimolo esterno. In questo tipo di reti, quindi, alcuni nodi sono sempre attivi (il loro consumo di energia è quindi alto) ed altri sono per la maggior parte del tempo spenti.

In generale, i protocolli ZigBee minimizzano il tempo di attività del radiotrasmettitore, al fine di ridurre il consumo di energia.

Nelle reti beacon enabled i nodi consumano energia solo nel periodo in cui c'è il beacon.

Nota: questo progetto prevede che l'NCAP rimanga sempre attivo e pronto a richiedere le letture al WTIM, che terrà sempre acceso il suo ricetrasmettitore (rete non- beacon enabled). Il routing in una rete ZigBee è gestito autonomamente dallo stack e non necessita di interventi dal livello applicazione. In ogni caso, nell'ambito di questo progetto risulta ininfluente e pertanto non verrà considerato.

2.11 - Operazioni di Rete

La rete ZigBee può essere creata solo da un coordinatore. All'accensione, esso cerca altri coordinatori tra i suoi canali a disposizione. Il tempo che un dispositivo impiega per la scansione delle reti disponibili e per la determinazione dell'energia di ogni canale è definito dal parametro ScanDuration. Per la banda 2.4 GHz, il tempo di scansione (in secondi) è determinato dalla seguente equazione:

$$0.01536 \cdot (2 \cdot ScanDuration + 1)$$

I router e gli end device eseguono una sola scansione per determinare le reti disponibili, mentre i coordinatori eseguono due scansioni, una per testare l'energia del canale e una per determinare le reti esistenti.

In base all'energia del canale e al numero di reti presenti in tali canali, il coordinatore stabilisce la propria rete e le attribuisce un PAN

ID a 16 bit univoco.

Da questo momento in poi router ed end device possono unirsi alla rete. In caso di conflitto causato da PAN ID uguali, uno dei due coordinatori avvia una procedura di risoluzione del conflitto, che però non è supportata dallo stack Microchip utilizzato in questo progetto.

I dispositivi ZigBee salvano nella memoria non volatile le informazioni relative ai nodi genitori e figli in una tabella chiamata tabella dei vicini. All'accensione, un nodo può determinare se faceva precedentemente parte di una rete e, in tal caso, avvia una procedura per ricongiungersi ad essa come nodo orfano.

I nodi che ricevono tale richiesta verificano se l'orfano era proprio figlio e, in caso affermativo, comunicano la loro posizione all'interno della rete; altrimenti, se tale procedura fallisce oppure il nodo orfano non ha nessun genitore memorizzato nella propria tabella dei vicini, tenterà di connettersi alla rete come nuovo nodo, generando una lista di potenziali genitori e determinando la posizione migliore (in termini di distanza dal coordinatore). Una volta nella rete, un dispositivo può abbandonarla sia su richiesta del proprio genitore sia autonomamente.

2.12 – Caratteristiche Tecniche

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

2.13 – Phy

Il protocollo ZigBee opera in banda non licenziata a 2.4 GHz, 915 MHz e 868 MHz. Nella banda 2.4 GHz ci sono 16 canali ZigBee, da 3 MHz ciascuno.

I trasmettitori radio usano una codifica DSSS. Il data rate over-the-air

è di 250 kb/s per canale nella banda 2.4 GHz, 40 kb/s per canale nella banda 915 MHz e 20 kb/s nella banda 868 MHz.

Il range di funzionamento è compreso tra 10 e 75 metri, dipendentemente dall'ambiente circostante.

La massima potenza trasmessa è in genere 0 dBm (1 mW).

2.14 - Mac

La modalità base di accesso al canale, specificata da IEEE 802.15.4-2003, è il Carrier Sense Multiple Access/Collision Avoidance (CSMA/CA). Questo significa che i nodi controllano se il canale è libero quando devono trasmettere. Vi sono alcune eccezioni all'uso del CSMA: i segnali di beacon, inviati secondo uno schema prefissato, i messaggi di acknowledge e le trasmissioni di dispositivi in reti beacon-oriented, che hanno necessità di bassa latenza ed usano Guaranteed Time Slots (GTS) che per definizione non fa uso di CSMA.

La lunghezza massima dei pacchetti MAC definiti dal IEEE 802.15.4 è di 127 byte, compreso un campo CRC a 16 bit per il controllo dell'integrità del frame; inoltre lo standard IEEE 802.15.4 prevede l'uso (opzionale) di un meccanismo di acknowledge tramite l'impostazione di un flag di ACK all'interno dei frame inviati.

Un messaggio ZigBee può essere quindi formato al più da 127 byte, così suddivisi:

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

  Nota: tale header è trasparente al livello applicazione, pertanto esso non verrà interessato da questo progetto.

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

- Application Support Sub-Layer (APS) header: contiene informazioni riguardo al profilo, al cluster e all'endpoint di destinazione.
- APS payload: dati utili, la cui gestione spetta al livello applicazione.

2.15 - Stack ZigBee di Microchip

Lo stack ZigBee di Microchip trova il suo impiego per tutte le applicazioni costituenti reti wireless a basso bit-rate. Esso comprende le funzioni di creazione e di gestione della rete, messaggistica e ricerca di dispositivi rispettando le regole dello standard ZigBee. Tali funzioni prevedono l'utilizzo di un apposito modulo hardware contenente tutta l'elettronica necessaria alla realizzazione della comunicazione, il quale si interfaccia al microcontrollore per mezzo del protocollo di comunicazione SPI.

Lo stack è implementabile sia sui PIC18 sia sui PIC24 ed è caratterizzato da un'elevata flessibilità, in quanto permette all'utente di configurare diversi parametri in modo semplice ed immediato, per mezzo delle macro di configurazione presenti nei file di intestazione.

Cap. 3: L'architettura TCP/IP

3.1 – Introduzione

L'architettura TCP/IP, detta anche architettura Internet, si è evoluta a partire dall'esperienza di una precedente rete a commutazione di pacchetto chiamata ARPANET, entrambe fondate da ARPA (Advanced Research Projects Agency): un'agenzia del Dipartimento della Difesa degli Stati Uniti d'America che si occupa di finanziare la ricerca e lo sviluppo. L'obiettivo dell'architettura TCP/IP è quello di consentire l'interconnessione di reti di natura eterogenea.

3.2 – Elementi Hardware elementari

Le reti di calcolatori contengono due categorie di elementi hardware elementari: nodi e linee di collegamento (link).

3.2.1 – Nodi

I nodi sono solitamente calcolatori a utilizzo generico, come una stazione di lavoro da scrivania, un calcolatore, un PC, oppure, nel caso del presente progetto il nodo NCAP di una rete di sensori compatibile allo standard IEEE 1451. In una rete di calcolatori esistono due tipi di nodi:

- Calcolatori general purpose: ad esempio PC o workstation
- Hardware con funzioni specifiche: ne fanno parte i router e gli switch, che sono nodi in grado di inoltrare pacchetti IP destinati ad altri nodi.

3.2.2 – Linee di collegamento (link)

Le linee di collegamento di una rete sono realizzate con molti mezzi

fisici diversi, tra i quali i cavi in doppino, i cavi coassiali, le fibre ottiche e l'etere. Qualunque sia il mezzo fisico, esso viene usato per propagare segnali.

3.3 – Modello a strati

All'aumentare della complessità dei sistemi, i progettisti solitamente introducono un nuovo livello di astrazione. Il concetto di astrazione consiste nella definizione di un modello unificante che sia in grado di cogliere alcuni aspetti importanti del sistema, incapsulare tale modello in un oggetto dotato di un'interfaccia che possa essere manipolata dagli altri componenti del sistema e nascondere agli utenti degli oggetti i dettagli di come essi siano realizzati.

Il modello che consente tale unificazione è il modello a strati ISO/OSI, che suddivide i vari protocolli in strati di appartenenza, a seconda delle funzionalità da essi implementate. Un esempio semplificato di tale stack protocollare è schematizzato in Fig. 3.3.1.

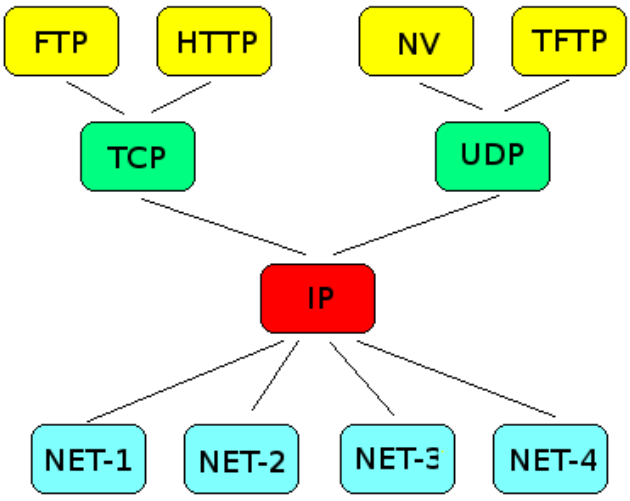

Fig. 3.3.1: Il grafo dei protocolli Internet

Al livello più basso sono presenti i protocolli di rete, denominati NET-1, NET-2 e così via. In questo strato, ad esempio, si trovano i protocolli Ethernet e FDDI: tali protocolli a loro volta, possono in realtà essere costituiti da parecchi sottolivelli, ma l'architettura Internet non fa nessuna ipotesi in merito. Questi protocolli permettono di gestire la comunicazione attraverso il particolare mezzo fisico adottato.

Il secondo strato è formato dal solo protocollo IP (Internet Protocol), il quale fornisce il supporto per l'interconnessione di più tecnologie di rete per formare, dal punto di vista logico, un'unica internetwork.

Il terzo strato contiene due protocolli principali, TCP (Transmission Control Protocol) e UDP (User Datagram Protocol), i quali forniscono due canali logici alternativi per i programmi applicativi: rispettivamente uno affidabile l'altro non affidabile.

Al di sopra dello strato di trasporto si trova un insieme di protocolli applicativi, come FTP, TFTP, Telnet ed SMTP, i quali consentono l'esecuzione delle applicazioni più diffuse.

L'architettura Internet ha due caratteristiche fondamentali:

- Non implica una stratificazione rigida, ovvero, un'applicazione può liberamente scavalcare lo strato di trasporto e usare direttamente IP o una delle reti sottostanti.
- Ha una forma a clessidra, largo in alto e in basso, stretto al centro. Al di sopra di IP vi possono essere molti protocolli di trasporto, ciascuno per implementare una diversa astrazione di canale da offrire come servizio ai programmi applicativi. Al di sotto di IP, l'architettura consente l'utilizzo di un numero arbitrariamente grande di tecnologie di rete, spaziando da Ethernet a reti wireless, a singoli collegamenti punto-punto.

3.4 – Stack TCP/IP di Microchip

Lo stack TCP/IP è un insieme di codici sorgente i quali permettono l'implementazione su un microcontrollore PIC di una grande varietà di servizi offerti dall'architettura Internet. Esso è implementato in una struttura modulare che rispecchia l'architettura a strati di TCP/IP e permette l'abilitazione e la configurazione dei diversi servizi, definiti ad un alto livello di astrazione, per mezzo delle macro di configurazione contenute nei file di intestazione, il che rende la struttura molto flessibile e facile da configurare. Analogamente allo stack ZigBee, anche lo stack TCP/IP prevede l'utilizzo di un apposito modulo hardware contenente tutta l'elettronica necessaria all'implementazione della comunicazione e che si interfaccia al microcontrollore per mezzo della comunicazione SPI.

Cap. 4: Integrazione dei due stack

4.1 – Premessa

Al fine di implementare le comunicazioni ZigBee ed Ethernet, Microchip fornisce delle librerie pronte all'uso, open source, le quali, almeno per l'applicazione di demo, garantiscono il funzionamento, mascherando al programmatore tutti gli aspetti che riguardano le specifiche delle due comunicazioni, pur lasciando in evidenza il codice sorgente.

Tuttavia, il codice fornito risulta spesso poco flessibile, difficilmente trasportabile e scarsamente integrabile. In seguito verrà analizzata e discussa la possibile integrazione di due librerie, una relativa al protocollo TCP/IP e l'altra relativa al protocollo ZigBee, in un unico integrato.

Il presente lavoro di tesi è da considerarsi la continuazione di altri lavori; si farà riferimento, a tal fine, all'operato dell'Ing. P. Russo e alla sua tesi di laurea: "Realizzazione di un gateway TCP/IP – ZigBee per reti di sensori wireless (Standard IEEE 1451)".

4.1.1 – Considerazioni

I test dei due stack di Microchip, effettuati singolarmente forniscono esito positivo: la libreria per la comunicazione Ethernet permette ad un PC di connettersi e di utilizzare la pagina HTTP memorizzata all'interno del sistema; la libreria per la comunicazione ZigBee permette al microcontrollore di interagire con altri nodi (WTIM).

Nel momento in cui i due stack coesistono si verifica un'anomalia: pur senza manifestare errori in compilazione e nella sequenza degli stati il nodo NCAP non riesce ad inizializzare la rete, non rileva il nodo WTIM

e di conseguenza non instaura alcuna comunicazione.

Proseguendo nella direzione proposta dal dott. Russo, si è verificato l'utilizzo, da parte dei due stack, delle risorse del microcontrollore per gestire le temporizzazioni. Il risultato della verifica ha evidenziato che i due codici non sono in conflitto per quanto riguarda l'utilizzo dei timer, infatti, per la comunicazione su Ethernet è utilizzato il timer 1, mentre per la comunicazione su ZigBee sono utilizzati i timer 2 e 3 come unico timer a 32 bit.

A seguito di un'analisi del funzionamento del sistema tramite il debugger integrato nell'ambiente di sviluppo MPLAB si è giunti all'ipotesi secondo la quale la causa dell'anomalia è da attribuirsi alla condivisione delle periferiche sul bus SPI del microcontrollore, le quali permettono la comunicazione tra il PIC e i moduli hardware che favoriscono le trasmissioni Ethernet e ZigBee.

Il PIC24FJ128GA010 dispone di due periferiche interne per l'implementazione delle comunicazioni seriali di tipo SPI, e sono utilizzate entrambe sia dallo stack TCP/IP sia dallo stack ZigBee. Le parti hardware da interfacciare a tali periferiche sono tre: i due moduli per le comunicazioni Ethernet e ZigBee e una memoria EEPROM esterna nella quale sono salvate le pagine HTTP dalla libreria TCP/IP e varie informazioni relative alla rete ZigBee dell'omonima libreria. Nella fase di integrazione dei due stack le risorse sono state divise nel seguente modo, come visibile in Fig. 4.1.1.

- Modulo ZigBee connesso alla periferica SPI 1.
- Modulo Ethernet connesso alla periferica SPI 2.
- Memoria EEPROM connessa alla periferica SPI 2.

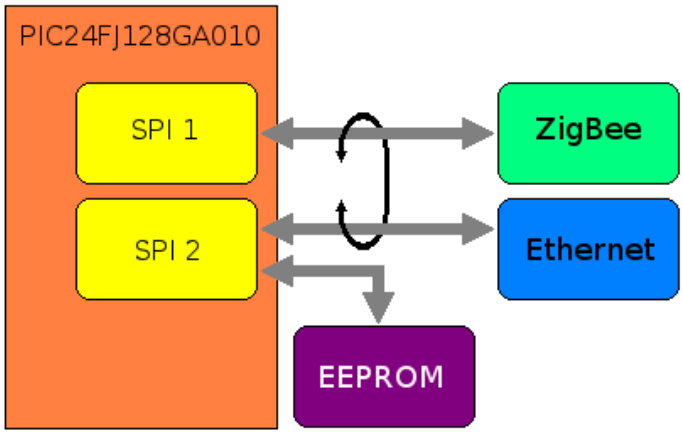

Fig. 4.1.1: Suddivisione originaria delle risorse SPI.

Di questi hardware esterni, il modulo Ethernet, ovvero la PicTail Ethernet, è l'unico ad essere configurato come Master del bus SPI, infatti il circuito integrato di cui esso è equipaggiato, ENC24J60, si occupa di fornire il clock e di iniziare la comunicazione. Per quanto riguarda le comunicazioni con il modulo ZigBee e con la memoria esterna, il Master del bus è il microcontrollore.

Si osserva, pertanto, che il modulo Ethernet non può essere fisicamente connesso alla periferica SPI 2, in quanto la sua configurazione è in conflitto con quella della EEPROM.

Un aspetto che sostiene la suddetta ipotesi si basa sull'osservazione che, lasciando operare il sistema senza connettere la PicTail Ethernet il modulo ZigBee riesce a rilevare la presenza del nodo WTIM.

È necessario, pertanto, provvedere a scambiare fisicamente le connessioni dei moduli di comunicazione al fine di ottenere sulla periferica SPI 2 solamente le parti hardware funzionanti da slave del bus; sulla periferica SPI 1 solamente le parti hardware funzionanti da master.

Un ulteriore aspetto da considerare è il conflitto di spazio occupato dai due stack nella memoria EEPROM esterna. La libreria ZigBee prevede

anche il non utilizzo di tale memoria, ridefinendo opportunamente la macro di configurazione visibile nel Cod. 4.1.1, tratto dal file zigbee.def.

```
// SPI and Non-volatile Storage Information
#define USE_EXTERNAL_NVM
```

Cod. 4.1.1: Macro di configurazione che specifica l'utilizzo della memoria esterna.

Non definendo tale macro, il codice dovrebbe occuparsi di salvare i dati nella memoria di programma del microcontrollore; in realtà si presentano diversi errori.

Il presente lavoro di tesi prosegue nella direzione di risolvere tali errori al fine di liberare il codice dello stack ZigBee dalla memoria EEPROM. La Fig. 4.1.2 fornisce informazioni circa l'occupazione della memoria di programma del microcontrollore: prima e dopo le modifiche introdotte. In particolare, la parte interna del grafico, che rappresenta la situazione prima dell'introduzione delle modifiche, è interessata da una porzione che rappresenta lo spazio di memoria occupato dal codice sorgente e da una porzione che rappresenta lo spazio vuoto rimanente. Nella parte esterna del grafico, relativa all'occupazione della memoria dopo le modifiche, è introdotta una porzione che rappresenta la memoria occupata dai dati della rete ZigBee. I dati relativi al grafico sono riportati in Tabella 1, espressi in parole di istruzione.

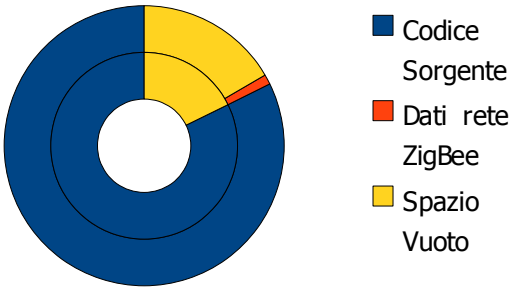

Fig. 4.1.2: Impiego della memoria di programma:
- prima della modifica (parte interna);
- dopo la modifica (parte esterna).

	Codice Sorgente	Rete ZigBee	Spazio Vuoto
Prima	36221	0	7809
Dopo	36221	512	7297

Tabella 1: Occupazione di memoria di programma (in Instruction Word).

La Fig. 4.1.3 mostra l'occupazione della memoria EEPROM esterna: la parte interna del grafico è relativa all'occupazione dopo le modifiche apportate; la parte esterna, invece, è relativa all'occupazione prima di esse.

In questo contesto non si indicano i dati relativi al grafico, in quanto variano sensibilmente a seconda delle pagine web impiegate dai servizi TCP/IP. La figura vuole focalizzare l'attenzione sul fatto che dopo le modifiche lo stack ZigBee lascia l'intera capacità della EEPROM allo stack TCP/IP: di fatto, la parte in rosso, rappresentante proprio lo spazio occupato dalle informazioni relative alla rete ZigBee, viene rimossa.

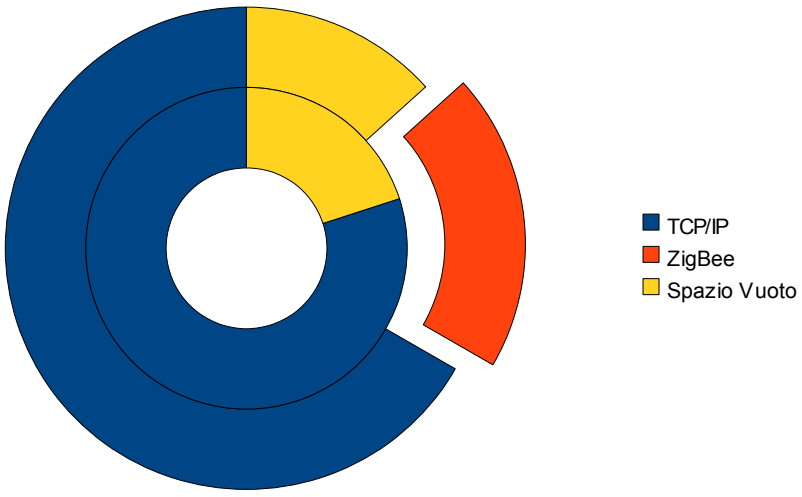

Fig. 4.1.3: Occupazione della memoria EEPROM esterna:

- prima della modifica (parte esterna);
- dopo la modifica (parte interna).

Cap. 5: Implementazione dello Stack ZigBee nella memoria interna

5.1 – Introduzione

Come già accennato nel capitolo 4, non definendo la macro che specifica l'utilizzo della memoria esterna, compaiono diversi errori, sia di compilazione che di esecuzione. Si procede pertanto nella risoluzione di tali errori, iniziando naturalmente, da quelli di compilazione.

5.2 – Risoluzione degli errori di compilazione

Partendo dal progetto di Demo rilasciato da Microchip (DemoPIC24FCoordinator), scaricabile gratuitamente dal sito internet dello stesso produttore si effettuano i seguenti passi:

1. Si cambia la macro *#define USE_EXTERNAL_NVM* in *#undef USE_EXTERNAL_NVM* nel file zigbee.def, per indicare allo stack ZigBee di salvare le informazioni relative alla rete nella memoria di programma del PIC.
2. Si aggiorna il compilatore alla versione 3.23 (o superiore), in quanto le versioni precedenti presentano un bug nell'istruzione expand_mult.
3. Si definiscono le macro *ERASE_BLOCK_SIZE* e *WRITE_BLOCK_SIZE* nel file zNVM.c, le quali indicano le dimensioni dello spazio in memoria destinato alle informazioni relative alla rete ZigBee. Le modifiche allo stack, documentate in

seguito non ne fanno uso, ma per risolvere gli errori di compilazione è possibile definirle assegnando loro un valore arbitrario.

4. Si riscrive la funzione NVMWrite(…) presente nel file zNVM.c per poterla adattare ai microcontrollori PIC24, in quanto la funzione rilasciata da Microchip è destinata ai PIC18. Questo argomento sarà oggetto dei seguenti paragrafi.
5. Nella funzione ClearNVM(…) del file zNVM.c è definita la parola chiave "rom" la quale la si deve sostituire con la parola chiave "ROM", poiché la definizione originale è un errore di sintassi.
6. Nel file zigbee.def si definisce la direttiva *#undef EE_AND_RF_SHARE_SPI*. Questa modifica non è strettamente indispensabile, ma aiuta coloro i quali devono leggere e lavorare sulla libreria.

5.3 – Le funzioni di storage

Le funzioni NVMWrite(…) e NVMRead(…) vengono fornite da Microchip direttamente per i soli PIC18. Devono pertanto essere adattate ai PIC24.

Gli aggiornamenti sono effettuati secondo quanto indicato dalle informazioni rilasciate nei documenti seguenti, reperibili dal sito del produttore:

1. DS51456F – "16bit Language Tools Libraries", sezione 4.7: "Function For Erasing and Writing Flash Memory"
2. DS39715A – "Section 4. Program Memory", sez. 4.6.1 e 4.6.2

Il comportamento delle funzioni è differente a seconda della definizione della macro *USE_EXTERNAL_NVM*. Infatti, se lo stack è configurato per salvare le informazioni relative alla rete ZigBee nella EEPROM esterna, esse sono implementate in modo tale da impiegare la comunicazione

sul bus SPI; altrimenti sono implementate per salvare i dati nella memoria FLASH di programma del microcontrollore. Per esprimere questo concetto in linguaggio C si fa uso della direttiva *#ifdef* come mostrato, nel caso della funzione NVMWrite, nel frammento di codice 5.3.1. Anche la firma è diversa nei due casi: nel primo il parametro *dest* indica la locazione nella memoria EEPROM nella quale iniziare a scrivere i dati; nel secondo lo stesso parametro è un puntatore alla locazione di memoria di programma del PIC ed è di tipo NVM_ADDR che, come si vedrà nella sezione 5.5 ed in particolare nel frammento di codice 5.5.2, corrisponde ad un byte.

```
#ifdef(USE_EXTERNAL_NVM)

void NVMWrite(WORD dest, BYTE *src, BYTE count){
      // Impiego del bus SPI per scrivere nella EEPROM esterna.
}
#else

void NVMWrite(NVM_ADDR *dest, BYTE *src, BYTE count){
      // scrittura nella memoria di programma del PIC.
}

#endif
```

Cod. 5.3.1: Dichiarazione della funzione NVMWrite a seconda della macro USE_EXTERNAL_NVM.

5.3.1 – Organizzazione hardware della memoria

Come accennato in chiusura del paragrafo 4.1.1, le informazioni relative alla rete ZigBee possono essere salvate nella memoria di programma del microcontrollore. Questo aspetto è reso possibile dall'organizzazione hardware della memoria del PIC, di tipo Harvard modificata.

L'organizzazione Harvard prevede la separazione fisica degli spazi di memoria di programma e di memoria dati, pertanto la CPU ha la possibilità di accedere contemporaneamente ad entrambe. In questo modo il PIC è in grado di eseguire un'istruzione in soli due periodi di clock: il primo per eseguire i fetch dell'istruzione e degli operandi; il

secondo per lasciare agire l'unità aritmetico-logica.

La modifica all'organizzazione Harvard presente nei microcontrollori Microchip permette la scrittura e la lettura run-time delle istruzioni, ovvero rende possibile trasferire valori dal bus di memoria dati al bus di memoria di programma e viceversa, per mezzo di istruzioni particolari, denominate TBLWR e TBLRD, le quali agiscono su un'apposita unità funzionale interna al microcontrollore detta Data Access Control Block.

In Fig. 5.3.1 è rappresentato uno schema a blocchi molto semplificato di tale organizzazione.

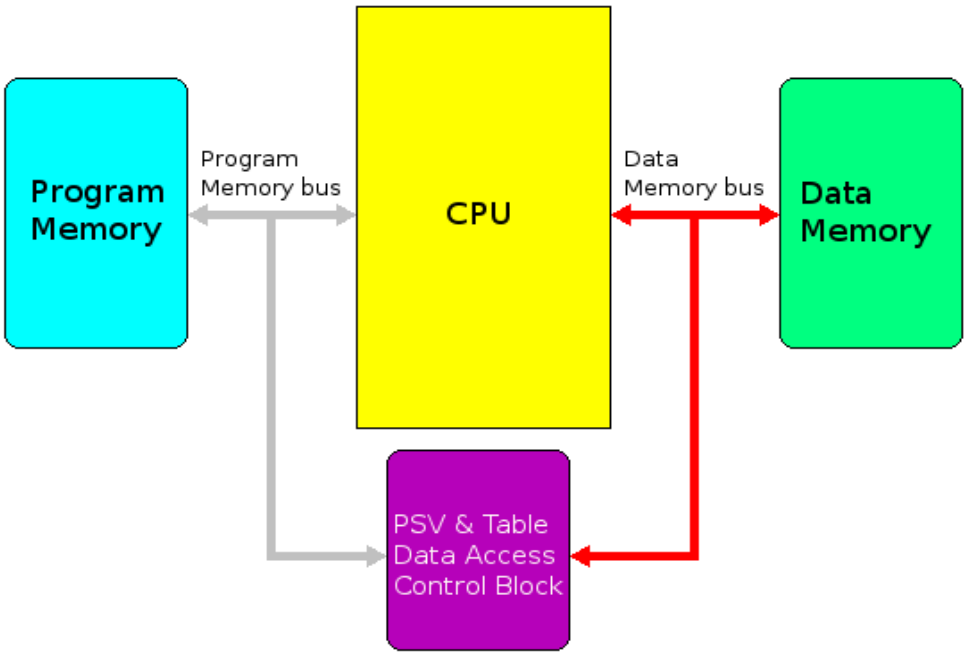

Fig. 5.3.1: Schema a blocchi semplificato dell'organizzazione Harvard modificata

5.3.2 – Allocazione in memoria di programma

Al fine di accedere in scrittura alla memoria di istruzioni, in fase di programmazione è necessario riservare uno spazio al suo interno, al

quale sarà possibile accedere per mezzo di un opportuno puntatore. La dichiarazione di un puntatore di questo tipo non è prevista dal linguaggio C standard, infatti, normalmente le variabili sono allocate nello spazio di memoria dati (RAM). Questo aspetto è uno dei tanti nel mondo della programmazione dei microcontrollori a richiedere alcune estensioni del linguaggio di programmazione.

Il compilatore C-30, sviluppato ad hoc per la programmazione dei PIC a 16 bit, prevede l'esistenza delle sopracitate estensioni; in particolare, per allocare una variabile nello spazio di memoria di programma, si specifica un apposito attributo, visibile nel frammento di codice 5.3.2, al seguito della sua dichiarazione.

```
__attribute__((space(prog), aligned(_FLASH_PAGE*2)))
```

Cod. 5.3.2: Attributo per l'allocazione in memoria di programma.

L'attributo è composto da due parole chiave: "space" e "aligned". La prima fornisce al compilatore indicazioni circa lo spazio di memoria in cui deve essere allocata una variabile; la seconda indica il suo allineamento all'interno della memoria di destinazione.

Il valore dell'argomento dell'attributo "aligned" deve essere calcolato in modo tale che la parte della memoria di programma riservata a contenere i dati non sia compresa tra parole di istruzione che costituiscono il codice, come visibile in Fig. 5.3.2.

Fig. 5.3.2: Occupazione corretta della memoria di programma.

Nel presente progetto è stato dichiarato un buffer allocato in memoria di programma di dimensioni pari a 512 istruzioni, come visibile nel frammento di codice 5.3.3.

```
/** Definizione del tipo per il buffer.
 *  Attributo space(prog) per allocare il buffer in memoria di
 *  programma.
 *  Attributo aligned(_FLASH_PAGE) per allineare il buffer a una
 *  pagina nella flash.
*/
#define NVM_BUFFER BYTE
        __attribute__((space(prog), aligned(2*_FLASH_PAGE)))

/** Definizione del buffer. Questo buffer ha la funzione di
 * allocare gli indirizzi in memoria di programma.
 *
 *  Riservo in memoria 2*_FLASH_PAGE = 1024 byte
 */
extern NVM_BUFFER NVMBuffer[2*_FLASH_PAGE];
```

Cod. 5.3.3: Allocazione dello spazio in memoria di programma.

Una parola di istruzione è composta da 24 bit, ma è bene considerarla come se fosse formata da 32 bit, suddivisa in due WORD di 16 bit ciascuna: "Most Significant Word" (MSW) e "Least Significant Word" (LSW), in cui gli otto bit più significativi della MSW non sono implementati in hardware e nel loro insieme sono detti "Phantom Byte", come visibile in Fig. 5.3.3. Una singola parola di istruzione occupa due locazioni in memoria di programma: una per la LSW e una per la MSW. Gli indirizzi della locazione di memoria della LSW risultano essere pari, mentre quelli della MSW sono dispari.

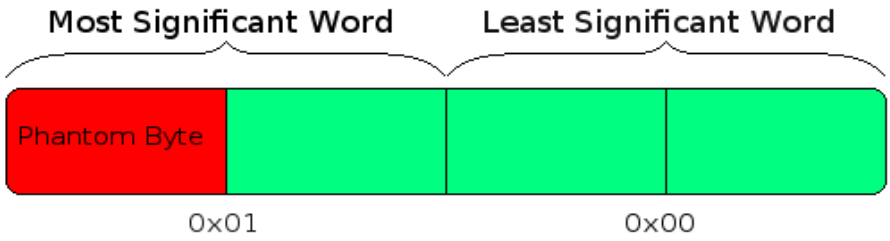

Fig. 5.3.3: Organizzazione di una parola di istruzione.

5.3.3 – Funzione di lettura

L'aggiornamento della funzione di lettura – come del resto quella di scrittura – focalizza l'attenzione su un aspetto molto importante per la sua integrazione nel contesto dell'intero stack ZigBee: non ne apporta modifiche alla firma (visibile nel frammento di codice 5.3.4) né al comportamento, al fine di mantenerne "l'interfaccia" invariata. In questo modo si ottiene un codice compatibile, nei limiti del possibile, con la libreria.

```
/** Legge dalla memoria di programma.
 *    @param * source: l'indirizzo della memoria di programma
 *                     dalla quale leggere i dati.
 *    @param * data:   l'indirizzo di un buffer allocato in RAM
 *                      nel quale scrivere i dati letti.
 *    @param count:    il numero di BYTE da leggere.
*/
void NVMRead(BYTE * source, BYTE * data, WORD count);
```

Cod. 5.3.4: La firma della funzione di lettura.

Come si può osservare dalla definizione nel frammento di codice 5.3.4, la funzione accetta tre parametri: il primo è il puntatore alla locazione di memoria di programma dalla quale si iniziano a leggere i dati, il secondo è il puntatore alla locazione in RAM nella quale scrivere i dati letti, mentre il terzo è un valore che indica quanti byte si devono leggere.

Sebbene il compilatore C-30 permetta di riservare uno spazio in memoria di programma, per effettuarne la lettura di una determinata locazione non è possibile semplicemente assegnare il suo valore direttamente ad una variabile allocata in RAM, ma è necessario coinvolgere le istruzioni speciali di TBLRD.

Le istruzioni di questo tipo sono due: TBLRDH per leggere la word più significativa di una parola di istruzione; TBLRDL per leggere quella meno significativa. Tali funzioni necessitano di due parametri: il puntatore ad una locazione in RAM nel quale sarà trasferito il dato

letto; ed una WORD contenente la parte meno significativa dell'indirizzo della locazione. Per completare tale indirizzo, le funzioni si appoggiano al registro TBLPAG, il quale contiene gli otto bit più significativi che lo compongono. Questo concetto è rappresentato schematicamente nelle Fig. 5.3.4 e 5.3.5.

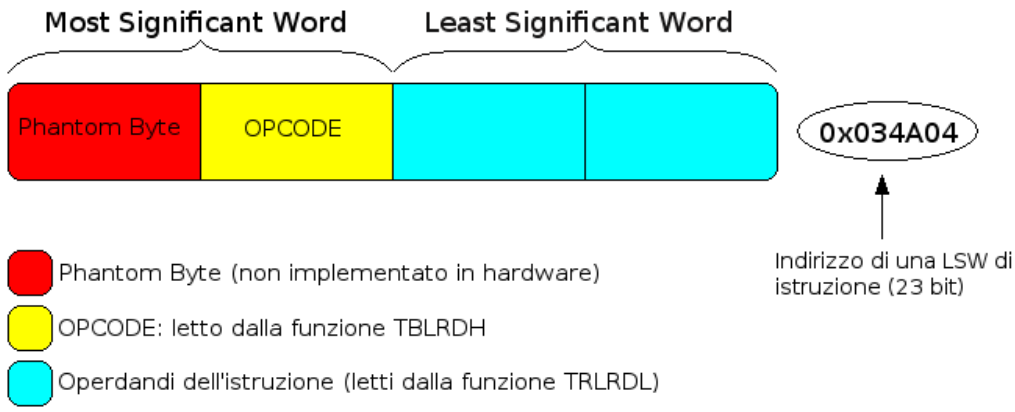

Fig. 5.3.4: Accesso delle funzioni TBLRD alla parola di istruzione.

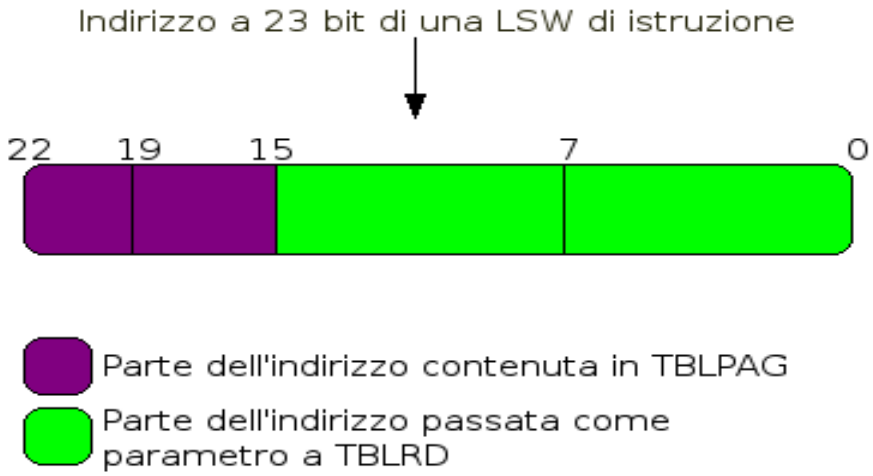

Fig. 5.3.5: Composizione dell'indirizzo per la lettura di una parola di istruzione

In questo contesto la memoria di programma è utilizzata per salvare dati; pertanto è buona norma impiegare come spazio utile la sola parte LSW della parola di istruzione ed impostare il suo codice operativo al valore 0xFF, il quale identifica l'istruzione Nop. In questo modo,

qualora per errore il Program Counter puntasse ad un indirizzo in memoria FLASH contenente dati, la CPU non eseguirebbe alcuna istruzione, potenzialmente dannosa. La lettura dei dati interessa, pertanto i soli valori forniti dalla funzione TBLRDL, poiché sono quelli memorizzati nella parte LSW della parola di istruzione.

Nel frammento di codice 5.3.5 è rappresentata la funzione NVMRead. È possibile osservare l'inizializzazione del registro TBLPAG, con i sette bit più significativi dell'indirizzo da leggere, mentre alle funzioni di TBLRD è passato un valore contenente i sedici bit meno significativi dello stesso indirizzo ed un puntatore ad una variabile allocata in RAM nella quale sono scritti i dati letti dalla FLASH.

Una volta conclusa la lettura delle locazioni in memoria di programma, è richiesta l'esecuzione di due operazioni Nop. Le ragioni di tale richiesta sono da attribuire all'organizzazione hardware del microcontrollore, ma Microchip non fornisce ulteriori dettagli in merito.

Le variazioni della firma della funzione, rispetto all'originale sono due: la prima riguarda il tipo del parametro *count*; la seconda riguarda l'ordine invertito dei parametri *source* e *data*. Di queste la prima è indispensabile, in quanto in alcune fasi si ha la necessità di leggere un'intera FLASH_PAGE (512 istruzioni, 1024 byte), e tale valore non è rappresentabile su 8 bit. La seconda è stata introdotta per uniformità con la funzione di scrittura: il primo argomento è un puntatore alla memoria di programma, il secondo è un puntatore alla memoria RAM. Come si vedrà in seguito, quest'ultima modifica non comporta cambiamenti di massa nelle parti dello stack originale, in quanto tutte le funzioni che si occupano di leggere le diverse informazioni dalla memoria di programma sono definite in un unico file: tali definizioni sono le uniche parti di codice nell'intero stack ad essere interessate dalla suddetta inversione di ordine.

```
void NVMRead(BYTE * source, BYTE * data, WORD count){
      // Dichiarazione delle variabili:
      int VarWord; // Lettura della HI-Word della locazione in
                   // memoria di programma.
      int i, temp; // Indice del ciclo e contenitore temporaneo
                   // della WORD letta da flash.
      int lim = count/2; // Numero di iterazioni del ciclo. (La
                         // divisione per 2 è perché ogni lettura
                         // dalla flash sono 2 byte).
      int addrOffset; // Locazione di memoria dalla quale leggere
                      // una WORD.

      // Carico la locazione di memoria da leggere in TBLPAG e in
      // addrOffset
      TBLPAG = (((WORD)source & 0x7F0000)>>16);
      addrOffset = ((WORD)source & 0x00FFFF);

      // Se devo leggere un numero di byte dispari allora
      // lim = 1 + count/2;
      if(count & 0x0001) lim++;
            // Ciclo di lettura delle locazioni. Ogni ciclo legge
            // una WORD, la scompone in 2 byte e li memorizza nel
            // corrispondente indice del buffer data.
          for(i=0; i<lim; i++){
              asm("tblrdh.w [%1], %0" : "=r"(VarWord) :
                  "r"(addrOffset+2*i));
              asm("tblrdl.w [%1], %0" : "=r"(temp) :
                  "r"(addrOffset+2*i));
              *(data+2*i) = (BYTE)(temp & 0x00FF);
              if(!(i==count/2 && (count%2==1)))
                  *(data+2*i+1) = (BYTE)((temp & 0xFF00)>>8);
          }

      // Due Nop richiesti dall'hardware.
      Nop();
      Nop();
}
```

Cod. 5.3.5: Il codice della funzione NVMRead.

5.3.4 - La funzione di scrittura

Analogamente alla funzione di lettura, anche per la funzione di scrittura si è prestato attenzione a mantenerne l'interfaccia invariata, al fine di preservare la sua compatibilità con il codice presente nello stack.

```
/** Scrive nella memoria di programma.
 *  @param * dest: l'indirizzo della memoria di programma nella
 *                 quale scrivere i dati.
 *  @param * data: l'indirizzo di un buffer allocato in RAM che
 *                 contiene i dati da scrivere.
 *  @param count: il numero di BYTE da scrivere.
*/
void NVMWrite(BYTE * dest, BYTE * data, WORD count);
```

Cod. 5.3.6: La firma della funzione di scrittura.

Anche questa funzione accetta tre parametri, che corrispondono a quelli della funzione di lettura. Infatti, il primo è un puntatore ad una locazione in memoria di programma nella quale ha inizio la scrittura dei dati, il secondo è un puntatore ad una locazione in RAM contenente i dati da scrivere, il terzo indica quanti byte devono essere scritti.

Sebbene il compilatore C-30 permetta di dichiarare una variabile allocandola in memoria di programma, non è possibile semplicemente assegnarle un valore direttamente da una variabile allocata in RAM, ma è necessario coinvolgere le istruzioni speciali di TBLWR. In questo contesto è risultata molto utile la libreria "libpic30", la quale contiene alcune funzioni che semplificano la programmazione del dispositivo, mascherando al programmatore l'impiego di TBLWR. In particolare la funzione che ha riscontrato maggiore utilità è la _write_flash16, la quale si occupa di scrivere una determinata parola di istruzione, avendo cura di impostarne il valore del codice operativo a 0xFF, corrispondente all'istruzione Nop.

I parametri richiesti da quest'ultima istruzione sono due: un puntatore alla parola in memoria di programma da scrivere, ed un puntatore ad una locazione in RAM, contenente il valore da scrivere. Di questi, il primo non è un puntatore ad un valore standard del linguaggio C, ma è definito all'interno della libreria "libpic30", con l'etichetta *_prog_addressT* ed è inizializzato ad una locazione di memoria di

programma per mezzo della funzione *_init_prog_address*.

```
void NVMWrite(BYTE * dest, BYTE * data, WORD count){
	// Dichiarazione delle variabili.
	int i;			// Indice del ciclo.
	int offset;
	WORD temp[_FLASH_PAGE]; // Dati letti dalla Flash.
	BYTE * t;
	_prog_addressT p; // Puntatore alla memoria di programma.

	// Inizializzazione del puntatore alla memoria di programma.
	_init_prog_address(p, NVMBuffer);

	// Lettura di 512 istruzioni: 1 istruzione = 2 byte ->
	// Byte da leggere = 1024
	NVMRead(&NVMBuffer[0], (BYTE*)temp, 2*_FLASH_PAGE);

	// Aggiornamento del buffer temp. Tale buffer sarà
	// completamente riscritto in ROM. Calcolo l'offset in base
	// al fatto se dest è pari o dispari.
	if((int)dest & 0x01) // Se dest è dispari...
		offset = (dest - NVMBuffer - 1)/2;
	else // Se dest è pari...
		offset = (dest - NVMBuffer)/2;

	// Inizializzo il puntatore t.
	t = (BYTE*)temp + 2*offset;
	for(i=0; i<count; i++){
		*(t+i)=*(data+i);
	}
	// Cancella la pagina puntata da p: una pagina = 512 istr;
	// 1 istruzione = 2 byte.
	_erase_flash(p);

	// Scrittura nella memoria flash.
	for(i=0; i<_FLASH_PAGE; i+=_FLASH_ROW){
		// Scrive una riga di programma: 64 istruzioni.
		_write_flash16(p, (int*)(temp+i));
		// Il puntatore deve avanzare di 2*_FLASH_ROW perché
		// _FLASH_ROW indica il numero di istruzioni in una
		// riga, ma ogni istruzione occupa 2 locazioni.
		p+=2*_FLASH_ROW;
	}

	// Due Nop richiesti dall'hardware.
	Nop();
	Nop();
	return;
}
```

Cod. 5.3.7: Il codice della funzione NVMWrite

Per ragioni dovute all'implementazione hardware dell'unità funzionale Data Access Control Block, visibile in Fig. 5.3.1, per poter scrivere all'interno della memoria FLASH è necessario prima cancellarne il contenuto. Pertanto la funzione NVMWrite inizialmente legge il contenuto della memoria di programma – per mezzo della funzione NVMRead – allocandolo in RAM, ne aggiorna i valori (sempre in RAM), cancella la memoria di programma ed infine, scrive i dati aggiornati dalla memoria dati in memoria FLASH.

Lo spazio in RAM nel quale sono memorizzati e aggiornati i dati letti da FLASH sono allocati dal buffer *temp*. L'aggiornamento è composto da un ciclo *for*, il quale ad ogni iterazione scrive due byte che occuperanno una parola di istruzione.

Le variazioni della firma della funzione NVMWrite rispetto all'originale, riguardano solamente il tipo del parametro *count*: analogamente a quanto visto per la funzione di lettura il numero di byte da scrivere può essere maggiore di 255, pertanto si rende necessario un dato di dimensioni maggiori di 8 bit.

5.4 – Utilizzo delle funzioni di storage

In seguito all'aggiornamento delle funzioni di storage si rende necessario verificarne la correttezza nell'utilizzo da parte dello stack. Pertanto il lavoro è stato eseguito ricercando all'interno della libreria ZigBee le relative chiamate, elencate nelle successive sezioni. In seguito, si procederà all'esplorazione di ognuna di esse per verificare se e quali modifiche sarà necessario apportare.

Tale ricerca, all'interno dell'intero progetto, è stata eseguita servendosi dell'ausilio dei mezzi messi a disposizione dell'ambiente di sviluppo MPLAB. In particolare, si è fatto uso della ricerca globale, accessibile tramite i tasti CTRL+SHIFT+F. Le stringhe da ricercare sono

"NVMWrite" e *"NVMRead"* rispettivamente per la funzione di scrittura e di lettura, come mostrato in Fig. 5.4.1.

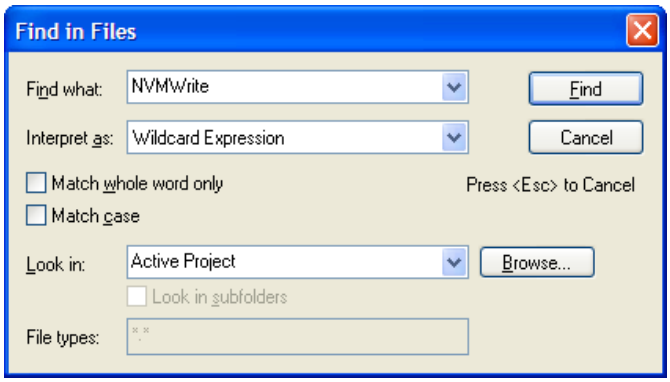

Fig. 5.4.1: Strumento di ricerca delle porzioni di codice.

5.4.1 - Chiamate alle funzioni di scrittura

```
#define ZDOPutCapabilityInfo(x)
NVMWrite((NVM_ADDR*)&Config_Node_Descriptor.NodeMACCapabilityFlags
, (void *)x, 1)

#define PutMACAddress(x)
NVMWrite((NVM_ADDR*)&macLongAddr, (BYTE*)x, sizeof(LONG_ADDR))

#define PutAPSAddress(x, y)
NVMWrite((NVM_ADDR*)x, (BYTE*)y, sizeof(APS_ADDRESS_MAP))

#define PutAPSAddressValidityKey(x)
NVMWrite((NVM_ADDR *)&apsAddressMapValidityKey,(BYTE *)x,
sizeof(WORD))

#define PutBindingRecord( x, y )
NVMWrite((NVM_ADDR *)x, (BYTE *)y, sizeof(BINDING_RECORD))

#define PutBindingSourceMap(x, y)
NVMWrite((NVM_ADDR *)&bindingTableSourceNodeMap[y>>3], (BYTE *)x,
1 )

#define PutBindingUsageMap(x, y)
NVMWrite((NVM_ADDR *)&bindingTableUsageMap[y>>3], (BYTE *)x, 1 )

#define PutBindingValidityKey( x )
NVMWrite((NVM_ADDR *)&bindingValidityKey, (BYTE *)x, sizeof(WORD))

#define PutNeighborRecord(x, y)
NVMWrite((NVM_ADDR *)x, (BYTE *)y, sizeof(NEIGHBOR_RECORD))

#define PutNeighborTableInfo()
 NVMWrite((NVM_ADDR*)&neighborTableInfo,
      (BYTE*)&currentNeighborTableInfo,
sizeof(NEIGHBOR_TABLE_INFO))

#define PutRoutingEntry(x, y)
      NVMWrite((NVM_ADDR *)x, (BYTE*)y, sizeof(ROUTING_ENTRY))

#define PutTrustCenterAddress(x)
NVMWrite((NVM_ADDR *)&trustCenterLongAddr, (BYTE *)x,
sizeof(LONG_ADDR))

#define PutNwkActiveKeyNumber(x)
NVMWrite((NVM_ADDR *)&nwkActiveKeyNumber, (BYTE *)x, 1)
```

Cod. 5.4.1: Definizioni delle chiamate alla funzione di scrittura.

```
#define PutNwkKeyInfo(x, y)
      NVMWrite((NVM_ADDR *)x, (BYTE *)y, sizeof(NETWORK_KEY_INFO))

#define PutGroupAddress(x, y)
NVMWrite((NVM_ADDR *)x, (BYTE *)y, sizeof(GROUP_ADDRESS_RECORD))
```

5.4.2 - Chiamate alle funzioni di lettura

```
1. #define ZDOGetCapabilityInfo(x)
   NVMRead(x,
       (ROM void*)
   (&Config_Node_Descriptor.NodeMACCapabilityFlags), 1)

2. #define GetMACAddress(x)
   NVMRead((BYTE *)x, (ROM void*)&macLongAddr,
   sizeof(LONG_ADDR))

3. #define GetMACAddressByte(y, x)
   NVMRead((BYTE *)x, (ROM void*)((int)&macLongAddr + (int)y),
   1)

4. #define ProfileGetNodeDesc(p)
   NVMRead((BYTE *)p, (ROM void*)&Config_Node_Descriptor,
           sizeof(NODE_DESCRIPTOR))

5. #define ProfileGetNodePowerDesc(p)
   NVMRead((BYTE *)p, (ROM void*)&Config_Power_Descriptor,
           sizeof(NODE_POWER_DESCRIPTOR))

6. #define ProfileGetSimpleDesc(p, i)
   NVMRead((BYTE *)p, (ROM
   void*)&(Config_Simple_Descriptors[i]),
           sizeof(NODE_SIMPLE_DESCRIPTOR))

7. #define GetAPSAddress(x, y)
   NVMRead((BYTE *)x, (ROM void*)y, sizeof(APS_ADDRESS_MAP))

8. #define GetAPSAddressValidityKey(x)
   NVMRead((BYTE *)x, (ROM void*)&apsAddressMapValidityKey,
   sizeof(WORD))

9. #define GetBindingRecord(x, y)
   NVMRead((BYTE *)x, (ROM void*)y, sizeof(BINDING_RECORD))

10.#define GetBindingSourceMap(x, y)
   NVMRead((BYTE *)x,
           (ROM void*)&bindingTableSourceNodeMap[y>>3], 1)
```

```
11.#define GetBindingUsageMap(x, y)
   NVMRead((BYTE *)x, (ROM void*)&bindingTableUsageMap[y>>3],
   1)

12.#define GetBindingValidityKey(x)
   NVMRead((BYTE *)x, (ROM void*)&bindingValidityKey,
   sizeof(WORD))

13.#define GetNeighborRecord( x, y )
   NVMRead((BYTE *)x, (ROM void*)y, sizeof(NEIGHBOR_RECORD))

14.#define GetNeighborTableInfo()
   NVMRead((BYTE *)&currentNeighborTableInfo,
   (ROM void*)&neighborTableInfo, sizeof(NEIGHBOR_TABLE_INFO))

15.#define GetRoutingEntry(x, y)
   NVMRead((BYTE *)x, (ROM void*)y, sizeof(ROUTING_ENTRY))

16.#define GetTrustCenterAddress(x)
   NVMRead((BYTE *)x, (NVM_ADDR *)&trustCenterLongAddr,
   sizeof(LONG_ADDR))

17.#define GetTrustCenterAddressByte(y, x)
   NVMRead((BYTE *)x, (NVM_ADDR *)((int)&trustCenterLongAddr +
   (int)y), 1)

18.#define GetNwkActiveKeyNumber(x)
   NVMRead(( BYTE *)x, (ROM void *)&nwkActiveKeyNumber, 1)

19.#define GetNwkKeyInfo(x, y)
   NVMRead((BYTE *)x,(ROM void *)y, sizeof(NETWORK_KEY_INFO))

20.#define GetGroupAddress(x, y)
   NVMRead((BYTE *)x, (ROM void *)y,
   sizeof(GROUP_ADDRESS_RECORD))
```

Cod. 5.4.2: Definizione delle chiamate alle funzioni di lettura

5.4.3 - Considerazioni

Le chiamate elencate nelle precedenti sezioni devono risultare alla fine adattate al comportamento delle funzioni di lettura e scrittura. Alcune già si adattano, altre subiranno delle modifiche: importanti o lievi a seconda dei casi.

5.5 – Esplorazione degli APS ADDRESSES

In seguito a quanto riportato nella sezione 5.4 – Utilizzo delle funzioni di storage si procede alla ricerca delle funzioni PutAPSAddress(x, y) e GetAPSAddress(x, y) definite nel file zNVM.h alle righe 283 e 285 rispettivamente.

Al fine di apprendere le motivazioni delle modifiche apportate al codice è necessario tenere presente le firme delle funzioni originali NVMWrite ed NVMRead (si veda a tal fine il frammento di codice 5.5.1).

```
void NVMWrite(NVM_ADDR * dest, BYTE * src, BYTE count);

#define NVMRead(dest, src, count) memcpypgm2ram(dest, src, count)
```

Cod. 5.5.1: Firme delle funzioni originali di lettura e scrittura.

Dove il tipo NVM_ADDR è definito come visibile nel frammento di codice 5.5.2 e rappresenta una locazione in memoria di programma. Si noti che per come sono state riscritte le funzioni NVMWrite ed NVMRead (si vedano le sezioni 5.3.3 e 5.3.4) tale definizione non sarà più necessaria a modifiche ultimate.

```
#define NVM_ADDR ROM BYTE
```

Cod. 5.5.2: Definizione del tipo NVM_ADDR

Nelle successive sezioni, 5.5.1 e 5.5.2, saranno presentate le funzioni originali di scrittura e lettura degli APS ADDRESSES e le loro interazioni con le funzioni NVMWrite ed NVMRead originali, al fine di apprenderne il comportamento; nel paragrafo 5.5.3 saranno tratte tutte le considerazioni del caso e presentate le eventuali modifiche.

5.5.1 – Funzione PutAPSAddress

La funzione in esame si occupa della memorizzazione degli indirizzi a livello APS (si veda la Fig. 1.2.2) nella memoria non volatile interna al

microcontrollore. Il codice dello stack fa uso della macro di configurazione *USE_EXTERNAL_NVM* – di cui si è discusso nelle sezioni 4.1.1 e 5.2 – per la sua dichiarazione, come visibile nel frammento di codice 5.5.3.

```
#ifdef USE_EXTERNAL_NVM

#define PutAPSAddress(x, y) NVMWrite(x, (BYTE*)y,
sizeof(APS_ADDRESS_MAP))

#else

#define PutAPSAddress(x, y)
        NVMWrite((NVM_ADDR*)x, (BYTE*)y, sizeof(APS_ADDRESS_MAP))
#endif
```

Cod. 5.5.3: Definizione originale della funzione PutAPSAddress a seconda della definizione della macro USE_EXTERNAL_NVM.

La funzione rimappa semplicemente NVMWrite: una chiamata a PutAPSAddress prevede due argomenti che saranno trasferiti ai primi due parametri di NVMWrite. Il terzo parametro di quest'ultima, che indica quanti byte devono essere scritti, è costante e corrisponde alla dimensione, naturalmente in byte, di un indirizzo a livello APS. Si ricorda che in base a quanto visto nella sezione 5.3, a seconda della definizione della macro *USE_EXTERNAL_NVM*, la funzione NVMWrite è implementata per salvare i dati nella memoria EEPROM esterna, oppure nella memoria interna di programma del microcontrollore.

In questo contesto ci si occupa dei casi in cui tale macro non è definita, in quanto l'obiettivo prefissato è quello di liberare lo stack ZigBee dall'uso della memoria esterna. In questo caso le chiamate a PutAPSAddress sono dei tre tipi visibili nel frammento di codice 5.5.4: il primo parametro è un puntatore alla locazione di destinazione del

```
• PutAPSAddress(&apsAddressMap[i], &currentAPSAddress1);
• PutAPSAddress(&apsAddressMap[i], &currentAPSAddress);
• PutAPSAddress(&apsAddressMap[i], &tmpMap);
```

Cod. 5.5.4: Prototipi delle chiamate alla funzione PutAPSAddress

dato; mentre le variabili che compaiono al secondo argomento, sono dichiarate a livello locale e contengono il valore che deve essere salvato nella memoria non volatile.

Al fine di apprendere le motivazioni delle scelte tecniche che saranno adottate, è importante osservare le definizioni dei tipi di dato che interessano il contesto, iniziando dai tipi di dato più semplice e procedendo verso quelli più complessi.

```
typedef union _SHORT_ADDR{
    struct _SHORT_ADDR_bits{
        BYTE LSB;
        BYTE MSB;
    } byte;

    WORD Val;

    BYTE v[2];

} SHORT_ADDR;
```

Cod. 5.5.5: Definizione del tipo di dato SHORT_ADDR

Il tipo di dato SHORT_ADDR, la cui definizione è visibile nel frammento di codice 5.5.5, rappresenta un indirizzo di dimensione 2 byte. La particolarità della sua definizione consiste nell'utilizzo della parola chiave *union,* la quale permette di ottenere diversi tipi di accesso alla variabile dichiarata. In questo caso i possibili accessi sono tre: direttamente all'intero valore per mezzo del campo *Val*; ai singoli byte attraverso una struttura comprendente due campi

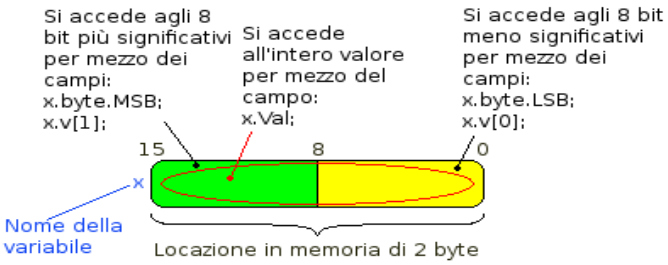

Fig. 5.5.1: Accessi alla variabile di tipo SHORT_ADDR

(LSB ed MSB) oppure attraverso un array di due elementi di tipo byte. Questo concetto è rappresentato graficamente nella figura 5.5.1.

Il tipo di dato LONG_ADDR, definito nel frammento di codice 5.5.7, rappresenta un indirizzo di otto byte. La sua struttura è dichiarata in

modo tale da poter accedere ad ognuno dei singoli otto frammenti dell'indirizzo per mezzo degli elementi dell'array v.

I due tipi di dato sopracitati sono definiti nel contesto dell'intero stack ZigBee e, come si vedrà in seguito, sono utilizzati anche per il Mac Address e per i record di binding.

Prima di procedere all'introduzione del tipo di dato che rappresenta un indirizzo a livello APS è necessario osservare un'ulteriore definizione, visibile nel frammento di codice 5.5.6: quella della parola ROM la quale si utilizza al momento della dichiarazione di una variabile per indicare che è costante. Tale definizione è stata introdotta da Microchip al fine di mantenere l'uniformità con il codice scritto per i PIC18, in quanto il relativo compilatore definisce le costanti per mezzo della parola chiave *rom*.

```
#define ROM const
```

Cod. 5.5.6: Definizione della parola chiave per indicare una costante

```
typedef struct _LONG_ADDR{
    BYTE v[8];
} LONG_ADDR;
```

Cod. 5.5.7: Definizione del tipo di dato LONG_ADDR.

Il tipo di dato che rappresenta un indirizzo a livello APS è visibile nel frammento di codice 5.5.8. Esso è formato da un indirizzo di tipo SHORT_ADDR e uno di tipo LONG_ADDR e pertanto ha dimensione 10 byte.

```
typedef struct _APS_ADDRESS_MAP{
    LONG_ADDR  longAddr;
    SHORT_ADDR shortAddr;
} APS_ADDRESS_MAP;
```

Cod. 5.5.8: Definizione del tipo APS_ADDRESS_MAP.

Al fine di memorizzare gli indirizzi APS, lo stack protocollare utilizza una delle tre chiamate visibili nel frammento di codice 5.5.4 all'interno di un ciclo *for*, come rappresentato nella porzione di software 5.5.9.

```
for (i=0; i<MAX_APS_ADDRESSES; i++){

    PutAPSAddress(&apsAddressMap[i], &currentAPSAddress);

}
```

Cod. 5.5.9: Parte di codice che salva tutti gli indirizzi APS.

La costante *MAX_APS_ADDRESSES* vale 8, pertanto lo spazio in memoria di programma da riservare agli indirizzi a livello APS è di 80 byte: ottenuto moltiplicando il valore della suddetta costante per la dimensione di un indirizzo APS.

5.5.2 - Funzione GetAPSAddress

La funzione GetAPSAddress si occupa della lettura degli indirizzi a livello APS dalla memoria non volatile; ovvero coinvolge la funzione NVMRead per trasferire i dati dalla memoria FLASH alla memoria RAM del microcontrollore per permettere alle parti dello stack ZigBee interessate di utilizzarne i valori.

Analogamente a quanto visto nel frammento di codice 5.5.9, per la funzione di memorizzazione degli stessi indirizzi, le chiamate a GetAPSAddress sono tutte all'interno di un ciclo *for*, le cui iterazioni si ripetono per MAX_APS_ADDRESSES (=8) volte, come mostrato nel frammento di codice 5.5.10.

```
for(i=0; i<MAX_APS_ADDRESSES; i++){
    GetAPSAddress(&currentAPSAddress, &apsAddressMap[i]);
}
```

Cod. 5.5.10: Parte di codice che legge tutti gli indirizzi APS.

Il parametro *currentAPSAddress* rappresenta la variabile locale in RAM nella quale è trasferito il valore dell'indirizzo APS i-esimo, salvato in memoria di programma alla locazione riferita dall'i-esimo elemento dell'array *apsAddressMap*.

5.5.3 – Considerazioni

Una prima incongruenza che si nota nel codice, riguarda la definizione dell'array *apsAddressMap,* visibile nel frammento di codice 5.5.11, tratto dal file zNVM.c. La funzione di tale array è quella di riservare in memoria di programma le locazioni nelle quali saranno memorizzati i vari indirizzi APS.

```
ROM APS_ADDRESS_MAP apsAddressMap[MAX_APS_ADDRESSES];
```

Cod. 5.5.11: Dichiarazione originale dell'array apsAddressMap.

Gli errori rilevati in questa dichiarazione sono due: il primo è la mancata inizializzazione dei suoi elementi, ai quali, essendo costanti, non potrà essere assegnato un valore neppure in una fase successiva; il secondo è la loro allocazione in RAM anziché in memoria di programma, in quanto non è specificato l'attributo *space(prog)* discusso nella sezione 5.3.2. Tali errori coinvolgono sia la funzione di lettura che quella di scrittura; la loro correzione consiste nel mutare la dichiarazione di *apsAddressMap,* come visibile nel frammento di codice 5.5.12, inizializzandone gli elementi agli indirizzi delle locazioni del buffer dichiarato in memoria di programma: NVMBuffer (introdotto nel frammento di codice 5.3.3). La funzione di tale buffer è riservare nella memoria di programma uno spazio di 512 istruzioni, il quale è utilizzato per salvare tutte le informazioni relative alla rete ZigBee.

```
ROM BYTE * apsAddressMap[MAX_APS_ADDRESSES] = {
      NVMBuffer + apsAddressMapOffset + 0*sizeof(APS_ADDRESS_MAP),
      NVMBuffer + apsAddressMapOffset + 1*sizeof(APS_ADDRESS_MAP),
      NVMBuffer + apsAddressMapOffset + 2*sizeof(APS_ADDRESS_MAP),
      NVMBuffer + apsAddressMapOffset + 3*sizeof(APS_ADDRESS_MAP),
      NVMBuffer + apsAddressMapOffset + 4*sizeof(APS_ADDRESS_MAP),
      NVMBuffer + apsAddressMapOffset + 5*sizeof(APS_ADDRESS_MAP),
      NVMBuffer + apsAddressMapOffset + 6*sizeof(APS_ADDRESS_MAP),
      NVMBuffer + apsAddressMapOffset + 7*sizeof(APS_ADDRESS_MAP)
};
```

Cod. 5.5.12: Dichiarazione ed inizializzazione corretta dell'array apsAddressMap.

Come sarà possibile constatare dall'esplorazione delle successive funzioni di lettura e scrittura nella FLASH, ogni informazione relativa alla rete ZigBee da salvare in memoria non volatile farà uso di un'opportuna variabile (o array) di remapping (nella fattispecie *apsAddressMap*). Tale variabile sarà sempre costante, di tipo puntatore a BYTE ed inizializzata in modo univoco agli elementi di NVMBuffer. Al fine di gestire correttamente l'indirizzamento di NVMBuffer, per ogni informazione da salvare si definiscono un offset e una dimensione: l'offset indica quanti byte intercorrono dalla locazione iniziale di NVMBuffer alla locazione nella quale ha inizio la memorizzazione dei valori relativi all'informazione; la dimensione indica quanti byte si devono ad essa riservare. Questo concetto è schematizzato genericamente in Fig. 5.5.2.

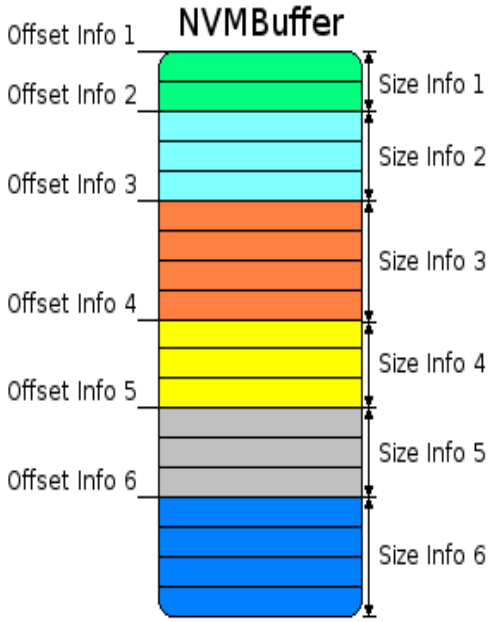

Fig. 5.5.2: Indirizzamento di NVMBuffer.

Le seguenti modifiche adattano la dichiarazione della variabile di remapping alla versione delle funzioni NVMWrite e NVMRead presentata nei paragrafi 5.3.3 e 5.3.4. Nello stack è però necessario effettuare un'ulteriore modifica, relativa alle chiamate alla funzione PutAPSAddress: con riferimento al frammento di codice 5.5.9, è necessario omettere il simbolo "&" avanti al primo parametro. Questo accorgimento si rende indispensabile poiché la variabile di remapping è già dichiarata come puntatore (si veda il simbolo "*" che precede il nome dell'array nel codice 5.5.12).

Una volta completati i suddetti aggiornamenti è possibile procedere a

definire le funzioni di scrittura e di lettura, le quali altro non sono che una mappatura delle funzioni NVMWrite ed NVMRead rispettivamente. In particolare la definizione della funzione di scrittura è riportata nel frammento di codice 5.5.13,

```
#define PutAPSAddress(x, y)
NVMWrite((BYTE*)x, (BYTE*)y, sizeof(APS_ADDRESS_MAP)
```

Cod. 5.5.13: Definizione aggiornata della funzione PutAPSAddress.

mentre la definizione della funzione di lettura è riportata nel

```
#define GetAPSAddress(x, y)
NVMRead((BYTE*)y, (BYTE*)x, sizeof(APS_ADDRESS_MAP))
```

Cod. 5.5.14: Definizione aggiornata della funzione GetAPSAddress.

frammento di codice 5.5.14.

Rispetto alle definizioni originali, riportate nei codici 5.4.1 e 5.4.2, la funzione di scrittura prevede il mutamento del tipo del cast sul primo argomento da NVM_ADDR a BYTE; mentre la funzione di lettura prevede il mutamento del tipo del cast sul secondo parametro da void a BYTE e l'inversione dell'ordine dei parametri: la y è passata al primo argomento di NVMWrite, la x al secondo. Tale inversione è la conseguenza del rovesciamento dei due parametri di NVMRead accennati nel paragrafo 5.3.3.

5.5.4 - Codice di esempio

Al fine di testare la validità delle ipotesi e delle considerazioni trattate nei paragrafi 5.5.1, 5.5.2 e 5.5.3, si scrive un'opportuna applicazione di esempio in un nuovo progetto - il quale successivamente dovrà essere integrato nell'applicazione costituente la rete di sensori - composto da cinque file.

- Il file Type.h è di intestazione, in esso sono importate tutte le definizioni dei tipi di dato utili ai vari test: nella fattispecie degli

indirizzi a livello APS si importano i tipi di dato SHORT_ADDR (Cod. 5.5.5), LONG_ADDR (Cod. 5.5.7), la definizione della parola ROM (Cod. 5.5.6) ed infine APS_ADDRESS_MAP (Cod. 5.5.8).

- Il file Config.h contiene alcune macro di configurazione dello stack utili per i test considerati.
- Il file NVM.h contiene le definizioni delle funzioni NVMWrite ed NVMRead, nonché le definizioni di offset, dimensioni, variabili di remapping e funzioni di lettura e scrittura nella memoria di programma, delle varie informazioni relative alla rete ZigBee.
- Il file NVM.c contiene le implementazioni di tutte le funzioni definite nel relativo file di intestazione: NVM.h.
- Il file Main.c contiene l'applicazione di test relativa alla lettura e al salvataggio delle informazioni, da e verso la memoria di programma.

Relativamente agli indirizzi a livello APS, le parti di codice riportate nel file NVM.h sono le due definizioni già viste nei frammenti di codice 5.5.13 e 5.5.14 e quelle visibili nel frammento di codice 5.5.15.

Essendo gli indirizzi a livello APS le prime informazioni trattate allo scopo prefissato in apertura del capitolo 5, il valore di *apsAddressMapOffset* è zero. Il valore delle dimensioni dello spazio in memoria ad essi riservato è ottenuto, come si è visto nel paragrafo 5.5.1, moltiplicando la dimensione di un singolo indirizzo APS per il numero totale degli indirizzi. La definizione dell'array di remapping, attraverso la parola chiave *extern*, è necessaria affinché i suoi elementi possano essere riferiti all'interno del file Main.c, la sua effettiva dichiarazione si trova nel file NVM.c ed è quella vista nel frammento di codice 5.5.12.

```
/** Numero di APS_Address da salvare in memoria di programma. */
#define MAX_APS_ADDRESSES 8

/** apsAddressMapOffset indica quanti byte intercorrono tra la
 *  locazione iniziale di NVMBuffer e la locazione dove ha inizio
 *  la memorizzazione di apsAddress.
 */
#define apsAddressMapOffset 0

/** Dimensione in BYTE dello spazio da allocare in memoria di
 *  programma riservato agli APS_ADDRESS.
 */
#define apsAddressMapSize sizeof(APS_ADDRESS_MAP) *
MAX_APS_ADDRESSES

/** Definizione dell'array apsAddressMap. L'elemento i-esimo di
 *  questo array contiene un puntatore alla locazione in memoria
 *  di programma nella quale deve essere salvato l'i-esimo
 *  APS_Address.
 */
extern ROM BYTE * apsAddressMap[MAX_APS_ADDRESSES];
```

Cod. 5.5.15: Definizioni nel file NVM.h relative agli indirizzi a livello APS.

La funzione main, riportata nell'omonimo file, è visibile nel frammento di codice 5.5.16. Essa contiene il codice costituente il test, ovvero: una parte di dichiarazione delle variabili; una parte di inizializzazione in RAM dei dati da salvare; la scrittura di questi in memoria di programma; la lettura dalla memoria FLASH.

Sono dichiarate due variabili: *currentAPSAddress* di tipo APS_ADDRESS_MAP e l'array *APSAddressTable* dello stesso tipo. La prima è utilizzata come cursore all'interno dei cicli iterativi di scrittura e lettura; la seconda contiene tutti i dati che devono essere scritti in memoria di programma e pertanto necessita di un'inizializzazione, effettuata per mezzo del primo ciclo *for*, la cui i-esima iterazione imposta l'elemento i-esimo di *apsAddressTable* nel seguente modo: il campo *shortAddr* è impostato al valore 0x4FA0 + i, pertanto il primo elemento avrà valore 0x4FA0, il secondo 0x4FA1 e così via fino a 0x4FA7; mentre ognuno degli otto byte del campo *longAddr* è impostato in modo tale da avere come prima cifra esadecimale il

valore dell'indice dell'iterazione, mentre come seconda cifra esadecimale il valore della propria posizione all'interno di *longAddr*.

```
int main(){
      // Dichiarazione delle variabili.
      APS_ADDRESS_MAP apsAddressTable[MAX_APS_ADDRESSES];
      APS_ADDRESS_MAP currentAPSAddress;              // Dati da
leggere

      // Inizializzazione dei dati
      int i;
      for(i=0; i<MAX_APS_ADDRESSES; i++){
            apsAddressTable[i].shortAddr.Val = 0x4FA0 + i;
            apsAddressTable[i].longAddr.v[0] = 0x00 + (i<<4);
            apsAddressTable[i].longAddr.v[1] = 0x01 + (i<<4);
            apsAddressTable[i].longAddr.v[2] = 0x02 + (i<<4);
            apsAddressTable[i].longAddr.v[3] = 0x03 + (i<<4);
            apsAddressTable[i].longAddr.v[4] = 0x04 + (i<<4);
            apsAddressTable[i].longAddr.v[5] = 0x05 + (i<<4);
            apsAddressTable[i].longAddr.v[6] = 0x06 + (i<<4);
            apsAddressTable[i].longAddr.v[7] = 0x07 + (i<<4);
      }

      // Scrittura dei dati in memoria di programma.
      for(i=0; i<MAX_APS_ADDRESSES; i++){
          currentAPSAddress = apsAddressTable[i];
          NVMWrite((BYTE*)apsAddressMap[i],
                   (BYTE*)&currentAPSAddress,
sizeof(APS_ADDRESS_MAP));
      }

      // Lettura dei dati dalla memoria di programma.
      for(i=0; i<MAX_APS_ADDRESSES; i++){
          NVMRead((BYTE*)apsAddressMap[i],
                  (BYTE*)&currentAPSAddress,
sizeof(APS_ADDRESS_MAP));
      }
      return 0;
}
```

Cod. 5.5.16: Il codice per il test della scrittura e lettura degli indirizzi APS.

Il salvataggio in memoria di programma comporta la scrittura delle locazioni della FLASH come visibile in Fig. 5.5.3: i valori sono memorizzati a partire dall'indirizzo esadecimale 0x400, poiché questo è puntato dall'elemento iniziale di NVMBuffer, in base a quanto specificato dall'argomento del parametro aligned, di cui si è discusso nel paragrafo 5.3.2).

I primi dati memorizzati sono quelli relativi al campo *longAddr* e successivamente quelli relativi al campo *shortAddr*. Tale ordine è analogo a quello della dichiarazione del tipo di dato APS_ADDRESS_MAP, visibile nel frammento di codice 5.5.8.

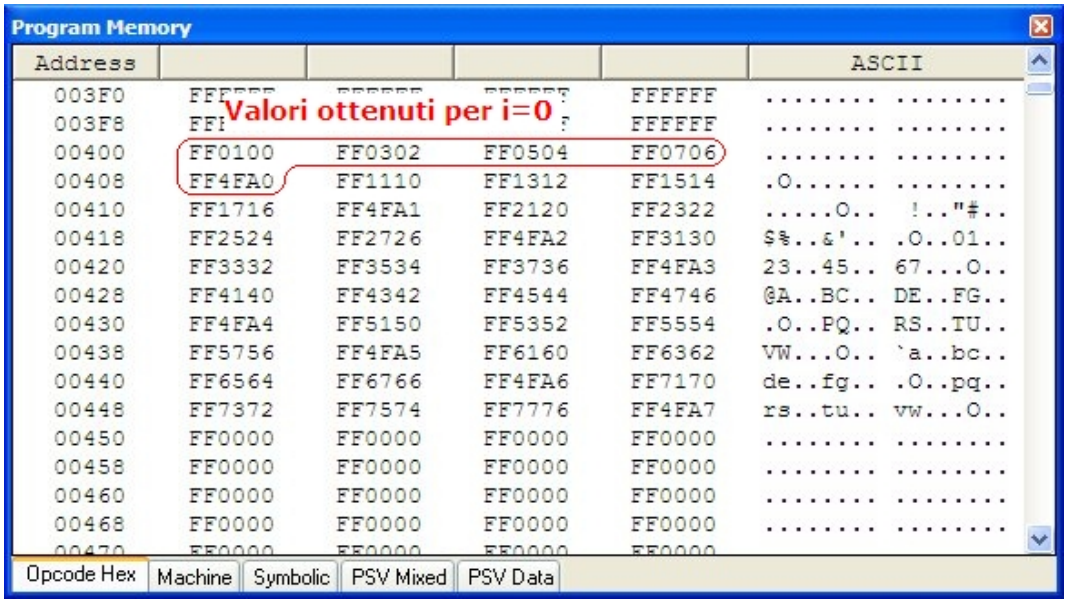

Fig. 5.5.3: Valori degli indirizzi a livello APS salvati nella memoria di programma.

In figura sono evidenziati i dati relativi al primo indirizzo: i valori di *longAddr* sono ordinati da 0 a 7 ed essendo di tipo byte compaiono in una parola di istruzione da destra verso sinistra, in quanto, come spiegato nella sezione 5.3.2 ed in particolare in Fig. 5.3.3, una parola di istruzione occupa due indirizzi e di questi il più piccolo corrisponde alla parte meno significativa. L'ultima parola di istruzione contiene il valore del campo *shortAddr*.

La funzione di lettura ha il compito di leggere i dati dalla FLASH e di trasferirli in RAM. Per mezzo del simulatore MPLAB SIM, integrato nell'ambiente di sviluppo, è possibile ad ogni iterazione osservare l'aggiornamento della variabile *currentAPSAddress*, in particolare, al termine dell'ultima iterazione i dati risultano quelli visibili in Fig. 5.5.4.

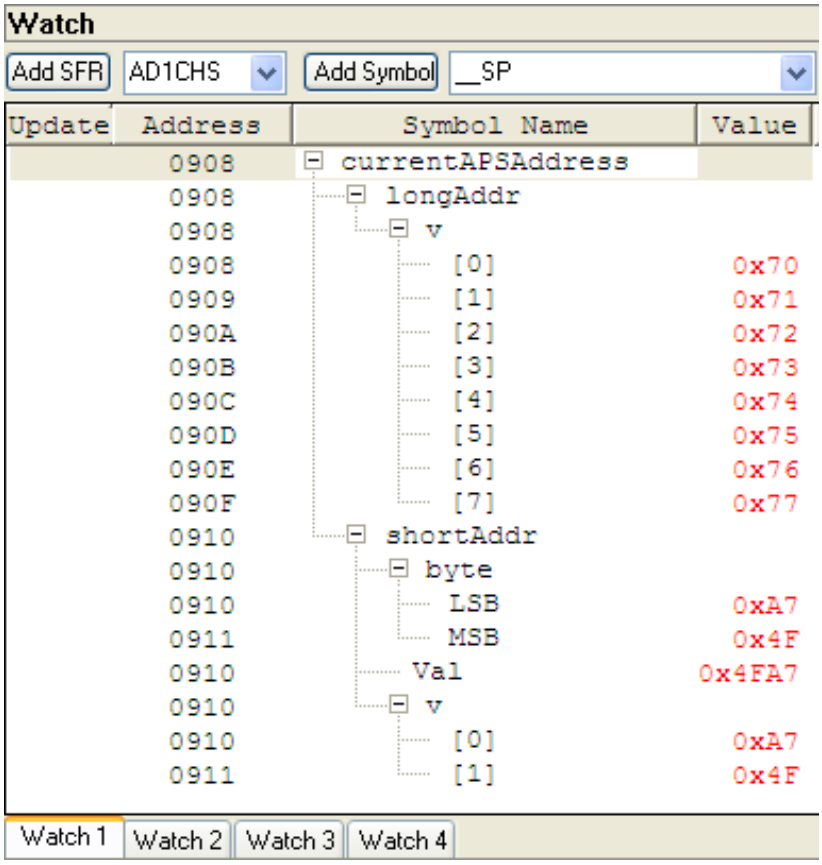

Fig. 5.5.4: Risultati prodotti dalla simulazione.

I valori sono espressi in esadecimale, infatti l'ambiente di sviluppo MPLAB utilizza la notazione 0x(valore in esadecimale) per rappresentare un numero in base 16. Si noti che effettivamente i valori degli elementi di *longAddr* hanno come prima cifra esadecimale il valore 7, che corrisponde al valore dell'indice del ciclo *for* per l'inizializzazione, all'ultima iterazione; come seconda cifra esadecimale la posizione che il singolo elemento occupa all'interno del campo *longAddr*. Anche il valore di *shortAddr*, corrisponde a quello atteso: 0x4FA7. Si noti dalla Fig. 5.5.4, inoltre, che i valori dei campi *byte* e *v*, che rappresentano la suddivisione in byte del campo *shortAddr*, sono in accordo con il valore del campo *Val*.

5.6 – Esplorazione del MAC ADDRESS

In seguito a quanto riportato nella sezione 5.4 – Utilizzo delle funzioni di storage, si procede alla ricerca delle funzioni PutMACAddress(x), GetMACAddress(x) GetMACAddressByte(x, y) definite nel file zNVM.h alle righe 277, 275 e 276 rispettivamente.

Esplorando il codice sorgente della libreria si nota che la memorizzazione del Mac Address avviene solamente in presenza della definizione della macro di configurazione *USE_EXTERNAL_NVM*. Tale aspetto potrebbe costituire un'ambiguità, in quanto il salvataggio dovrebbe essere richiesto anche se si evita l'uso della memoria EEPROM esterna.

Al fine di apprendere le motivazioni delle modifiche che saranno apportate al codice sorgente, risulta utile comprendere come viene costruito il Mac Address, il quale è formato da 8 byte ed è univoco per ogni dispositivo. Lo stack ZigBee definisce a tal fine otto costanti come visibile nel frammento di codice 5.6.1.

```
#define MAC_LONG_ADDR_BYTE0 0xaa
#define MAC_LONG_ADDR_BYTE1 0xaa
#define MAC_LONG_ADDR_BYTE2 0xaa
#define MAC_LONG_ADDR_BYTE3 0xaa
#define MAC_LONG_ADDR_BYTE4 0xaa
#define MAC_LONG_ADDR_BYTE5 0xaa
#define MAC_LONG_ADDR_BYTE6 0xaa
#define MAC_LONG_ADDR_BYTE7 0xaa
```

Cod. 5.6.1: Costanti che costituiscono il MAC ADDRESS.

L'indirizzo assegnato originariamente è arbitrario, il programmatore ha la facoltà di aggiornarlo, sia nel codice sorgente dello stack che in un apposito registro del circuito integrato MRF24J40, il quale costituisce il dispositivo principale del modulo ZigBee.

Per semplicità di notazione, lo stack completa la definizione del Mac Address introducendo la costante *macLongAddr* come si può osservare

nel frammento di codice 5.6.2, la quale è diversa a seconda della presenza della macro *USE_EXTERNAL_NVM*: nel caso in cui si deve salvare il Mac Address nella EEPROM, è definita come un array di byte; nel caso in cui il Mac Address deve essere salvato internamente al microcontrollore, è definita come un unica variabile di tipo LONG_ADDR, introdotto nel frammento di codice 5.5.7.

```
// Out Mac Address
#define MAC_ADDRESS_ROM_LOCATION 0x00002A

#if defined(USE_EXTERNAL_NVM) && defined (STORE_MAC_EXTERNAL)

    BYTE macLongAddrByte[8] = {
                  MAC_LONG_ADDR_BYTE0, MAC_LONG_ADDR_BYTE1,
                  MAC_LONG_ADDR_BYTE2, MAC_LONG_ADDR_BYTE3,
                  MAC_LONG_ADDR_BYTE4, MAC_LONG_ADDR_BYTE5,
                  MAC_LONG_ADDR_BYTE6, MAC_LONG_ADDR_BYTE7};

#else

    ROM LONG_ADDR macLongAddr = {{
                  MAC_LONG_ADDR_BYTE0, MAC_LONG_ADDR_BYTE1,
                  MAC_LONG_ADDR_BYTE2, MAC_LONG_ADDR_BYTE2,
                  MAC_LONG_ADDR_BYTE4, MAC_LONG_ADDR_BYTE5,
                  MAC_LONG_ADDR_BYTE6, MAC_LONG_ADDR_BYTE7}}

#endif
```

Cod. 5.6.2: Definizione del MAC ADDRESS.

Il presente lavoro di tesi è proceduto nella direzione di non affrontare l'ambiguità sopracitata, ma implementando le funzioni di storage relative al Mac Address, verificando come le funzioni NVMRead ed NVMWrite, aggiornate nei paragrafi 5.3.3 e 5.3.4, si adattano al contesto, in modo tale che qualora sia necessario il loro impiego esse siano disponibili.

5.6.1 – Funzione PutMACAddress

La funzione PutMACAddress(x), definita originariamente come nel frammento di codice 5.6.3 (tratto dall'elenco del Cod. 5.4.1), si occupa della memorizzazione dell'indirizzo del livello di accesso al mezzo

(riferimento alla Fig. 1.2.2) all'interno della memoria di programma del microcontrollore.

```
#define PutMACAddress(x)
NVMWrite((NVM_ADDR*)&macLongAddr, (BYTE*)x, sizeof(LONG_ADDR))
```

Cod. 5.6.3: Definizione originale della funzione PutMACAddress.

```
PutMACAddress(&macLongAddr);
```

Cod. 5.6.5: Prototipo delle chiamate a PutMACAddress.

Osservando il codice del frammento 5.6.2 si nota che il parametro da passare come argomento alla chiamata della funzione è la variabile *macLongAddr* (ovvero le chiamate a PutMACAddress sono tutte del tipo mostrato nel frammento di codice 5.6.5), e che la quantità di spazio richiesto in memoria di programma è di 8 byte; è pertanto possibile procedere alla definizione dell'offset su NVMBuffer, della dimensione e

```
/** macAddressMapOffset indica quanti byte intercorrono tra la
 *  locazione iniziale di NVMBuffer e la locazione dove ha inizio
 *  la memorizzazione di apsAddressMap.
 */
#define macAddressMapOffset   apsAddressMapOffset +
                              apsAddressMapSize

/** Dimensione in BYTE dello spazio da allocare in memoria di
 *  programma riservato ai MAC_ADDRESS.
 */
#define macAddressMapSize sizeof(LONG_ADDR)

/** Definizione della variabile macAddressMap. Questa variabile
 *  contiene un puntatore alla locazione in memoria di programma
 *  nella quale deve essere salvato il MAC_Address.
*/
extern ROM BYTE * macAddressMap;
```

Cod. 5.6.4: Definizioni di offset, dimensione e variabile di remapping per il MAC ADDRESS.

della variabile di remapping, al fine di ottenere i riferimenti alla memoria di programma, analogamente a quanto già visto per gli indirizzi a livello APS, nel paragrafo 5.5.3. Tali definizioni sono scritte nel file NVM.h e riportate nel frammento di codice 5.6.4.

Si osservi che la definizione dell'offset tiene conto dello stesso offset e della dimensione degli indirizzi APS. La dimensione, come accennato in precedenza, è di 8 byte, ottenuta in questo ambito tramite l'operatore *sizeof* del linguaggio C. Nel file NVM.c è necessario dichiarare la variabile di remapping, *macAddressMap*, già definita nel relativo file di intestazione, inizializzandola correttamente all'opportuna locazione di NVMBuffer, come visibile nel frammento di codice 5.6.6.

```
/** Variabile di remapping del MAC ADDRESS. */
ROM BYTE * macAddressMap = NVMBuffer + macAddressMapOffset;
```

Cod. 5.6.6: Dichiarazione e inizializzazione della variabile di remapping del MAC ADDRESS.

L'inizializzazione fa uso della definizione del relativo offset: in tal modo *macAddressMap* punta alla prima locazione libera dopo gli indirizzi APS.

5.6.2 - Funzione GetMACAddress

La funzione GetMACAddress si occupa della lettura del Mac Address dalla memoria di programma del microcontrollore; originariamente è definita come visibile nel frammento di codice 5.8.1, tratto dall'elenco di Cod. 5.4.2.

```
#define GetMACAddress(x)
NVMRead((BYTE *)x, (ROM void*)&macLongAddr, sizeof(LONG_ADDR))
```

Cod. 5.6.7: Definizione originale della funzione GetMACAddress.

Analogamente alla duale funzione di memorizzazione, anch'essa accetta un solo parametro, che contiene il riferimento alla locazione in RAM nella quale deve essere trasferito il dato letto; ovvero il prototipo delle chiamate a GetMACAddress è quello visibile nel frammento di codice 5.6.8.

```
GetMACAddress(&longAddressVar);
```

Cod. 5.6.8: Prototipo delle chiamate a GetMACAddress.

Si osservi che tale funzione non può agire sulla variabile *macLongAddr* (dichiarata nel codice 5.6.2) in quanto questa è una costante.

5.6.3 - Funzione GetMacAddressByte

La funzione GetMACAddressByte ha il compito di leggere un singolo byte dell'indirizzo di accesso al mezzo dalla memoria di programma e di trasferirne il valore in una locazione in RAM.

La definizione originale è visibile nel frammento di codice 5.6.9 tratto dall'elenco di Cod. 5.4.2.

```
#define GetMACAddressByte(y, x)
NVMRead((BYTE *)x, (ROM void*)((int)&macLongAddr + (int)y), 1)
```

Cod. 5.6.9: Definizione originale della funzione GetMACAddressByte.

Diversamente dalla funzione GetMACAddress, per tale funzione è necessario specificare due parametri: il primo indica quale degli otto byte del Mac Address si vuole ottenere; il secondo il riferimento alla locazione in RAM nella quale trasferire il dato letto dalla FLASH. Il prototipo della chiamata a GetMACAddressByte è visibile nel frammento di codice 5.6.10.

```
GetMacAddressByte(i, &byteVar);
```

Cod. 5.6.10: Prototipo della chiamata a GetMACAddressByte.

Tale codice trasferisce il byte i-esimo del Mac Address dalla memoria di programma alla locazione in RAM riferita dalla variabile *byteVar* (di tipo BYTE). Ovviamente il valore della variabile *i* deve essere compreso tra 0 e 7.

5.6.4 - Considerazioni

Nei precedenti paragrafi sono state presentate le funzioni di salvataggio e di lettura del Mac Address verso e dalla memoria di programma; in particolare si è visto il loro comportamento ed i prototipi alle relative

chiamate. Nel paragrafo 5.6.1 inoltre, sono stati introdotti: offset, dimensioni e variabile di remapping per permettere la corretta gestione della memoria di programma, al fine di riservarne lo spazio necessario al Mac Address. In questa sezione, di tali funzioni saranno presentati gli aggiornamenti delle definizioni, pertanto è utile tenere presente l'interfaccia di NVMWrite e di NVMRead (riscritte nei paragrafi 5.3.3 e 5.3.4) riportate nello pseudo-codice nel riquadro 5.6.11.

```
NVMWrite(Locazione in FLASH, Locazione in RAM, N° di byte)

NVMRead(Locazione in FLASH, Locazione in RAM, N° di byte)
```

Cod. 5.6.11: Interfacce delle funzioni di storage.

Secondo quanto visto nel paragrafo 5.6.1, la funzione PutMACAddress può essere ridefinita come nel frammento di codice 5.6.12.

```
/** Definizione della funzione PutMACAddress che salva il
 *  MAC ADDRESS nella memoria FLASH.
 */
#define PutMACAddress(x)
NVMWrite((BYTE*)macAddressMap, (BYTE*)x, sizeof(LONG_ADDR));
```

Cod. 5.6.12: Definizione aggiornata della funzione PutMACAddress.

Rispetto alla definizione originale, riportata nel riquadro di codice 5.6.3, si è mutato il riferimento alla locazione in memoria di programma: da *macLongAddr* a *macAddressMap*. Questo aspetto, nella definizione originale, infatti, costituiva un errore, in quanto *macLongAddr* è allocata in RAM anziché in memoria di programma.

Secondo quanto riportato nel paragrafo 5.6.2, la funzione GetMACAddress può essere ridefinita come nel frammento di codice

```
/** Definizione della funzione GetMACAddress che legge il
 *  MAC ADDRESS dalla memoria FLASH.
 */
#define GetMACAddress(x)
NVMRead((BYTE*)macAddressMap, (BYTE*)x, sizeof(BYTE));
```

Cod. 5.6.13: Definizione aggiornata della funzione GetMACAddress.

5.6.13.

Rispetto alla definizione originale, riportata nel riquadro di codice 5.8.1, si è mutato il riferimento alla locazione in memoria di programma, analogamente a quanto appena visto per la funzione PutMACAddress, e l'ordine dei primi due argomenti (questo cambiamento è la conseguenza dell'inversione dei parametri nella funzione NVMRead, aggiornata nel paragrafo 5.3.3).

Infine, la funzione GetMACAddressByte, in base a quanto riportato nel paragrafo 5.6.3, può essere ridefinita come nel frammento di codice 5.6.14.

```
/** Definizione della funzione GetMACAddressByte che legge un
 *  singolo byte del MAC ADDRESS dalla memoria FLASH.
 *    @param x: indice (offset) del byte del MAC ADDRESS che
 *              si intende leggere.
 *    @param y: puntatore ad una locazione in RAM nella quale
 *              scrivere il dato letto.
 */
#define GetMACAddressByte(x, y)
NVMRead((BYTE*)macAddressMap+x, (BYTE*)y, sizeof(BYTE));
```

Cod. 5.6.14: Definizione aggiornata della funzione GetMACAddressByte.

Rispetto alla definizione originale, riportata nel riquadro di codice 5.6.9, si è mutato il riferimento alla locazione in memoria di programma, analogamente alle funzioni GetMACAddress e PutMACAddress, e l'ordine degli argomenti (in conseguenza all'inversione dei parametri della funzione NVMRead). Si osservi che il riferimento al byte richiesto del Mac Address avviene sommando all'indirizzo iniziale di *macAddressMap* il valore dell'argomento x, il quale rappresenta proprio l'indice del byte richiesto.

Le definizioni appena introdotte altro non sono che una mappatura delle funzioni NVMRead ed NVMWrite, e sono tutte inserite nel file NVM.h, insieme alle definizioni di offset, dimensioni e variabile di remapping.

5.6.5 – Codice di esempio

Al fine di testare la validità delle ipotesi e delle considerazioni trattate nei paragrafi 5.6.1, 5.6.2, 5.6.3 e 5.6.4, si amplia il progetto creato appositamente (introdotto nel paragrafo 5.5.4) con i frammenti di codice presentati nei paragrafi sopracitati e con una parte di codice, inserita nella funzione main, che utilizza le funzioni di storage relative al Mac Address.

Quest'ultima è visibile nel frammento di codice 5.6.15, nel quale si è omessa la parte relativa agli indirizzi a livello APS e contiene: una parte di dichiarazione delle variabili; la scrittura del Mac Address in memoria di programma; la lettura di questo dalla memoria FLASH.

```
int main(){
    // Dichiarazione delle variabili
    ROM LONG_ADDR macLongAddr =     // Valore del MAC ADDRESS
        {{0xF1, 0xE2, 0xD3, 0xC4, 0xB5, 0xA6, 0x97, 0x88}};

    // Scrittura del MAC ADDRESS in memoria di programma.
    PutMACAddress(macLongAddr);

    // Lettura del MAC ADDRESS dalla memoria di programma.
    LONG_ADDR temp;     // Variabile temporanea per la lettura
    GetMACAddress(&temp);

    return 0;
}
```

Cod. 5.6.15: Parte integrante nella funzione main per testare le funzioni di storage del MAC ADDRESS.

Nel codice si dichiara una costante: *macLongAddr* di tipo LONG_ADDR, la quale contiene il valore del Mac Address da salvare.

Il salvataggio in memoria di programma comporta la scrittura delle locazioni della FLASH come visibile in Fig. 5.6.1: i valori sono memorizzati a partire dall'indirizzo esadecimale 0x450, poiché questa è la prima locazione libera dopo la memorizzazione degli indirizzi a livello APS. Tale allineamento si riesce ad ottenere grazie alla gestione degli indirizzamenti tramite offset e dimensioni, per ogni informazione da salvare in memoria.

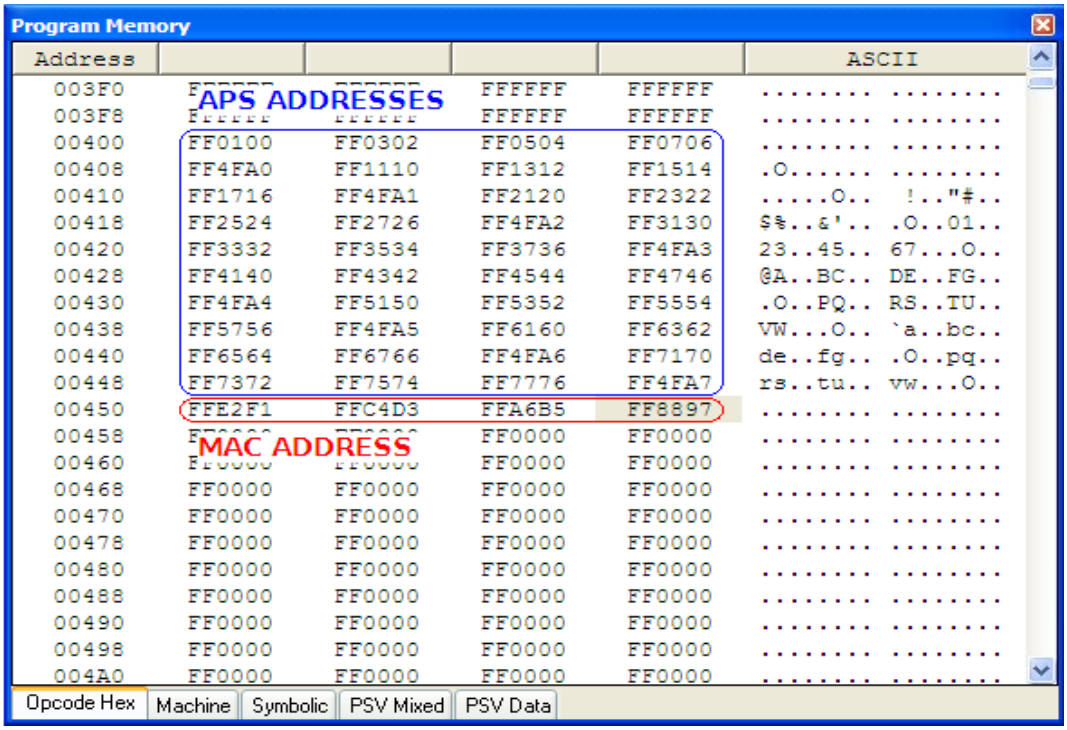

Fig. 5.6.1: Valore del MAC ADDRESS salvato nella memoria di programma.

Confrontando i valori della figura con quelli della costante *macLongAddr* del codice 5.6.15, si può affermare che la scrittura avviene correttamente. Essendo di tipo byte, i dati compaiono in una parola di istruzione da destra verso sinistra, in quanto, come spiegato nella sezione 5.3.2 ed in particolare in Fig. 5.3.3, una parola di istruzione occupa due indirizzi e di questi il più piccolo corrisponde alla parte meno significativa.

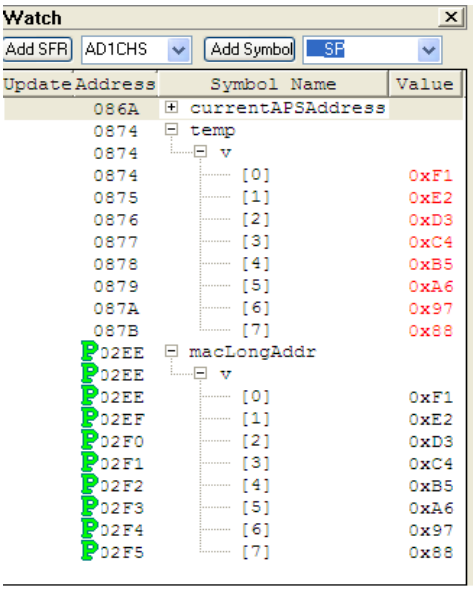

Fig. 5.6.2: Risultati prodotti dalla simulazione.

La funzione di lettura ha il compito di leggere i dati dalla FLASH e di trasferirli in RAM. Per mezzo del simulatore MPLAB SIM, integrato nell'ambiente di sviluppo, è possibile osservare l'aggiornamento della variabile *temp*, la quale a lettura ultimata assume il valore visibile in Fig. 5.6.2,

in accordo con i valori salvati, visibili in Fig. 5.6.1 e con quelli impostati nel codice 5.6.15.

5.7 – Esplorazione di APSAddressValidityKey

In seguito a quanto riportato nella sezione 5.4 – Utilizzo delle funzioni di storage, si procede alla ricerca delle funzioni PutAPSAddressValidityKey(x) e GetAPSAddressValidityKey(x) definite nel file zNVM.h alle righe 286 e 284 rispettivamente.

5.7.1 – Funzione PutAPSAddressValidityKey

La funzione in esame si occupa di memorizzare la chiave di validazione per gli indirizzi a livello APS all'interno della memoria di programma del microcontrollore. La sua definizione originale, tratta dall'elenco di Cod. 5.4.1 è visibile nel frammento di codice 5.7.1:

```
#define PutAPSAddressValidityKey(x)
NVMWrite((NVM_ADDR*)&apsAddressMapValidityKey, (BYTE*)x,
          sizeof(WORD))
```

Cod. 5.7.1: Definizione originale della funzione PutAPSAddressValidityKey.

dove la variabile *apsAddressMapValidityKey* dovrebbe essere il riferimento alla locazione nella memoria FLASH nella quale sarà memorizzata la chiave di validazione; in realtà nella sua dichiarazione originaria non è specificato l'attributo *space(prog)* e pertanto tale variabile è allocata in RAM.

La funzione accetta un solo argomento: il riferimento alla locazione in RAM nella quale si trova il valore della chiave di validazione da memorizzare. Pertanto il prototipo della sua chiamata è quello mostrato nel riquadro di codice 5.7.2, dove la variabile *ValidKey* è una WORD

dichiarata a livello locale. La ridefinizione della funzione dovrà essere compatibile con tale prototipo, al fine di non dover effettuare alcuna modifica di massa all'interno dello stack.

```
PutAPSAddressValidityKey(&ValidKey);
```

Cod. 5.7.2: Prototipo della chiamata alla funzione PutAPSAddressValidityKey

Dalla definizione di Cod. 5.7.1 e dal tipo della variabile *ValidKey* è possibile affermare che lo spazio in memoria di programma da riservare alla chiave di validazione è di 2 byte.

5.7.2 – Funzione GetAPSAddressValidityKey

La funzione in esame si occupa di leggere la chiave di validazione per gli indirizzi a livello APS dalla memoria di programma del microcontrollore e di trasferirne il valore in RAM. La sua definizione originale, tratta dall'elenco di Cod. 5.4.2 è visibile nel frammento di codice 5.7.3:

```
#define GetAPSAddressValidityKey(x)
NVMRead((BYTE*)apsAddressMapValidityKey, (BYTE*)x, sizeof(WORD))
```

Cod. 5.7.3: Definizione originale della funzione GetAPSAddressValidityKey.

La funzione accetta un solo argomento: il riferimento alla locazione in RAM nella quale trasferire il valore della chiave di validazione dalla memoria di programma, pertanto il prototipo della sua chiamata è quello mostrato nel riquadro di codice 5.7.4 ed è del tutto analogo a quello della funzione PutAPSAddressValidityKey.

```
GetAPSAddressValidityKey(&ValidKey);
```

Cod. 5.7.4: Prototipo della chiamata alla funzione GetAPSAddressValidityKey

5.7.3 - Considerazioni

Nei precedenti paragrafi sono state presentate le funzioni di salvataggio e di lettura della chiave di validazione degli indirizzi a livello APS, verso e dalla memoria di programma; in particolare si è visto il loro comportamento ed i prototipi alle relative chiamate. In questa sezione ci si occuperà delle dichiarazioni dei parametri necessari alla corretta gestione della memoria di programma al fine di riservarne lo spazio necessario alla chiave di validazione e saranno presentati gli aggiornamenti delle definizioni delle funzioni PutAPSAddressValidityKey e GetAPSAddressValidityKey.

Osservando la Fig. 5.6.1 si nota che la prima locazione libera in memoria di programma è all'indirizzo 0x458. Per garantire la scrittura della chiave di validazione in tale locazione è necessario dichiarare l'offset tenendo conto dello stesso offset e della dimensione del Mac Address. Inoltre è possibile procedere alla definizione della dimensione della chiave di validazione, in quanto, come osservato nel paragrafo

```
/** apsAddressMapValidityKeyOffset indica quanti byte intercorrono
 *  tra la locazione iniziale di NVMBuffer e la locazione dove ha
 *  inizio la memorizzazione di apsAddressValidityKey.
*/
#define apsAddressMapValidityKeyOffset macAddressMapOffset +
                                       macAddressMapSize

/** Dimensione in BYTE dello spazio da allocare in memoria di
 *  programma riservato ad APS ADDRESS VALIDITY KEY.
 */
#define apsAddressMapValidityKeySize sizeof(WORD)

/** Definizione della variabile apsAddressMapValidityKey. La
 *  variabile contiene è un puntatore alla locazione in memoria di
 *  programma nella quale deve essere salvata la APS ADDRESS
 *  VALIDITY KEY.
 */
extern BYTE * apsAddressMapValidityKey ;
```

Cod. 5.7.5: Definizione dei parametri per la memorizzazione della chiave nella memoria di programma.

5.7.1, lo spazio di memoria richiesto è di 2 byte. Nel file NVM.h è necessario definire la variabile di remapping, attraverso la parola chiave *extern*, al fine di poterne fare riferimento all'interno del file Main.c. Tali definizioni sono visibili nel frammento di codice 5.7.5.

La dichiarazione della variabile di remapping, avviene nel file NVM.c ed è inizializzata alle opportune locazioni di memoria di programma grazie all'offset della chiave di validazione, come visibile nel frammento di codice 5.7.6.

```
ROM BYTE * apsAddressMapValidityKey =
            NVMBuffer + apsAddressMapValidityKeyOffset;
```

Cod. 5.7.6: Dichiarazione e inizializzazione della variabile di remapping della chiave di validazione.

Una volta completate le definizioni dei parametri per il salvataggio della chiave nella memoria di programma è possibile procedere alla ridefinizione delle funzioni di lettura e scrittura.

Secondo quanto visto nel paragrafo 5.7.1, la funzione PutAPSAddressValidityKey può essere ridefinita come nel frammento di codice 5.7.7.

```
#define PutAPSAddressValidityKey(x)
NVMWrite((BYTE*)apsAddressMapValidityKey, (BYTE*)x, sizeof(WORD));
```

Cod. 5.7.7: Definizione aggiornata della funzione PutAPSAddressValidityKey.

Rispetto alla definizione originale, riportata nel riquadro di codice 5.7.1, si è omesso il simbolo "&" avanti al primo parametro. Questa modifica si rende necessaria in quanto la variabile di remapping è stata dichiarata come puntatore.

Secondo quanto riportato nel paragrafo 5.7.2, la funzione GetAPSAddressValidityKey può essere ridefinita come nel frammento di

```
#define GetAPSAddressValidityKey(x)
NVMRead((BYTE*)apsAddressMapValidityKey, (BYTE*)x, sizeof(WORD));
```

Cod. 5.7.8: Definizione aggiornata della funzione GetAPSAddressValidityKey.

codice 5.7.8. Rispetto alla definizione originale, riportata nel riquadro di codice 5.7.3 non è stata introdotta alcuna mutazione.

Gli aggiornamenti non comportano alcuna modifica al prototipo delle chiamate, pertanto nel codice dello stack non è necessario effettuare alcuna ulteriore modifica.

5.7.4 - Codice di esempio

Al fine di testare la validità delle ipotesi e delle considerazioni trattate nei paragrafi 5.7.1, 5.7.2 e 5.7.3, si amplia il progetto creato appositamente (introdotto nel paragrafo 5.5.4) con i frammenti di codice presentati nei paragrafi sopracitati e con una parte di codice, inserita nella funzione main, che utilizza le funzioni di storage relative alla chiave di validazione.

Tale codice è visibile nel frammento 5.7.9, nel quale sono state omesse le parti relative alle informazioni trattate precedentemente e contiene: una parte di dichiarazione delle variabili; la scrittura della chiave di validazione in memoria di programma; la lettura di questa dalla memoria FLASH.

```
int main(){
    // Dichiarazione delle variabili.
    WORD ValidKey = 0xABCD; // Valore della APSAddressValidityKey

    // Scrittura della APS Address Validity Key.
    PutAPSAddressValidityKey(&ValidKey);

    // Lettura della APS ADDRESS Validity key.
    ValidKey = 0;
    GetAPSAddressValidityKey(&ValidKey);

    return 0;
}
```

Cod. 5.7.9: Parte integrante nella funzione main per testare le funzioni di storage della chiave di validazione.

In tale codice si dichiara una sola variabile: *ValidKey* di tipo BYTE, inizializzandola al valore della chiave di validazione da salvare.

Il salvataggio in memoria di programma comporta la scrittura delle locazioni della FLASH come visibile in Fig. 5.7.1: i valori sono memorizzati a partire dall'indirizzo esadecimale 0x458, poiché questa è la prima locazione libera dopo la memorizzazione del Mac Address. Tale allineamento, come osservato nel paragrafo 5.7.3, è ottenuto grazie

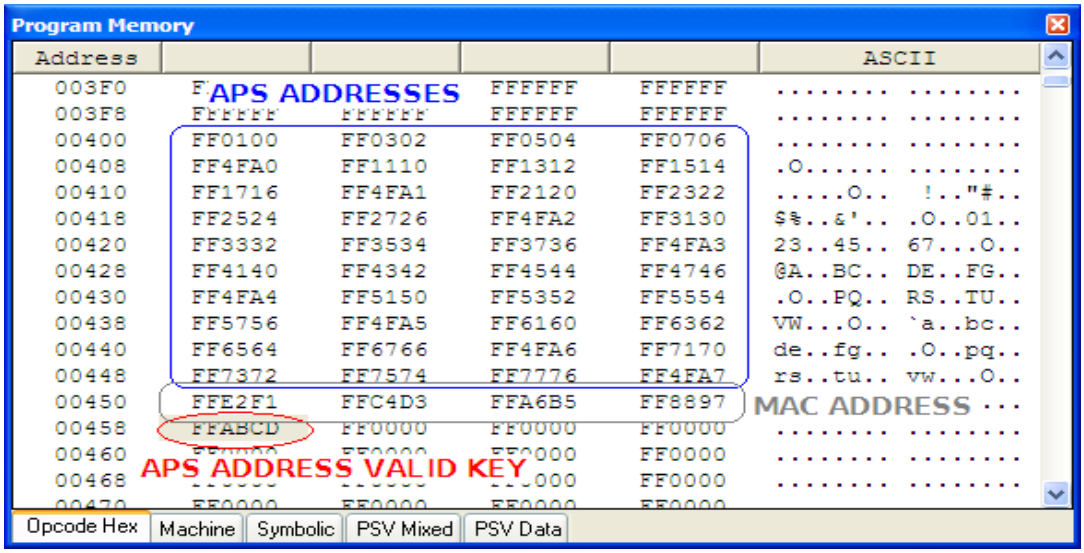

Fig. 5.7.1: Valore del MAC ADDRESS salvato nella memoria di programma.

alla gestione degli indirizzamenti tramite offset e dimensioni relativi ad ogni informazione da salvare in memoria. Confrontando i valori della figura con quelli della variabile *ValidKey* del codice 5.7.9, si può affermare che la scrittura avviene correttamente.

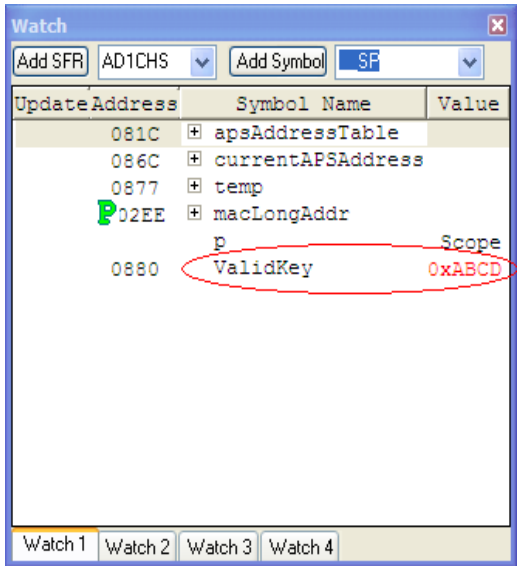

Fig. 5.7.2: Risultati prodotti dalla simulazione.

La funzione di lettura ha il compito di leggere i dati dalla FLASH e di trasferirli in RAM. Per mezzo del simulatore MPLAB SIM, integrato nell'ambiente di sviluppo, è possibile osservare l'aggiornamento della variabile *ValidKey*,

la quale a lettura ultimata assume il valore visibile in Fig. 5.7.2, in accordo con il valore salvato, visibile in Fig. 5.7.1 e con quello impostato nel codice 5.7.9.

5.8 – Esplorazione dei Binding Record

In seguito a quanto riportato nella sezione 5.4 – Utilizzo delle funzioni di storage, si procede all'esplorazione delle funzioni PutBindingRecord(x, y) e GetBindingRecord(x, y) definite nel file zNVM.h alle righe 302 e 298 rispettivamente.

Al fine di apprendere il funzionamento delle funzioni di storage, risulta utile esaminare le definizioni dei tipi WORD_VAL e BINDING_RECORD.

```
#if defined(__GNUC__)
    #define __EXTENSION __extension__
#else
    #define __EXTENSION
#endif
```

Cod. 5.8.1: Definizione della macro __EXTENSION.

Il tipo WORD_VAL rappresenta un dato di 2 byte al quale è possibile accedere in diversi modi: ai singoli byte attraverso gli array *v* e *byte*; all'intero dato attraverso il campo *Val*; ai singoli bit attraverso la struttura *bits*. Prima di esaminare la sua definizione, visibile nel frammento di codice 5.8.3, è utile osservare le definizioni delle macro *__EXTENSION* e *__PACKED*, visibili nei frammenti 5.8.1 e 5.8.2 rispettivamente: la prima è un'opzione del compilatore GCC; la seconda è stata introdotta solamente per migliorare la leggibilità del codice.

La sintassi introdotta nella struttura *bits* del codice 5.8.3 indica al compilatore quanto segue: il numero dopo il simbolo ":" specifica a quanti bit accede il corrispondente campo; lo stesso campo accede a tanti bit iniziando da quello successivo all'ultimo a cui accede il campo precedente. Questo concetto è

```
#if !defined(__PACKED)
    #define __PACKED
#endif
```

Cod. 5.8.2: Definizione della macro __PACKED.

schematizzato in Fig. 5.8.1, dove ogni campo accede ad un singolo bit: il campo *b0* accede al bit 0, il campo *b1* accede al bit 1 in quanto questo è quello successivo all'ultimo a cui accede *b0*. Il procedimento si ripete analogamente fino al bit 15.

```
typedef union _WORD_VAL{
    BYTE v[2] __PACKED;
    WORD Val;

    struct __PACKED{
        BYTE LSB;
        BYTE MSB;
    } byte;

    struct __PACKED{
        __EXTENSION BYTE b0:1;
        __EXTENSION BYTE b1:1;
        __EXTENSION BYTE b2:1;
        __EXTENSION BYTE b3:1;
        __EXTENSION BYTE b4:1;
        __EXTENSION BYTE b5:1;
        __EXTENSION BYTE b6:1;
        __EXTENSION BYTE b7:1;
        __EXTENSION BYTE b8:1;
        __EXTENSION BYTE b9:1;
        __EXTENSION BYTE b10:1;
        __EXTENSION BYTE b11:1;
        __EXTENSION BYTE b12:1;
        __EXTENSION BYTE b13:1;
        __EXTENSION BYTE b14:1;
        __EXTENSION BYTE b15:1;
    } bits;

} WORD_VAL;
```

Cod. 5.8.3: Definizione del tipo WORD_VAL.

La definizione del tipo BINDING_RECORD consiste in una struttura che nell'insieme occupa uno spazio in memoria di 6 byte, essendo composta da due campi di 1 byte ciascuno e da due campi di 2 byte ciascuno, ma siccome l'architettura del microcontrollore è a 16 bit, un BINDING_RECORD occupa uno spazio totale di 8 byte.

La definizione di tale tipo di dato è visibile nel frammento di codice 5.8.4.

```
typedef struct _BINDING_RECORD{
    SHORT_ADDR    shortAddr;
    BYTE          endPoint;
    WORD_VAL      clusterID;
    BYTE          nextBindingRecord;
} BINDING_RECORD;
```

Cod. 5.8.4: Definizione del tipo BINDING_RECORD.

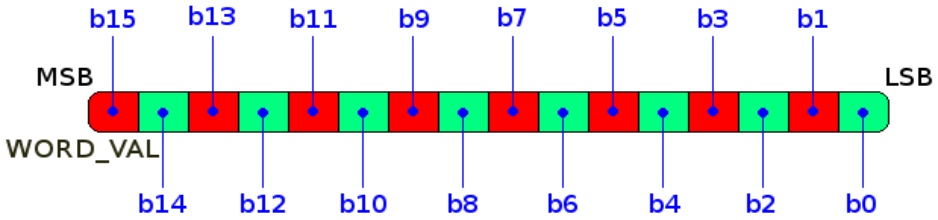

Fig. 5.8.1: Accessi ai singoli bit di un dato.

5.8.1 – Funzione PutBindingRecord

La funzione PutBindingRecord(x, y), definita originariamente nel frammento di codice 5.8.5 (tratto dall'elenco di Cod. 5.4.1), si occupa della memorizzazione di un record di binding all'interno della memoria di programma del microcontrollore.

```
#define PutBindingRecord(x, y )
NVMWrite((NVM_ADDR *)x, (BYTE *)y, sizeof(BINDING_RECORD))
```

Cod. 5.8.5: Definizione originale della funzione PutBindingRecord.

Osservando il codice del frammento 5.8.5 si nota che i parametri da passare come argomento alla chiamata della funzione sono: la locazione in memoria di programma nella quale salvare il record di binding e una variabile di tipo BINDING_RECORD, contenente i dati da memorizzare. Ovvero le chiamate a PutBindingRecord sono tutte del tipo mostrato nel frammento di codice 5.8.6.

```
while(...){ // Inizio di un generico ciclo
    // Inizializzazione di currentBindingRecord.
    currentBindingRecord = ...;
    // Inizializzazione di pCurrentBindingRecord.
    pCurrentBindingRecord = &apsBindingTable[index];
    // Scrittura in memoria di programma.
    PutBindingRecord(pCurrentBindingRecord,&currentBindingRecord);
}
```

Cod. 5.8.6: Prototipo delle chiamate a PutBindingRecord.

In questo codice la variabile *currentBindingRecord*, di tipo BINDING_RECORD, è utilizzata come cursore e contiene il valore del record di binding che deve essere salvato in memoria. La variabile *pCurrentBindingRecord* rappresenta un puntatore alla locazione nella quale salvare il record di binding ed è inizializzato attraverso l'array *apsBindingTable*. Tale array è di remapping ma nella dichiarazione originale è stato commesso l'errore di non specificare l'attributo *space(prog)* e pertanto è allocato in RAM anziché in memoria di programma, come visibile nel frammento di codice 5.8.7.

```
ROM BINDING_RECORD apsBindingTable[MAX_BINDINGS] = {0x00};
```

Cod. 5.8.7: Dichiarazione originale dell'array di remapping.

Il ciclo iterativo si ripete tante volte quanti sono i record da salvare, e ad ogni iterazione viene aggiornato il valore di *index* per riferire la corretta allocazione in memoria FLASH.

La costante *MAX_BINDINGS* rappresenta il numero di record di binding

```
/** Numero di record di binding da salvare in memoria di
 *  programma.
*/
#define MAX_BINDINGS 15

/** bindingRecordOffset indica quanti byte intercorrono tra la
 *  locazione iniziale di NVMBuffer e la locazione dove ha inizio
 *  la memorizzazione dei record di binding.
 */
#define bindingRecordOffset   apsAddressMapValidityKeyOffset +
                              apsAddressMapValidityKeySize

/** Dimensione in BYTE dello spazio da allocare in memoria di
 *  programma riservato ai MAC_ADDRESS.
 */
#define bindingRecordMapSize sizeof(BINDING_RECORD) * MAX_BINDINGS

/** Definizione della variabile macAddressMap. Questa variabile
 *  contiene un puntatore alla locazione in memoria di programma
 *  nella quale deve essere salvato il MAC_Address.
*/
extern ROM BYTE * apsBindingTable[MAX_BINDINGS];
```

Cod. 5.8.8: Definizioni nel file NVM.h relative alle informazioni sui record di binding.

da salvare in memoria, il suo valore è 15. Si evince dunque che lo spazio totale richiesto per salvare tutti i record di binding è 120 byte, ottenuto moltiplicando la dimensione di un singolo record – di 8 byte come visto nel paragrafo 5.8 – per il valore di *MAX_BINDINGS*. È pertanto possibile procedere alle definizioni dell'offset su NVMBuffer, della dimensione e della variabile di remapping al fine di ottenere i riferimenti alla memoria di programma, analogamente a quanto già visto per le informazioni trattate nei paragrafi precedenti. Tali definizioni sono scritte nel file NVM.h e riportate nel frammento di codice 5.8.8.

Si osservi che la definizione dell'offset tiene conto dello stesso offset e della dimensione della chiave di validazione degli indirizzi APS. Nel file NVM.c è necessario dichiarare la variabile di remapping, *apsBindingTable*, definita nel file di intestazione, inizializzandola correttamente alle opportune locazioni di NVMBuffer, come visibile nel frammento di codice 5.8.9.

L'inizializzazione fa uso della definizione del relativo offset: in tal modo *apsBindingTable* punta alla prima locazione libera dopo la chiave di

```
/** Variabile di remapping del MAC ADDRESS. */
ROM BYTE * apsBindingTable[MAX_BINDINGS] = {
      NVMBuffer + bindingRecordOffset + 0*sizeof(BINDING_RECORD),
      NVMBuffer + bindingRecordOffset + 1*sizeof(BINDING_RECORD),
      NVMBuffer + bindingRecordOffset + 2*sizeof(BINDING_RECORD),
      NVMBuffer + bindingRecordOffset + 3*sizeof(BINDING_RECORD),
      NVMBuffer + bindingRecordOffset + 4*sizeof(BINDING_RECORD),
      NVMBuffer + bindingRecordOffset + 5*sizeof(BINDING_RECORD),
      NVMBuffer + bindingRecordOffset + 6*sizeof(BINDING_RECORD),
      NVMBuffer + bindingRecordOffset + 7*sizeof(BINDING_RECORD),
      NVMBuffer + bindingRecordOffset + 8*sizeof(BINDING_RECORD),
      NVMBuffer + bindingRecordOffset + 9*sizeof(BINDING_RECORD),
      NVMBuffer + bindingRecordOffset + 10*sizeof(BINDING_RECORD),
      NVMBuffer + bindingRecordOffset + 11*sizeof(BINDING_RECORD),
      NVMBuffer + bindingRecordOffset + 12*sizeof(BINDING_RECORD),
      NVMBuffer + bindingRecordOffset + 13*sizeof(BINDING_RECORD),
      NVMBuffer + bindingRecordOffset + 14*sizeof(BINDING_RECORD)
};
```

Cod. 5.8.9: Dichiarazione e inizializzazione della variabile di remapping dei record di binding.

validazione.

5.8.2 – Funzione GetBindingRecord

La funzione GetBindingRecord si occupa della lettura dei record di binding dalla memoria di programma del microcontrollore; originariamente è definita come visibile nel frammento di codice 5.8.10, tratto dall'elenco di Cod. 5.4.2.

```
#define GetBindingRecord(x, y)
NVMRead((BYTE *)x, (ROM void*)y, sizeof(BINDING_RECORD))
```

Cod. 5.8.10: Definizione originale della funzione GetBindingRecord.

Analogamente alla duale funzione di memorizzazione, accetta due parametri: il primo contiene il riferimento alla locazione in memoria di programma nella quale si trova il record da leggere; il secondo contiene il riferimento alla locazione in RAM nella quale deve essere trasferito il dato letto; ovvero il prototipo delle chiamate a GetBindingRecord è quello visibile nel frammento di codice 5.8.11, dove la variabile *currentBindingRecord* è di tipo BINDING_RECORD, come visto nel paragrafo 5.8.1.

```
GetBindingRecord(pCurrentBindingRecord, &currentBindingRecord);
```

Cod. 5.8.11: Prototipo delle chiamate a GetBindingRecord.

5.8.3 – Considerazioni

Nei precedenti paragrafi sono state presentate le funzioni di salvataggio e di lettura dei record di binding verso e dalla memoria di programma; in particolare si è visto il loro comportamento ed i prototipi alle relative chiamate. Nel paragrafo 5.8.1 inoltre, sono stati introdotti: offset, dimensioni e variabile di remapping per permettere la corretta gestione della memoria di programma, al fine di riservarne lo spazio necessario ai vari record. In questa sezione, di tali funzioni saranno presentati gli aggiornamenti delle definizioni.

Secondo quanto visto nel paragrafo 5.8.1, la funzione PutBindingRecord

può essere ridefinita come nel frammento di codice 5.8.12.

```
/** Definizione della funzione PutBindingRecord che salva un
 *  record di binding nella memoria FLASH.
 */
#define PutBindingRecord(x, y)
NVMWrite((BYTE*)x, (BYTE*)y, sizeof(BINDING_RECORD));
```

Cod. 5.8.12: Definizione aggiornata della funzione PutBindingRecord.

Rispetto alla definizione originale, riportata nel riquadro di codice 5.8.5, si è mutato solamente il tipo del cast del primo parametro, da NVM_ADDR a BYTE.

Secondo quanto riportato nel paragrafo 5.8.2, la funzione GetBindingRecord può essere ridefinita come nel frammento di codice 5.8.13.

```
/** Definizione della funzione GetBindingRecord che legge un
 *  record di binding dalla memoria FLASH.
 */
#define GetBindingRecord(x, y)
NVMRead((BYTE*)y, (BYTE*)x, sizeof(BYTE));
```

Cod. 5.8.13: Definizione aggiornata della funzione GetBindingRecord.

Rispetto alla definizione originale, riportata nel riquadro di codice 5.8.10, si è mutato l'ordine dei primi due argomenti (questo cambiamento è la conseguenza dell'inversione dei parametri nella funzione NVMRead, aggiornata nel paragrafo 5.3.3) e il tipo del cast del secondo parametro da void a BYTE. Tale cambiamento si effettua in quanto la funzione NVMRead ha la necessità di incrementare l'indirizzo del puntatore di un byte alla volta, al fine di aggiornare il buffer locale *temp* (si veda il paragrafo 5.3.3, in particolare il frammento di codice 5.3.5).

Le definizioni appena introdotte altro non sono che una mappatura delle funzioni NVMWrite ed NVMRead, e sono tutte inserite nel file NVM.h, insieme alle definizioni di offset, dimensioni e variabile di remapping.

5.8.4 – Codice di esempio

Al fine di testare la validità delle ipotesi e delle considerazioni trattate nei paragrafi 5.8.1, 5.8.2 e 5.8.3, si amplia il progetto creato appositamente (introdotto nel paragrafo 5.5.4) con i frammenti di codice presentati nei paragrafi sopracitati e con una parte di codice, inserita nella funzione main, che utilizza le funzioni di storage relative ai record di binding.

Quest'ultima è visibile nel frammento di codice 5.8.14, nel quale sono state omesse le parti relative alle informazioni trattate precedentemente e contiene: una parte di dichiarazione delle variabili; una parte di inizializzazione in RAM dei dati da scrivere nella memoria di programma; la scrittura dei record di binding; la lettura di un record campione dalla memoria FLASH e relativo trasferimento del valore in RAM.

Nel codice si dichiarano tre variabili: *bindingRecordTable,* un array allocato in RAM contenente tutti i record di binding da salvare in memoria di programma; *pCurrentBindingRecord,* un puntatore alla locazione nella memoria FLASH il quale sarà inizializzato ad un elemento della array di remapping prima di ogni scrittura; *currentBindingRecord*, di tipo BINDING_RECORD, utilizzata come cursore all'interno del ciclo di scrittura.

```
int main(){
    // DICHIARAZIONE DELLE VARIABILI
    BINDING_RECORD bindingRecordTable[MAX_BINDINGS];
    ROM BYTE * pCurrentBindingRecord; // Puntatore in FLASH.
    BINDING_RECORD currentBindingRecord; // Puntatore in RAM.

    // --- Inizializzazione dei valori in RAM:
    for(i=0; i<MAX_BINDINGS; i++){
        bindingRecordTable[i].shortAddr.Val = i;
        bindingRecordTable[i].endPoint = i + 15;
        bindingRecordTable[i].clusterID.Val = i + 16;
        bindingRecordTable[i].nextBindingRecord = 0xFF - i;
    }

    // --- Scrittura in memoria di programma.
    for(i=0; i<MAX_BINDINGS; i++){
        currentBindingRecord = bindingRecordTable[i];
        pCurrentBindingRecord = apsBindingTable[i];
        PutBindingRecord(pCurrentBindingRecord,
                         &currentBindingRecord);
    }

    // --- Lettura dalla memoria di programma.
    pCurrentBindingRecord = apsBindingTable[5];
    GetBindingRecord(&currentBindingRecord,
                     pCurrentBindingRecord);
```

Cod. 5.8.14: Parte integrante nella funzione main per testare le funzioni di storage dei record di binding.

Il primo dei due cicli *for* provvede all'inizializzazione dei record da salvare. L'inizializzazione è effettuata in modo tale che i campi dello i-esimo record siano impostati come segue: il campo *shortAddr* al valore di *i*; il campo *endPoint* al valore *i+15*; il campo *clusterID* al valore *i+16*; il campo *nextBindingRecord* al valore *0xFF-i*.

Il secondo dei cicli *for* provvede, ad ogni iterazione, a salvare un record di binding. Prima di ogni salvataggio l'iterazione aggiorna la variabile cursore *currentBindingRecord* e imposta la variabile puntatore *pCurrentBindingRecord* alla locazione di programma nella quale scrivere i dati che corrisponde all'i-esimo elemento dell'array di remapping.

Il salvataggio in memoria di programma comporta la scrittura delle locazioni della FLASH come visibile in Fig. 5.8.2: i valori sono memorizzati a partire dall'indirizzo esadecimale 0x45A, poiché questa è

la prima locazione libera dopo la memorizzazione della chiave di validazione per gli indirizzi a livello APS. Tale allineamento si riesce ad ottenere grazie alla gestione degli indirizzamenti tramite offset e dimensioni per ogni informazione da salvare in memoria. In figura sono evidenziati in giallo i valori relativi al primo record di binding: la prima parola di istruzione contiene il valore del campo *shortAddr*; la seconda contiene il valore del campo *endPoint*; la terza contiene il valore del campo *clusterID* e la quarta contiene il valore del campo *nextBindingRecord*. Come si può vedere dalla figura, i valori delle prime tre parole di istruzione coincidono con quelli dei campi corrispondenti; mentre il valore memorizzato nell'ultima parola di istruzione (del primo record) è 0x15FF, diversamente dal valore di *nextBindingRecord* il quale è 0xFF. Il motivo di questa discordanza è dovuto al fatto che *nextBindingRecord* è di tipo byte, pertanto compare in una parola di istruzione da destra verso sinistra, poiché, come spiegato nella sezione 5.3.2 ed in particolare in Fig. 5.3.3, una parola di istruzione occupa due indirizzi e di questi il più piccolo corrisponde alla parte meno significativa. I due byte più significativi della parola di istruzione, in questo caso contengono un valore che dipende dalla locazione in RAM di *currentBindingRecord*, in quanto una variabile di tipo BINDING_RECORD, come osservato nel paragrafo 5.8, controlla 6 byte, ma di fatto occupa uno spazio in memoria di 8 byte poiché l'architettura del microcontrollore è a 16 bit.

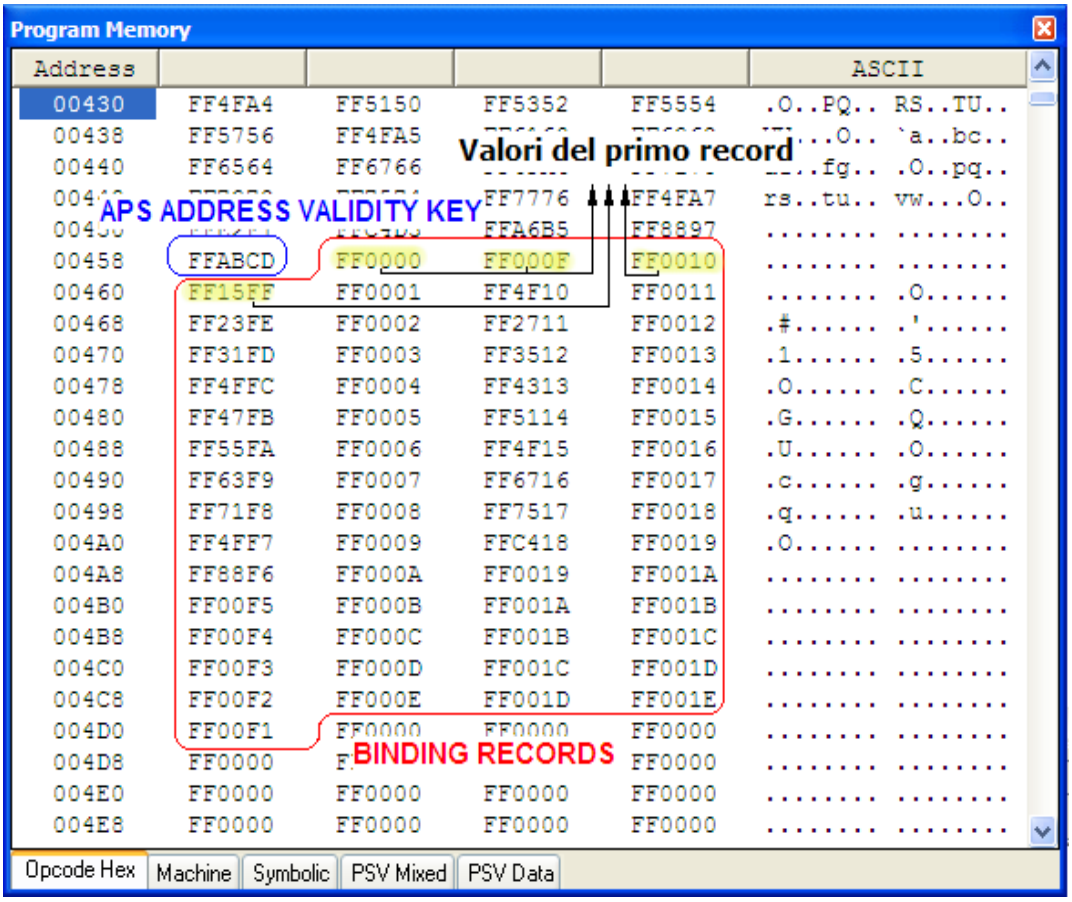

Fig. 5.8.2: Valori dei record di binding salvati nella memoria di programma.

La funzione di lettura ha il compito di leggere i dati dalla FLASH e di trasferirli in RAM. Per mezzo del simulatore MPLAB SIM integrato nell'ambiente di sviluppo, è possibile osservare l'aggiornamento della variabile *currentBindingRecord*, la quale a lettura ultimata del sesto (indice *i=5*) record di binding assume il valore visibile in Fig. 5.8.3, in accordo con i

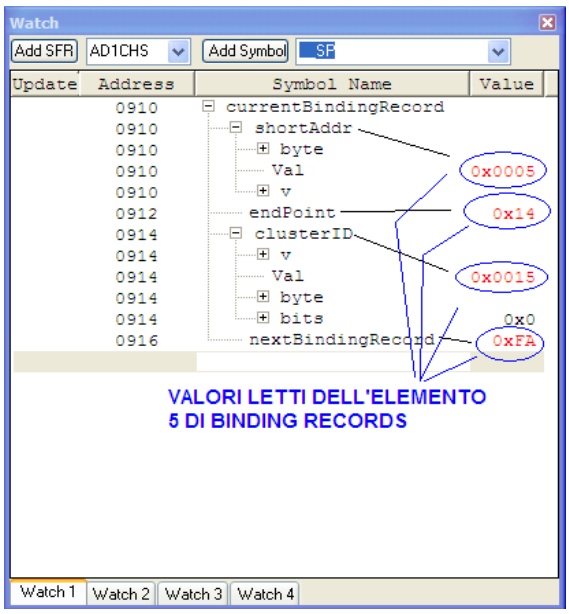

Fig. 5.8.3: Risultati prodotti dalla simulazione.

valori salvati, visibili in Fig. 5.8.2 e con quelli impostati nel codice 5.8.14.

5.9 – Esplorazione di Binding Source e Binding Usage

In seguito a quanto riportato nella sezione 5.4 – Utilizzo delle funzioni di storage, si procede alla ricerca delle funzioni PutBindingSourceMap(x, y) e GetBindingSourceMap(x, y) definite nel file zNVM.h alle righe 303 e 299 rispettivamente, e delle funzioni PutBindingUsageMap(x, y) e GetBindingUsageMap(x, y) definite nel file zNVM.h alle righe 304 e 300 rispettivamente.

5.9.1 – Funzione PutBindingSourceMap

La funzione in esame si occupa di memorizzare una sorgente dei binding all'interno della memoria di programma del microcontrollore. La sua definizione originale, tratta dall'elenco di Cod. 5.4.1 è visibile nel frammento di codice 5.9.1:

```
#define PutBindingSourceMap(x, y)
NVMWrite((NVM_ADDR *)&bindingTableSourceNodeMap[y>>3], (BYTE *)x,
          1)
```

Cod. 5.9.1: Definizione originale della funzione PutBindingSourceMap.

dove la variabile *bindingTableSourceNodeMap* dovrebbe essere il riferimento alla locazione nella memoria FLASH nella quale sarà memorizzata la chiave di validazione; in realtà nella sua dichiarazione originaria non è specificato l'attributo *space(prog)* e pertanto tale variabile è allocata in RAM.

La funzione accetta due argomenti: il riferimento alla locazione in RAM nella quale si trova il valore della sorgente di binding da memorizzare e un indice che specifica quale sorgente di binding si sta salvando. Pertanto il prototipo della sua chiamata è quello mostrato nel riquadro

di codice 5.9.2, dove le variabili *bindingMapSourceByte* e *bindingSrcIndex* sono entrambe di tipo BYTE e dichiarate a livello locale. La ridefinizione della funzione dovrà essere compatibile con tale prototipo, al fine di non dover effettuare alcuna modifica di massa all'interno dello stack.

```
PutBindingSourceMap(&bindingMapSourceByte, bindingSrcIndex);
```

Cod. 5.9.2: Prototipo della chiamata alla funzione PutBindingSourceMap.

Dalla definizione della costante *BINDING_USAGE_MAP_SIZE* (si vedano i frammenti di codice 5.8.8 e 5.9.3), che indica il numero delle sorgenti di binding da salvare in memoria, e dal tipo della variabile *bindingMapSourceByte* è possibile affermare che lo spazio in memoria di programma da riservare alla chiave di validazione è di 2 byte.

```
#if (MAX_BINDINGS % 8) == 0
    #define BINDING_USAGE_MAP_SIZE (MAX_BINDINGS/8)
#else
    #define BINDING_USAGE_MAP_SIZE (MAX_BINDINGS/8 + 1)
#endif
```

Cod. 5.9.3: Definizione della costante BINDING_USAGE_MAP_SIZE.

5.9.2 - Funzione GetBindingSourceMap

La funzione in esame si occupa di leggere una sorgente di binding dalla memoria di programma del microcontrollore e di trasferirne il valore in RAM. La sua definizione originale, tratta dall'elenco di Cod. 5.4.2 è visibile nel frammento di codice 5.9.4:

```
#define GetBindingSourceMap(x, y)
NVMRead((BYTE *)x, (ROM void*)&bindingTableSourceNodeMap[y>>3], 1)
```

Cod. 5.9.4: Definizione originale della funzione GetBindingSourceMap.

La funzione accetta due argomenti: il riferimento alla locazione in RAM nella quale trasferire il valore dalla memoria di programma e un indice che specifica quale sorgente di binding si vuole leggere, pertanto il prototipo della sua chiamata è quello mostrato nel riquadro di codice

5.9.5 ed è del tutto analogo a quello della funzione PutBindingSourceMap.

```
GetBindingSourceMap(&bindingMapSourceByte, bindingSrcIndex);
```

Cod. 5.9.5: Prototipo della chiamata alla funzione GetBindingSourceMap.

5.9.3 – Funzione PutBindingUsageMap

La funzione in esame si occupa di memorizzare un impiego dei binding all'interno della memoria di programma del microcontrollore. La sua definizione originale, tratta dall'elenco di Cod. 5.4.1 è visibile nel frammento di codice 5.9.6:

```
#define PutBindingUsageMap(x, y)
NVMWrite((NVM_ADDR *)&bindingTableUsageMap[y>>3], (BYTE *)x, 1 )
```

Cod. 5.9.6: Definizione originale della funzione PutBindingSourceMap.

dove la variabile *bindingTableUsageMap* dovrebbe essere il riferimento alla locazione nella memoria FLASH nella quale sarà memorizzata la chiave di validazione; in realtà nella sua dichiarazione originaria non è specificato l'attributo *space(prog)* e pertanto tale variabile è allocata in RAM.

La funzione accetta due argomenti: il riferimento alla locazione in RAM nella quale si trova il valore della sorgente di binding da memorizzare e un indice che specifica quale degli impieghi dei binding si sta salvando. Pertanto il prototipo della sua chiamata è quello mostrato nel riquadro di codice 5.9.7, dove le variabili *bindingMapUsageByte* e *bindingSrcIndex* sono entrambe di tipo BYTE e dichiarate a livello locale. La ridefinizione della funzione dovrà essere compatibile con tale prototipo, al fine di non dover effettuare alcuna modifica di massa

```
PutBindingUsageMap(&bindingMapUsageByte, bindingSrcIndex);
```

Cod. 5.9.7: Prototipo della chiamata alla funzione PutBindingUsageMap.

all'interno dello stack.

Analogamente alle sorgenti dei binding, lo spazio da riservare in memoria è di 2 byte, in quanto la variabile *bindingMapUsageByte* è di tipo byte, ed il numero totale degli impieghi è sempre definito dalla costante *BINDING_USAGE_MAP_SIZE*.

5.9.4 - Funzione GetBindingUsageMap

La funzione in esame si occupa di leggere un impiego dei binding dalla memoria di programma del microcontrollore e di trasferirne il valore in RAM. La sua definizione originale, tratta dall'elenco di Cod. 5.4.2 è visibile nel frammento di codice 5.9.8:

```
#define GetBindingUsageMap(x, y)
NVMRead((BYTE *)x, (ROM void*)&bindingTableUsageMap[y>>3], 1)
```

Cod. 5.9.8: Definizione originale della funzione GetBindingUsageMap.

La funzione accetta due argomenti: il riferimento alla locazione in RAM nella quale trasferire il valore dalla memoria di programma e un indice che specifica quale sorgente di binding si vuole leggere, pertanto il prototipo della sua chiamata è quello mostrato nel riquadro di codice 5.9.9 ed è del tutto analogo a quello della funzione PutBindingUsageMap.

```
GetBindingUsageMap(&bindingMapUsageByte, bindingSrcIndex);
```

Cod. 5.9.9: Prototipo della chiamata alla funzione GetBindingUsageMap.

5.9.5 - Considerazioni

Nei precedenti paragrafi sono state presentate le funzioni di salvataggio e di lettura di una sorgente e di un impiego dei binding, verso e dalla memoria di programma; in particolare si è visto il loro comportamento ed i prototipi alle relative chiamate. In questa sezione ci si occuperà

delle dichiarazioni dei parametri necessari alla corretta gestione della memoria di programma al fine di riservarne lo spazio necessario alla chiave di validazione e saranno presentati gli aggiornamenti delle definizioni delle funzioni PutBindingSourceMap, GetBindingSourceMap, PutBindingUsageMap, GetBindingUsageMap.

Osservando la Fig. 5.8.2 si nota che la prima locazione libera in memoria di programma è all'indirizzo 0x4D2. Per garantire la scrittura della sorgente di binding in tale locazione è necessario dichiararne l'offset tenendo conto di quello dei record di binding e della relativa dimensione; per allineare correttamente il salvataggio dei valori degli impieghi dei binding, è necessario dichiararne l'offset tenendo conto di quello delle sorgenti dei binding e delle relative dimensioni.

È possibile procedere alla definizione delle dimensioni sia della sorgente che degli impieghi dei binding, in quanto, come osservato nei paragrafi 5.9.1 e 5.9.3, lo spazio in memoria richiesto è di 2 byte per entrambe le informazioni.

Nel file NVM.h è necessario inoltre, definire la variabile di remapping,

```
/** bindingSourceMapOffset indica quanti byte intercorrono tra la
 *  locazione iniziale di NVMBuffer e la locazione dove ha inizio
 *  la memorizzazione dei Binding Source Map.
 */
#define bindingSourceMapOffset      bindingRecordOffset +
                                    bindingRecordMapSize

/** Dimensione in BYTE dello spazio da allocare in memoria di
 *  programma riservato ai Binding Source Map.
 */
#define bindingSourceMapSize BINDING_USAGE_MAP_SIZE * sizeof(BYTE)

/** Definizione dell'array bindingTableSourceNodeMap. L'elemento
 *  i-esimo di questo array contiene un puntatore alla locazione
 *  in memoria di programma nella quale deve essere salvato
 *  l'i-esimo binding source.
 */
extern ROM BYTE *
            bindingTableSourceNodeMap[BINDING_USAGE_MAP_SIZE];
```

Cod. 5.9.10: Definizione dei parametri per la memorizzazione delle sorgenti dei binding.

attraverso la parola chiave *extern*, al fine di poterne fare riferimento all'interno del file Main.c.

```
/** bindingUsageMapOffset indica quanti byte intercorrono tra la
 *  locazione iniziale di NVMBuffer e la locazione dove ha inizio
 *  la memorizzazione dei Binding Usage.
 */
#define bindingUsageMapOffset       bindingSourceMapOffset +
                                    bindingSourceMapSize

/** Dimensione in BYTE dello spazio da allocare in memoria di
 *  programma riservato ai Binding Usage.
 */
#define bindingUsageMapSize BINDING_USAGE_MAP_SIZE * sizeof(WORD)

/** Definizione dell'array bindingTableUsageMap. L'elemento
 *  i-esimo di questo array contiene un puntatore alla locazione
 *  in memoria di programma nella quale deve essere salvato
 *  l'i-esimo binding usage.
 */
extern ROM BYTE * bindingTableUsageMap[BINDING_USAGE_MAP_SIZE];
```

Cod. 5.9.11: Definizione dei parametri per la memorizzazione degli impieghi dei binding.

Tali definizioni sono visibili nel frammento di codice 5.9.10 per le sorgenti dei binding e nel frammento di codice 5.9.11 per gli impieghi.

Le dichiarazioni delle variabili di remapping, avvengono nel file NVM.c e sono inizializzate alle opportune locazioni di memoria di programma grazie agli offset della sorgente e degli impieghi dei binding, come visibile nel frammento di codice 5.9.12.

```
ROM BYTE * bindingTableSourceNodeMap[BINDING_USAGE_MAP_SIZE] = {
    NVMBuffer + bindingSourceMapOffset + 0*sizeof(WORD),
    NVMBuffer + bindingSourceMapOffset + 1*sizeof(WORD)
};

ROM BYTE * bindingTableUsageMap[BINDING_USAGE_MAP_SIZE] = {
    NVMBuffer + bindingUsageMapOffset + 0*sizeof(WORD),
    NVMBuffer + bindingUsageMapOffset + 1*sizeof(WORD)
};
```

Cod. 5.9.12: Dichiarazione e inizializzazione delle variabili di remapping.

Una volta completate le definizioni dei parametri per il salvataggio delle sorgenti e degli impieghi dei binding nella memoria di programma è

possibile procedere alla ridefinizione delle relative funzioni di lettura e scrittura.

Secondo quanto visto nel paragrafo 5.9.1, la funzione PutBindingSourceMap può essere ridefinita come nel frammento di codice 5.9.13.

```
#define PutBindingSourceMap(x, y)
NVMWrite((BYTE*)bindingTableSourceNodeMap[y>>3], (BYTE*)x,
          sizeof(BYTE))
```

Cod. 5.9.13: Definizione aggiornata della funzione PutBindingSourceMap.

Rispetto alla definizione originale, riportata nel riquadro di codice 5.9.1, si è omesso il simbolo "&" avanti al primo parametro. Questa modifica si rende necessaria in quanto la variabile di remapping è stata dichiarata come puntatore.

Secondo quanto riportato nel paragrafo 5.9.2, la funzione GetBindingSourceMap può essere ridefinita come nel frammento di codice 5.9.14.

```
#define GetBindingSourceMap(x, y)
NVMRead((BYTE*)bindingTableSourceNodeMap[y>>3], (BYTE*)x,
sizeof(BYTE))
```

Cod. 5.9.14: Definizione aggiornata della funzione GetBindingSourceMap.

Rispetto alla definizione originale, riportata nel riquadro di codice 5.9.4 si è variato l'ordine dei primi due parametri (in conseguenza all'inversione dei parametri nella funzione NVMRead aggiornata nel paragrafo 5.3.3).

Secondo quanto visto nel paragrafo 5.9.3, la funzione PutBindingUsageMap può essere ridefinita come nel frammento di codice 5.9.15.

```
#define PutBindingUsageMap(x, y)
NVMWrite((BYTE*)bindingTableUsageMap[y>>3], (BYTE*)x,
sizeof(BYTE))
```

Cod. 5.9.15: Definizione aggiornata della funzione PutBindingUsageMap.

Rispetto alla definizione originale, riportata nel riquadro di codice 5.9.6, si è omesso il simbolo "&" avanti al primo parametro. Questa modifica si rende necessaria in quanto la variabile di remapping è stata dichiarata come puntatore.

Secondo quanto riportato nel paragrafo 5.9.4, la funzione GetBindingUsageMap può essere ridefinita come nel frammento di codice 5.9.16.

```
#define GetBindingUsageMap(x, y)
NVMRead((BYTE*)bindingTableUsageMap[y>>3], (BYTE*)x, sizeof(BYTE))
```

Cod. 5.9.16: Definizione aggiornata della funzione GetBindingUsageMap.

Rispetto alla definizione originale, riportata nel riquadro di codice 5.9.6 si è variato l'ordine dei primi due parametri (in conseguenza all'inversione dei parametri nella funzione NVMRead aggiornata nel paragrafo 5.3.3).

Gli aggiornamenti effettuati non comportano alcuna modifica al prototipo delle chiamate, pertanto nel codice dello stack non è necessario effettuare alcuna ulteriore modifica.

5.9.6 – Codice di esempio

Al fine di testare la validità delle ipotesi e delle considerazioni trattate nei paragrafi 5.9.1, 5.9.2, 5.9.3, 5.9.4 e 5.9.5, si amplia il progetto creato appositamente (introdotto nel paragrafo 5.5.4) con i frammenti di codice presentati nei paragrafi sopracitati e con una parte di codice, inserita nella funzione main, che utilizza le funzioni di storage relative alle sorgenti e agli impieghi dei binding.

Tale codice è visibile nel frammento 5.9.17, nel quale sono state omesse le parti relative alle informazioni trattate precedentemente e contiene: una parte di dichiarazione delle variabili; la scrittura di due sorgenti di binding in memoria di programma; la lettura di queste dalla memoria FLASH.

```
int main(){
    // DICHIARAZIONE DELLE VARIABILI
    BYTE temp_byte = 0xA3;

    // SCRITTURE NELLA MEMORIA DI PROGRAMMA.
    // --- Binding Source.
    PutBindingSourceMap(&temp_byte, 0);
    temp_byte = 0x4A;
    PutBindingSourceMap(&temp_byte, 8);
    // --- Binding Usage.
    temp_byte = 0xBF;
    PutBindingUsageMap(&temp_byte, 0);
    temp_byte = 0xCA;
    PutBindingUsageMap(&temp_byte, 8);

    // LETTURE DALLA MEMORIA DI PROGRAMMA.
    // --- Binding Source.
    GetBindingSourceMap(&temp_byte, 0);
    GetBindingSourceMap(&temp_byte, 8);
    // --- Binding Usage.
    GetBindingUsageMap(&temp_byte, 0);
    GetBindingUsageMap(&temp_byte, 8);

    return 0;
}
```

Cod. 5.9.17: Parte integrante nella funzione main per testare le funzioni di storage delle sorgenti e degli impieghi dei binding.

In tale codice si dichiara una sola variabile: *temp_byte* di tipo BYTE, la quale è inizializzata ai valori da memorizzare prima di ogni scrittura.

Il salvataggio in memoria di programma comporta la scrittura delle

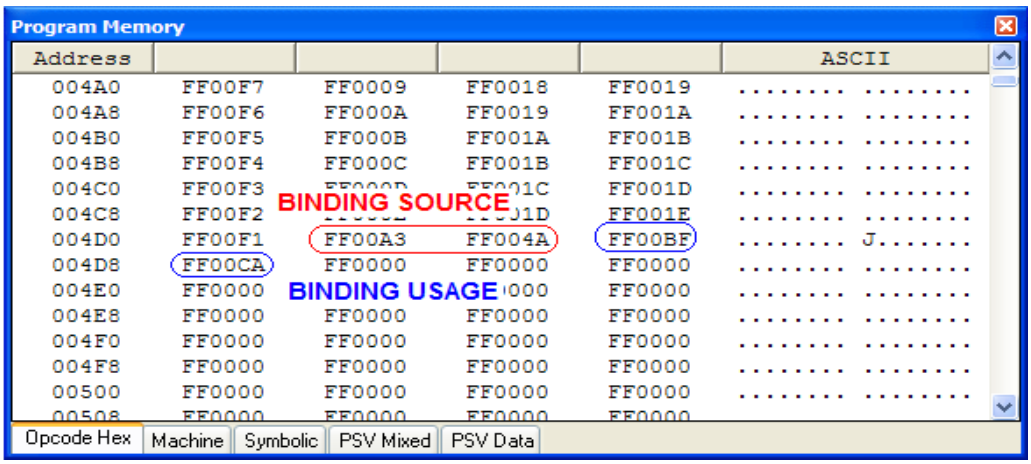

Fig. 5.9.1: Valore del MAC ADDRESS salvato nella memoria di programma.

locazioni della FLASH come visibile in Fig. 5.9.1: i valori sono memorizzati a partire dall'indirizzo esadecimale 0x4D2 per le sorgenti dei binding, poiché questa è la prima locazione libera dopo la memorizzazione dei record di binding e dall'indirizzo esadecimale 0x4D6 per gli impieghi. Tali allineamenti, come osservato nel paragrafo 5.9.5, sono ottenuti grazie alla gestione degli indirizzamenti tramite offset e dimensioni relativi ad ogni informazione da salvare in memoria. Confrontando i valori della figura con quelli della variabile *temp_byte* nel codice 5.9.17, si può affermare che le scritture avvengono correttamente per entrambe le informazioni.

Le funzioni di lettura hanno il compito di leggere i dati dalla FLASH e di trasferirli in RAM. Per mezzo del simulatore MPLAB SIM, integrato nell'ambiente di sviluppo, è possibile osservare l'aggiornamento della variabile *temp_byte*, la quale ultimata la lettura della seconda sorgente assume il valore visibile in Fig. 5.9.2, ultimata la lettura del secondo impiego assume il valore visibile in Fig. 5.9.3, entrambi in accordo con i valori salvati, visibili in Fig. 5.9.1.

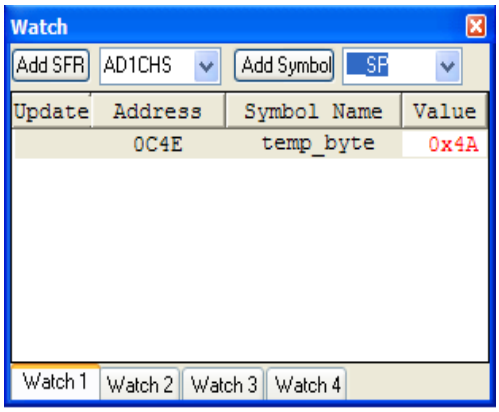

Fig. 5.9.2: Risultati del simulatore dopo la lettura della seconda sorgente di binding.

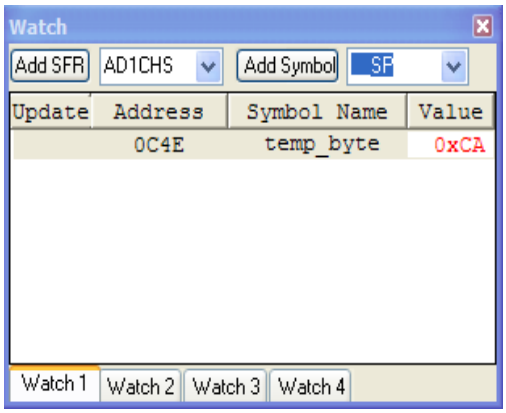

Fig. 5.9.3: Risultati del simulatore dopo la lettura degli impieghi dei binding.

5.10 – Esplorazione di Binding Validity Key

In seguito a quanto riportato nella sezione 5.4 – Utilizzo delle funzioni di storage, si procede alla ricerca delle funzioni PutBindingValidityKey(x) e GetBindingValidityKey(x) definite nel file zNVM.h alle righe 305 e 301 rispettivamente.

5.10.1 – Funzione PutBindingValidityKey

La funzione in esame si occupa di memorizzare la chiave di validazione per i record di binding all'interno della memoria di programma del microcontrollore. La sua definizione originale, tratta dall'elenco di Cod. 5.4.1 è visibile nel frammento di codice 5.10.1:

```
#define PutBindingValidityKey(x)
NVMWrite((NVM_ADDR *)&bindingValidityKey, (BYTE *)x, sizeof(WORD))
```

Cod. 5.10.1: Definizione originale della funzione PutBindingValidityKey.

dove la variabile *bindingValidityKey* dovrebbe essere il riferimento alla locazione nella memoria FLASH nella quale sarà memorizzata la chiave di validazione; in realtà nella sua dichiarazione originaria non è specificato l'attributo *space(prog)* e pertanto tale variabile è allocata in RAM.

La funzione accetta un solo argomento: il riferimento alla locazione in RAM nella quale si trova il valore della chiave di validazione da memorizzare. Pertanto il prototipo della sua chiamata è quello mostrato nel riquadro di codice 5.10.2, dove la variabile *key* è una WORD dichiarata a livello locale. La ridefinizione della funzione dovrà essere compatibile con tale prototipo, al fine di non dover effettuare alcuna modifica di massa all'interno dello stack.

```
PutBindingValidityKey(&key);
```

Cod. 5.10.2: Prototipo della chiamata alla funzione PutBindingValidityKey

Dalla definizione di Cod. 5.10.1 e dal tipo della variabile *key* è possibile affermare che lo spazio in memoria di programma da riservare alla chiave di validazione è di 2 byte.

5.10.2 – Funzione GetBindingValidityKey

La funzione in esame si occupa di leggere la chiave di validazione per i record di binding dalla memoria di programma del microcontrollore e di trasferirne il valore in RAM. La sua definizione originale, tratta dall'elenco di Cod. 5.4.2 è visibile nel frammento di codice 5.10.3:

```
#define GetBindingValidityKey(x)
NVMRead((BYTE *)x, (ROM void*)&bindingValidityKey, sizeof(WORD))
```

Cod. 5.10.3: Definizione originale della funzione GetBindingValidityKey.

La funzione accetta un solo argomento: il riferimento alla locazione in RAM nella quale trasferire il valore della chiave di validazione dalla memoria di programma, pertanto il prototipo della sua chiamata è quello mostrato nel riquadro di codice 5.10.4 ed è del tutto analogo a quello della funzione PutBindingValidityKey.

```
GetBindingValidityKey(&key);
```

Cod. 5.10.4: Prototipo della chiamata alla funzione GetBindingValidityKey.

5.10.3 – Considerazioni

Nei precedenti paragrafi sono state presentate le funzioni di salvataggio e di lettura della chiave di validazione dei record di binding, verso e dalla memoria di programma; in particolare si è visto il loro comportamento ed i prototipi alle relative chiamate. In questa sezione ci si occuperà delle dichiarazioni dei parametri necessari alla corretta gestione della memoria di programma al fine di riservarne lo spazio necessario alla chiave di validazione e saranno presentati gli aggiornamenti delle definizioni delle funzioni PutBindingValidityKey e

GetBindingValidityKey.

Osservando la Fig. 5.9.1 si nota che la prima locazione libera in memoria di programma è all'indirizzo 0x4DA. Per garantire la scrittura della chiave di validazione in tale locazione è necessario dichiararne l'offset tenendo conto di quello degli impieghi dei binding e delle relative dimensioni. Inoltre è possibile procedere alla definizione della dimensione della chiave di validazione, in quanto, come osservato nel paragrafo 5.10.1, lo spazio di memoria richiesto è di 2 byte. Nel file NVM.h è necessario definire la variabile di remapping, attraverso la parola chiave *extern*, al fine di poterne fare riferimento all'interno del file Main.c. Tali definizioni sono visibili nel frammento di codice 5.10.5.

```
/** bindingValidityKeyOffset indica quanti byte intercorrono tra
 *  la locazione iniziale di NVMBuffer e la locazione dove ha
 *  inizio la memorizzazione di bindingValidityKey.
 */
#define bindingValidityKeyOffset    bindingUsageMapOffset +
                                    bindingUsageMapSize

/** Dimensione in BYTE dello spazio da allocare in memoria di
 *  programma riservato a BINDING VALIDITY KEY.
 */
#define bindingValidityKeySize sizeof(WORD)

/** Definizione della variabile bindingValidityKey. La variabile
 *  contiene è un puntatore alla locazione in memoria di programma
 *  nella quale deve essere salvata la BINDING VALIDITY KEY.
 */
extern ROM BYTE * bindingValidityKey;
```

Cod. 5.10.5: Definizione dei parametri per la memorizzazione della chiave nella memoria di programma.

La dichiarazione della variabile di remapping, avviene nel file NVM.c ed è inizializzata alle opportune locazioni di memoria di programma grazie all'offset della chiave di validazione, come visibile nel frammento di

```
ROM BYTE * bindingValidityKey =     NVMBuffer +
                                    bindingValidityKeyOffset;
```

Cod. 5.10.6: Dichiarazione e inizializzazione della variabile di remapping della chiave di validazione.

codice 5.10.6.

Una volta completate le definizioni dei parametri per il salvataggio della chiave nella memoria di programma è possibile procedere alla ridefinizione delle funzioni di lettura e scrittura.

Secondo quanto visto nel paragrafo 5.10.1, la funzione PutBindingValidityKey può essere ridefinita come nel frammento di codice 5.10.7.

```
#define PutBindingValidityKey(x)
NVMWrite((BYTE*)bindingValidityKey, (BYTE*)x, sizeof(WORD))
```

Cod. 5.10.7: Definizione aggiornata della funzione PutBindingValidityKey.

Rispetto alla definizione originale, riportata nel riquadro di codice 5.10.1, si è omesso il simbolo "&" avanti al primo parametro. Questa modifica si rende necessaria in quanto la variabile di remapping è stata dichiarata come puntatore.

Secondo quanto riportato nel paragrafo 5.10.2, la funzione GetBindingValidityKey può essere ridefinita come nel frammento di codice 5.10.8.

```
#define GetBindingValidityKey(x)
NVMRead((BYTE*)bindingValidityKey, (BYTE*)x, sizeof(WORD))
```

Cod. 5.10.8: Definizione aggiornata della funzione GetBindingValidityKey.

Rispetto alla definizione originale, riportata nel riquadro di codice 5.10.3 si è variato il tipo del cast del secondo parametro da void a BYTE.

Gli aggiornamenti non comportano alcuna modifica al prototipo delle chiamate, pertanto nel codice dello stack non è necessario effettuare alcuna ulteriore modifica.

5.10.4 – Codice di esempio

Al fine di testare la validità delle ipotesi e delle considerazioni trattate

nei paragrafi 5.10.1, 5.10.2 e 5.10.3, si amplia il progetto creato appositamente (introdotto nel paragrafo 5.5.4) con i frammenti di codice presentati nei paragrafi sopracitati e con una parte di codice, inserita nella funzione main, che utilizza le funzioni di storage relative alla chiave di validazione.

```
int main(){
    // DICHIARAZIONE DELLE VARIABILI
    WORD temp_word;

    // --- Scrittura in memoria di programma.
    temp_word = 0xEFA0;
    PutBindingValidityKey(&temp_word);

    // --- Lettura dalla memoria di programma.
    temp_word = 0;
    GetBindingValidityKey(&temp_word);

    return 0;
}
```

Cod. 5.10.9: Parte integrante nella funzione main per testare le funzioni di storage della chiave di validazione.

Tale codice è visibile nel frammento 5.10.9, nel quale sono state omesse le parti relative alle informazioni trattate precedentemente e

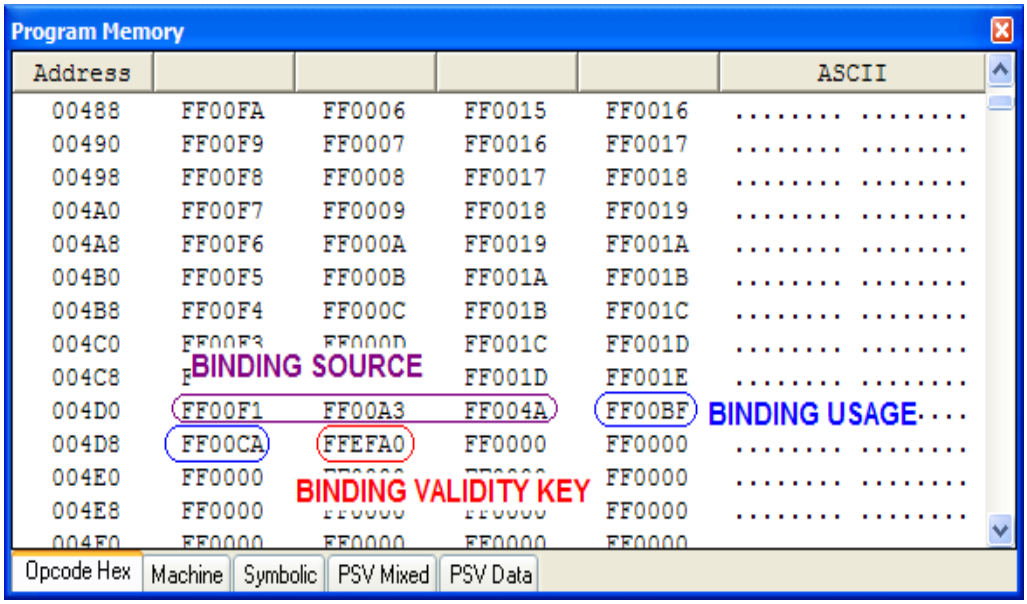

Fig. 5.10.1: Valore del MAC ADDRESS salvato nella memoria di programma.

contiene: una parte di dichiarazione delle variabili; la scrittura della chiave di validazione in memoria di programma; la lettura di questa dalla memoria FLASH.

In tale codice si dichiara una sola variabile: *temp_word* di tipo WORD, inizializzandola al valore della chiave di validazione da salvare.

Il salvataggio in memoria di programma comporta la scrittura delle locazioni della FLASH come visibile in Fig. 5.10.1: i valori sono memorizzati a partire dall'indirizzo esadecimale 0x4DA, poiché questa è la prima locazione libera dopo la memorizzazione degli impieghi dei binding. Tale allineamento, come osservato nel paragrafo 5.10.3, è ottenuto grazie alla gestione degli indirizzamenti tramite offset e dimensioni relativi ad ogni informazione da salvare in memoria. Confrontando i valori della figura con quelli della variabile *temp_word* del codice 5.10.9, si può affermare che la scrittura avviene correttamente.

La funzione di lettura ha il compito di leggere i dati dalla FLASH e di trasferirli in RAM. Per mezzo del simulatore MPLAB SIM, integrato nell'ambiente di sviluppo, è possibile osservare l'aggiornamento della variabile *temp_word*, la quale a lettura ultimata assume il valore visibile in Fig. 5.10.2, in accordo con il valore salvato, visibile in Fig. 5.10.1 e con quello impostato nel codice 5.10.9.

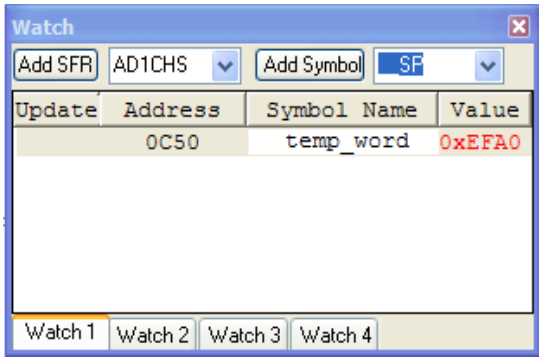

Fig. 5.10.2: Risultati prodotti dalla simulazione.

5.11 – Esplorazione dei Neighbor Record

In seguito a quanto riportato nella sezione 5.4 – Utilizzo delle funzioni di storage, si procede alla ricerca delle funzioni PutNeighborRecord(x, y) e GetNeighborRecord(x, y) definite nel file zNVM.h alle righe 311 e 307 rispettivamente.

Al fine di apprendere il funzionamento delle funzioni di storage, risulta utile esaminare alcune definizioni di tipo di dato non ancora trattate nei precedenti paragrafi, iniziando da quelli più semplici e procedendo verso quelli più complessi.

Il tipo di dato PAN_ADDR, definito come nel frammento di codice 5.11.1 rappresenta un indirizzo di un dispositivo PAN (Personal Area Network). La sua struttura è identica al tipo di dato SHORT_ADDR, la cui definizione è stata esaminata nel frammento di codice 5.5.5.

```
typedef SHORT_ADDR PAN_ADDR;
```

Cod. 5.11.1: Definizione del tipo PAN_ADDR.

Il tipo di dato NEIGHBOR_RECORD_DEVICE_INFO, definito nel frammento di codice 5.11.2, consiste in una struttura utile a contenere

```
typedef union _NEIGHBOR_RECORD_DEVICE_INFO{
    struct{
        BYTE LQI                : 8;
        BYTE Depth              : 4;
        BYTE StackProfile       : 4;
        BYTE ZigBeeVersion      : 4;
        BYTE deviceType         : 2;
        BYTE Relationship       : 2;
        BYTE RxOnWhenIdle       : 1;
        BYTE bInUse             : 1;
        BYTE PermitJoining      : 1;
        BYTE PotentialParent    : 1;
    } bits;
    DWORD Val;
} NEIGHBOR_RECORD_DEVICE_INFO;
```

Cod. 5.11.2: Definizione del tipo di dato NEIGHBOR_RECORD_DEVICE_INFO.

tutte le informazioni relative ad un dispositivo di un elemento della tabella dei vicini. Tale struttura occupa uno spazio in memoria di due WORD (4 byte), anche se di questi sono effettivamente utili solo 28 bit. La definizione fa uso della parola chiave *union* per permettere l'accesso in due modalità diverse: all'intero dato attraverso il campo *Val*, oppure alle singole informazioni attraverso i corrispondenti campi della struttura interna *bits*.

Il tipo di dato NEIGHBOR_RECORD, definito nel frammento di codice 5.11.3 rappresenta un elemento della tabella dei vicini e costituisce nell'insieme un dato di 18 byte qualora sia definita la macro di configurazione *I_SUPPORT_SECURITY*, di 17 byte altrimenti.

```
typedef struct _NEIGHBOR_RECORD{
    LONG_ADDR                       longAddr;
    SHORT_ADDR                      shortAddr;
    PAN_ADDR                        panID;
    NEIGHBOR_RECORD_DEVICE_INFO     deviceInfo;
    BYTE                            LogicalChannel;
#ifdef I_SUPPORT_SECURITY
    BOOL                            bSecured;
#endif
} NEIGHBOR_RECORD;
```

Cod. 5.11.3: Definizione del tipo NEIGHBOR_RECORD.

Si osservi che la definizione del tipo di dato booleano non è prevista dal linguaggio C standard, in questo contesto tale tipo di dato è definito come un char senza segno.

5.11.1 – Funzione PutNeighborRecord

La funzione in esame si occupa di memorizzare un record della tabella dei vicini all'interno della memoria di programma del microcontrollore.

```
#define PutNeighborRecord(x, y)
NVMWrite((NVM_ADDR *)x, (BYTE *)y, sizeof(NEIGHBOR_RECORD))
```

Cod. 5.11.4: Definizione originale della funzione PutBindingValidityKey.

La sua definizione originale, tratta dall'elenco di Cod. 5.4.1 è visibile

nel frammento di codice 5.11.4.

La funzione accetta due argomenti: il riferimento alla memoria di programma nella quale salvare il record e il riferimento alla locazione in RAM nella quale si trova il record da memorizzare, pertanto il prototipo della sua chiamata è quello mostrato nel riquadro di codice 5.11.5, dove la variabile *currentNeighborRecord* è di tipo NEIGHBOR_RECORD ed è dichiarata a livello locale; l'array *neighborTable*, ad elementi dello stesso tipo, è definito a livello globale ed ognuno dei suoi elementi dovrebbe puntare ad una locazione in memoria di programma nella quale salvare un record, in realtà nella dichiarazione originaria non è specificato l'attributo *space(prog)* e pertanto tutti gli elementi di tale array sono allocati in RAM.

```
PutNeighborRecord(&neighborTable[i], &currentNeighborRecord);
```

Cod. 5.11.5: Prototipo della chiamata alla funzione PutNeighborRecord.

La ridefinizione della funzione dovrà essere compatibile con tale prototipo, al fine di non dover effettuare alcuna modifica di massa all'interno dello stack.

5.11.2 – Funzione GetNeighborRecord

La funzione in esame si occupa di leggere un elemento della tabella dei vicini dalla memoria di programma del microcontrollore e di trasferirne il valore in RAM. La sua definizione originale, tratta dall'elenco di Cod. 5.4.2 è visibile nel frammento di codice 5.11.6:

```
#define GetNeighborRecord(x, y)
NVMRead((BYTE *)x, (ROM void*)y, sizeof(NEIGHBOR_RECORD))
```

Cod. 5.11.6: Definizione originale della funzione GetNeighborRecord.

La funzione accetta due argomenti: la locazione in memoria di programma dalla quale leggere i valori del record e il riferimento alla locazione in RAM nella quale trasferirne il valore, pertanto il prototipo

della sua chiamata è quello mostrato nel riquadro di codice 5.11.7, dove *pCurrentNeighborRecord* è inizializzato ad un elemento dell'array *neighborTable*.

```
GetNeighborRecord(&currentNeighborRecord, pCurrentNeighborRecord);
```

Cod. 5.11.7: Prototipo della chiamata alla funzione GetBindingValidityKey.

5.11.3 – Considerazioni

Nei precedenti paragrafi sono state presentate le funzioni di salvataggio e di lettura di un record della tabella dei vicini, verso e dalla memoria di programma; in particolare si è visto il loro comportamento ed i prototipi alle relative chiamate. In questa sezione ci si occuperà delle dichiarazioni dei parametri necessari alla corretta gestione della memoria di programma al fine di riservarne lo spazio necessario a contenere tutti i record della tabella dei vicini e saranno presentati gli aggiornamenti delle definizioni delle funzioni PutNeighborRecord e GetNeighborRecord.

Osservando la Fig. 5.10.1 si nota che la prima locazione libera in memoria di programma dopo le memorizzazioni degli impieghi dei binding è all'indirizzo 0x4DC. Per garantire la scrittura dei record a partire da tale locazione è necessario dichiararne l'offset tenendo conto di quello della chiave di validazione dei binding e delle relative dimensioni. Inoltre è possibile procedere alla definizione della dimensione della tabella dei vicini, in quanto lo spazio di memoria richiesto si ottiene moltiplicando la dimensione di un singolo record per il numero dei record, dato dalla costante *MAX_NEIGHBORS*, definita nel Cod. 5.11.8. Nel file NVM.h è necessario definire la variabile di remapping, attraverso la parola chiave *extern*, al fine di poterne fare riferimento all'interno del file Main.c. Tali definizioni sono visibili nel frammento di codice 5.11.8.

```
/** Numero di Neighbor Record da salvare in memoria di programma.
*/
#define MAX_NEIGHBORS 25

/** neighborRecordOffset indica quanti byte intercorrono tra la
locazione
 *  iniziale di NVMBuffer e la locazione dove ha inizio la
memorizzazione
 *  dei neighbor record.
 */
#define neighborRecordOffset  bindingValidityKeyOffset +
                              bindingValidityKeySize

/** Dimensione in BYTE dello spazio da allocare in memoria di
programma
 *  riservato ai neighbor record.
 */
#define neighborRecordMapSize sizeof(NEIGHBOR_RECORD) *
MAX_NEIGHBORS

/** Definizione dell'array neighborTable. L'elemento i-esimo di
questo array
 *  contiene un puntatore alla locazione in memoria di programma
nella quale
 *  deve essere salvato l'i-esimo neighbor record.
 */
extern ROM BYTE * neighborTable[MAX_NEIGHBORS];
```

Cod. 5.11.8: Definizione dei parametri per la memorizzazione della tabella nella memoria di programma.

La dichiarazione della variabile di remapping, avviene nel file NVM.c ed è inizializzata alle opportune locazioni di memoria di programma grazie all'offset della tabella dei vicini, come visibile nel frammento di codice 5.11.9.

```
ROM BYTE * neighborTable[MAX_NEIGHBORS]{
    NVMBuffer + neighborRecordOffset + 0*sizeof(NEIGHBOR_RECORD),
    NVMBuffer + neighborRecordOffset + 1*sizeof(NEIGHBOR_RECORD),
    // ...
    NVMBuffer + neighborRecordOffset + 23*sizeof(NEIGHBOR_RECORD),
    NVMBuffer + neighborRecordOffset + 24*sizeof(NEIGHBOR_RECORD)
};
```

Cod. 5.11.9: Dichiarazione e inizializzazione della variabile di remapping della tabella dei vicini.

Una volta completate le definizioni dei parametri per il salvataggio della

tabella nella memoria di programma è possibile procedere alla ridefinizione delle funzioni di lettura e scrittura.

Secondo quanto visto nel paragrafo 5.11.1, la funzione PutNeighborRecord può essere ridefinita come nel frammento di codice 5.11.10.

```
#define PutNeighborRecord(x, y)
NVMWrite((BYTE*)x, (BYTE*)y, sizeof(NEIGHBOR_RECORD))
```

Cod. 5.11.10: Definizione aggiornata della funzione PutNeighborRecord.

Rispetto alla definizione originale, riportata nel riquadro di codice 5.11.4, non è stata apportata alcuna modifica.

Secondo quanto riportato nel paragrafo 5.11.2, la funzione GetNeighborRecord può essere ridefinita come nel frammento di codice 5.11.11.

```
#define GetNeighborRecord(x, y)
NVMRead((BYTE*)y, (BYTE*)x, sizeof(NEIGHBOR_RECORD))
```

Cod. 5.11.11: Definizione aggiornata della funzione GetNeighborRecord.

Rispetto alla definizione originale, riportata nel riquadro di codice 5.11.6 si è variato il tipo del cast del secondo parametro da void a BYTE.

5.11.4 – Codice di esempio

Al fine di testare la validità delle ipotesi e delle considerazioni trattate nei paragrafi 5.11.1, 5.11.2 e 5.11.3, si amplia il progetto creato appositamente (introdotto nel paragrafo 5.5.4) con i frammenti di codice presentati nei paragrafi sopracitati e con una parte di codice, inserita nella funzione main, che utilizza le funzioni di storage relative ai record della tabella dei vicini.

Tale codice è visibile nel frammento 5.11.12, nel quale sono state omesse le parti relative alle informazioni trattate precedentemente e contiene: una parte di dichiarazione delle variabili; una parte di inizializzazione dei dati in RAM; la scrittura dei record in memoria di

programma; la lettura di questi dalla memoria FLASH.

```
int main(){
    // DICHIARAZIONE DELLE VARIABILI
    NEIGHBOR_RECORD neighborRecordTable[MAX_NEIGHBORS];
    ROM BYTE * pCurrentNeighborRecord;
    NEIGHBOR_RECORD currentNeighborRecord;

    // --- Inizializzazione:
    for(i=0; i<MAX_NEIGHBORS; i++){
        neighborRecordTable[i].longAddr.v[0] = 0x00 + (i<<4);
        neighborRecordTable[i].longAddr.v[1] = 0x01 + (i<<4);
        neighborRecordTable[i].longAddr.v[2] = 0x02 + (i<<4);
        neighborRecordTable[i].longAddr.v[3] = 0x03 + (i<<4);
        neighborRecordTable[i].longAddr.v[4] = 0x04 + (i<<4);
        neighborRecordTable[i].longAddr.v[5] = 0x05 + (i<<4);
        neighborRecordTable[i].longAddr.v[6] = 0x06 + (i<<4);
        neighborRecordTable[i].longAddr.v[7] = 0x07 + (i<<4);
        neighborRecordTable[i].shortAddr.Val = 0xBB00 + i;
        neighborRecordTable[i].panID.Val = 0xCC00 + i;
        neighborRecordTable[i].deviceInfo.Val = 0xAB0CDE00 + i;
        neighborRecordTable[i].LogicalChannel = 100 + i;
#ifdef I_SUPPORT_SECURITY
        neighborRecordTable[i].bSecured = (i&0x0001) ? TRUE :
FALSE;
#endif
    } // Fine del ciclo for.

    // --- Scrittura in memoria di programma.
    for(i=0; i<MAX_NEIGHBORS; i++){
        currentNeighborRecord = neighborRecordTable[i];
        pCurrentNeighborRecord = neighborTable[i];
        PutNeighborRecord(pCurrentNeighborRecord,
&currentNeighborRecord);
    }

    // --- Lettura dalla memoria di programma.
    for(i=0; i<MAX_APS_ADDRESSES; i++){
        pCurrentNeighborRecord = neighborTable[i];
        GetNeighborRecord(&currentNeighborRecord,
pCurrentNeighborRecord);
    }

    return 0;
}
```

Cod. 5.11.12: Parte integrante nella funzione main per testare le funzioni di storage della tabella dei vicini.

In tale codice si dichiarano tre variabili: l'array *neighborRecordTable* di tipo NEIGHBOR_RECORD, il quale contiene i valori dei record da salvare; *pCurrentNeighborRecord,* puntatore alla locazione di destinazione di un record, inizializzato ad un elemento dell'array di remapping prima di ogni scrittura; *currentNeighborRecord* utilizzato come cursore tra i record da salvare in memoria.

L'inizializzazione dell'elemento i–esimo dell'array *neighborRecordTable* avviene all'interno del primo ciclo iterativo nel seguente modo: di ogni byte del campo *longAddr* la prima cifra esadecimale è impostata al valore dell'indice *i* del ciclo, la seconda al valore della posizione che lo stesso byte occupa all'interno di *longAddr*; il valore del campo *panID* è impostato al valore 0xCC00 + *i*, pertanto il primo *panID* avrà valore 0xCC00, il secondo 0xCC01 e così via fino a 0xCC18; il campo *deviceInfo* è impostato al valore 0xAB0CDE00 + *i*; il campo *LogicalChannel* è impostato al valore di 100 + *i*; il campo *shortAddr* è impostato al valore 0xBB00 + *i*.

Il salvataggio in memoria di programma comporta la scrittura delle locazioni della FLASH come visibile in Fig. 5.11.1: i valori sono memorizzati a partire dall'indirizzo esadecimale 0x4DC, poiché questa è la prima locazione libera dopo la memorizzazione della chiave di validazione dei binding. Tale allineamento, come osservato nel paragrafo 5.11.3, è ottenuto grazie alla gestione degli indirizzamenti tramite offset e dimensioni relativi ad ogni informazione da salvare in memoria.

In figura sono evidenziati in giallo i valori relativi al primo record della tabella dei vicini: le prime quattro parole di istruzione contengono i valori del campo *longAddr*, i quali compaiono in una parola di istruzione da destra verso sinistra, poiché, come spiegato nella sezione 5.3.2 ed in particolare in Fig. 5.3.1, una parola di istruzione occupa due indirizzi e di questi il più piccolo corrisponde alla parte meno significativa; la quinta parola di istruzione contiene il valore del campo *shortAddr*; la sesta il valore del campo *panID*; la settima e l'ottava, lette da destra verso sinistra, il valore del campo *deviceInfo* e la nona

il valore del campo *LogicalChannel*.

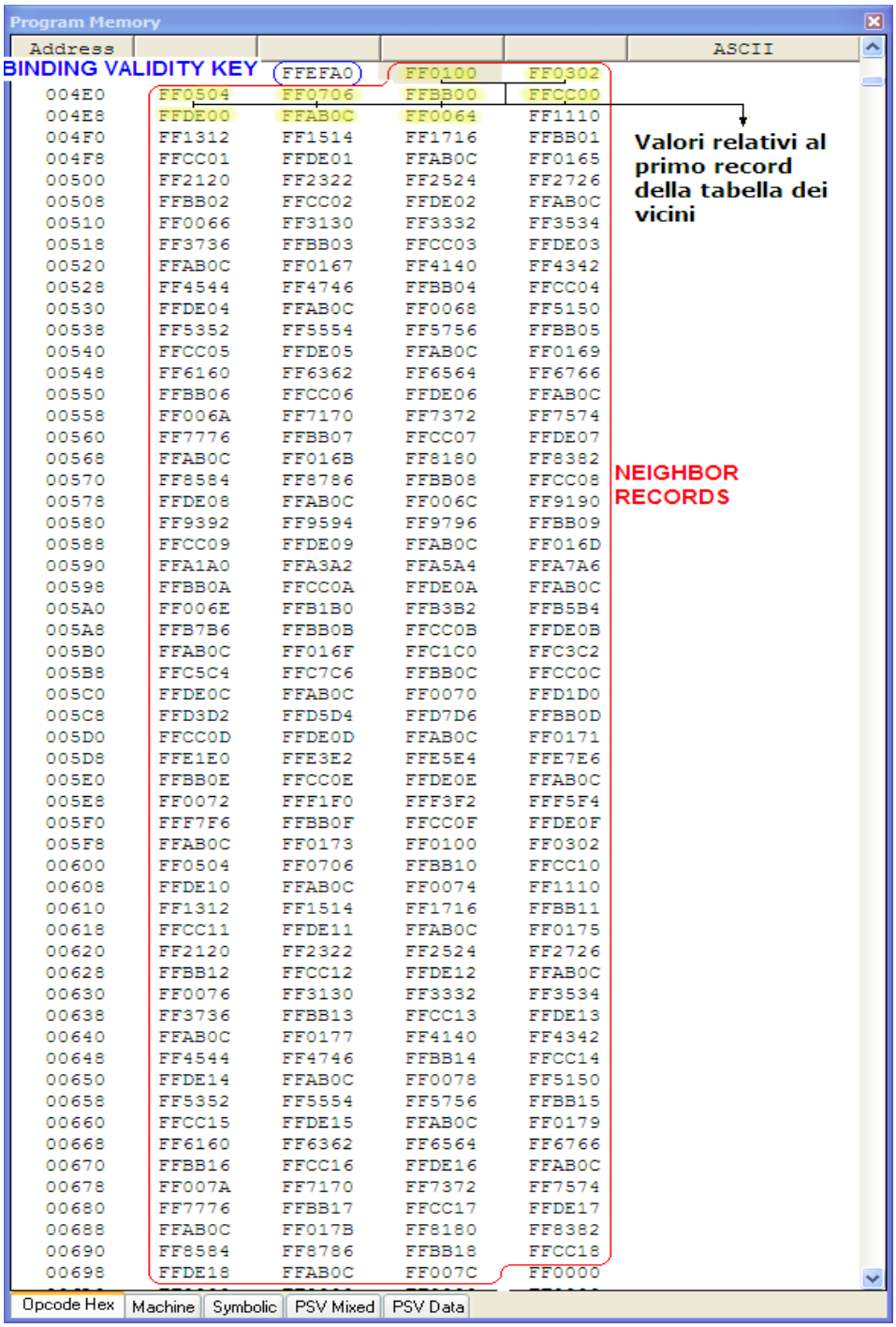

Fig. 5.11.1: Valore dei record della tabella dei vicini salvati in memoria di programma.

Confrontando i valori della figura con le inizializzazioni dell'array *neighborRecordTable* nel codice 5.11.12, si può affermare che la scrittura avviene correttamente.

5.12 – Esplorazione di Neighbor Table Info

In seguito a quanto riportato nella sezione 5.4 – Utilizzo delle funzioni di storage, si procede alla ricerca delle funzioni PutNeighborTableInfo() e GetNeighborTableInfo() definite nel file zNVM.h alle righe 312 e 308 rispettivamente.

Prima di procedere ad esaminare le sopracitate funzioni è opportuno introdurre il tipo di dato NEIGHBOR_TABLE_INFO, definito come nel riquadro di codice 5.12.1, il quale consiste in una struttura adatta a contenere le informazioni relative alla tabella dei vicini. Come si può osservare dal codice la struttura è diversa a seconda della implementazione del tipo di nodo della rete ZigBee. Essendo il nodo NCAP il coordinatore della rete, la macro di configurazione definita in questo ambito è

```
typedef struct _NEIGHBOR_TABLE_INFO
{
    WORD        validityKey;
    BYTE        neighborTableSize;

#ifndef I_AM_COORDINATOR
    BYTE        parentNeighborTableIndex;
#endif

#ifndef I_AM_END_DEVICE
    BYTE        depth;
    SHORT_ADDR  cSkip;
    SHORT_ADDR  nextEndDeviceAddr;
    SHORT_ADDR  nextRouterAddr;
    BYTE        numChildren;
    BYTE        numChildRouters;
    union _flags{
        BYTE    Val;
        struct _bits
        {
            BYTE bChildAddressInfoValid : 1;
        } bits;
    }flags;
#endif

} NEIGHBOR_TABLE_INFO;
```

Cod. 5.12.1: Definizione della struttura NEIGHBOR_TABLE_INFO.

I_AM_COORDINATOR, mentre *I_AM_DEVICE* è non definita.

5.12.1 - Funzione PutNeighborTableInfo

La funzione in esame si occupa di memorizzare le informazioni relative alla tabella dei vicini all'interno della memoria di programma del

```
#define PutNeighborTableInfo()
 NVMWrite((NVM_ADDR*)&neighborTableInfo,
(BYTE*)&currentNeighborTableInfo, sizeof(NEIGHBOR_TABLE_INFO))
```

Cod. 5.12.2: Definizione originale della funzione PutNeighborTableInfo.

microcontrollore. La sua definizione originale, tratta dall'elenco di Cod. 5.4.1 è visibile nel frammento di codice 5.12.2, dove la variabile *neighborRecordTableInfo* dovrebbe essere il riferimento alla locazione nella memoria FLASH nella quale sarà memorizzata la chiave di validazione; in realtà nella sua dichiarazione originaria non è specificato l'attributo *space(prog)* e pertanto tale variabile è allocata in RAM; la variabile *currentNeighborTableInfo* è definita a livello globale e contiene tutte le informazioni relative alla tabella dei vicini, la sua esistenza permette alla funzione di non dover accettare alcun argomento pertanto il prototipo alla sua chiamata coincide con il nome della funzione.

Osservando il frammento di codice 5.12.1 è possibile affermare che lo spazio in memoria da riservare alle informazioni relative alla tabella dei vicini è di 14 byte, anche se per di questi ne saranno utilizzati solamente 13.

5.12.2 - Funzione GetNeighborTableInfo

La funzione in esame si occupa di leggere le informazioni relative alla tabella dei vicini dalla memoria di programma del microcontrollore e di trasferirne il valore in RAM. La sua definizione originale, tratta dall'elenco di Cod. 5.4.2 è visibile nel frammento di codice 5.12.3.

La funzione è del tutto analoga a quella di memorizzazione: non

accetta argomenti, legge il valore dalla locazione riferita da *neighborTableInfo* e la trasferisce nella struttura riferita da *currentNeighborTableInfo*.

```
#define GetNeighborTableInfo()
NVMRead((BYTE *)&currentNeighborTableInfo,
        (ROM void*)&neighborTableInfo,
sizeof(NEIGHBOR_TABLE_INFO))
```

Cod. 5.12.3: Definizione originale della funzione GetNeighborTableInfo.

5.12.3 - Considerazioni

Nei precedenti paragrafi sono state presentate le funzioni di salvataggio e di lettura delle informazioni relative alla tabella dei vicini, verso e dalla memoria di programma; in particolare si è visto il loro comportamento ed i prototipi alle relative chiamate. In questa sezione ci si occuperà delle dichiarazioni dei parametri necessari alla corretta gestione della memoria di programma al fine di riservarne lo spazio necessario alle informazioni e saranno presentati gli aggiornamenti delle definizioni delle funzioni PutNeighborTableInfo e GetNeighborTableInfo.

La dichiarazione dell'offset relativo alle informazioni sulla tabella dei

```
/** neighborRecordOffset indica quanti byte intercorrono tra la
 *  locazione iniziale di NVMBuffer e la locazione dove ha inizio
 *  la memorizzazione dei neighbor record.
 */
#define neighborTableInfoOffset     neighborRecordOffset +
                                    neighborRecordMapSize

/** Dimensione in BYTE dello spazio da allocare in memoria di
 *  programma riservato ai neighbor record.
 */
#define neighborTableInfoSize sizeof(NEIGHBOR_TABLE_INFO)

/** Definizione della variabile neighborTableInfo. La variabile
 *  contiene è un puntatore alla locazione in memoria di programma
 *  nella quale devono essere salvate le informazioni relative
 *  alla tabella dei vicini.
 */
extern ROM BYTE * neighborTableInfo;
```

Cod. 5.12.4: Definizione dei parametri per la memorizzazione delle informazioni relative alla tabella dei vicini.

vicini, analogamente a quanto visto per tutte le informazioni precedenti, deve tenere conto dell'offset relativo all'informazione memorizzata precedentemente: i record della stessa tabella dei vicini. Come osservato nel paragrafo 5.12.1, lo spazio in memoria richiesto è di 14 byte, pertanto è possibile procedere alla definizione della dimensione. Nel file NVM.h è necessario inoltre definire la variabile di remapping, attraverso la parola chiave *extern*, al fine di poterne fare riferimento all'interno del file Main.c. Tali definizioni sono visibili nel frammento di codice 5.12.4.

La dichiarazione della variabile di remapping, avviene nel file NVM.c ed è inizializzata ad un'opportuna locazione di memoria di programma grazie all'offset, come visibile nel frammento di codice 5.12.5.

```
ROM BYTE * neighborTableInfo = NVMBuffer +
                               neighborTableInfoOffset;
```

Cod. 5.12.5: Dichiarazione e inizializzazione della variabile di remapping.

Una volta completate le definizioni dei parametri per il salvataggio della chiave nella memoria di programma è possibile procedere alla ridefinizione delle funzioni di lettura e scrittura.

Secondo quanto visto nel paragrafo 5.12.1, la funzione PutNeighborTableInfo può essere ridefinita come nel frammento di codice 5.12.6.

Rispetto alla definizione originale, riportata nel riquadro di codice 5.12.2, si è omesso il simbolo "&" avanti al primo parametro. Questa modifica si rende necessaria in quanto la variabile di remapping è stata dichiarata come puntatore.

```
#define PutNeighborTableInfo()
NVMWrite((BYTE*)neighborTableInfo,
    (BYTE*)&currentNeighborTableInfo, sizeof(NEIGHBOR_TABLE_INFO))
```

Cod. 5.12.6: Definizione aggiornata della funzione PutBindingValidityKey.

Secondo quanto riportato nel paragrafo 5.12.2, la funzione GetNeighborTableInfo può essere ridefinita come nel frammento di codice 5.12.7.

```
#define GetNeighborTableInfo()
NVMRead((BYTE*)neighborTableInfo,
        (BYTE*)&currentNeighborTableInfo,
sizeof(NEIGHBOR_TABLE_INFO))
```

Cod. 5.12.7: Definizione aggiornata della funzione GetNeighborTableInfo.

Rispetto alla definizione originale, riportata nel riquadro di codice 5.12.3 si è variato l'ordine dei primi due parametri, in conseguenza all'inversione dei parametri della funzione NVMRead aggiornata nel paragrafo 5.3.3, e analogamente alla funzione di scrittura si è omesso il simbolo "&" avanti al riferimento in memoria di programma.

Gli aggiornamenti non comportano alcuna modifica al prototipo delle chiamate, pertanto nel codice dello stack non è necessario effettuare alcuna ulteriore modifica.

5.12.4 - Codice di esempio

Al fine di testare la validità delle ipotesi e delle considerazioni trattate nei paragrafi 5.12.1, 5.12.2 e 5.12.3, si amplia il progetto creato appositamente (introdotto nel paragrafo 5.5.4) con i frammenti di codice presentati nei paragrafi sopracitati e con una parte di codice, inserita nella funzione main, che utilizza le funzioni di storage relative alle informazioni sulla tabella dei vicini.

Tale codice è visibile nel frammento 5.12.8, nel quale sono state omesse le parti relative alle informazioni trattate precedentemente e contiene: una parte di dichiarazione delle variabili; una parte di inizializzazione dei dati da memorizzare, la scrittura delle informazioni in memoria di programma; la lettura di queste dalla memoria FLASH.

Nel codice si dichiara una sola variabile: *currentNeighborTableInfo* di tipo NEIGHBOR_TABLE_INFO, e ne si inizializzano i vari campi. Il salvataggio in memoria di programma comporta la scrittura delle locazioni della FLASH come visibile in Fig. 5.12.1: i valori sono memorizzati a partire dall'indirizzo esadecimale 0xE9E, poiché questa è la prima locazione libera dopo la memorizzazione dei record della tabella. L'allineamento, come osservato nel paragrafo 5.12.3, è ottenuto grazie alla gestione degli indirizzamenti tramite offset e dimensioni relativi ad ogni informazione da salvare in memoria. Confrontando i valori della Fig. 5.12.1 con quelli della variabile

```
int main(){
    // DICHIARAZIONE DELLE VARIABILI
      NEIGHBOR_TABLE_INFO currentNeighborTableInfo;

    // --- Inizializzazione:
    currentNeighborTableInfo.validityKey = 0xABCD;
    currentNeighborTableInfo.neighborTableSize = 0xEF;
    currentNeighborTableInfo.depth = 0x35;
    currentNeighborTableInfo.nextEndDeviceAddr.Val = 0xE30D;
    currentNeighborTableInfo.nextRouterAddr.Val = 0xA13F;
    currentNeighborTableInfo.numChildren = 0xBB;
    currentNeighborTableInfo.numChildRouters = 0xCC;
    currentNeighborTableInfo.cSkip.Val = 0x1234;
    currentNeighborTableInfo.flags.Val = 0xDD;

    // --- Scrittura in memoria di programma.
    PutNeighborTableInfo();

    // --- Lettura dalla memoria di programma.
    currentNeighborTableInfo.validityKey = 0x0000;
    currentNeighborTableInfo.neighborTableSize = 0x00;
    currentNeighborTableInfo.depth = 0x00;
    currentNeighborTableInfo.nextEndDeviceAddr.Val = 0x0000;
    currentNeighborTableInfo.nextRouterAddr.Val = 0x0000;
    currentNeighborTableInfo.numChildren = 0x00;
    currentNeighborTableInfo.numChildRouters = 0x00;
    currentNeighborTableInfo.flags.Val = 0x00;
    currentNeighborTableInfo.cSkip.Val = 0x0000;
    GetNeighborTableInfo();

    return 0;
}
```

Cod. 5.12.8: Parte integrante nella funzione main per testare le funzioni di storage della chiave di validazione.

currentNeighborTableInfo nel codice 5.12.8, si può affermare che la scrittura avviene correttamente.

Si osservi che, rispetto a quanto visto nei paragrafi precedenti, le informazioni sono memorizzate in locazioni di memoria diverse. La motivazione di questo aspetto è da attribuire all'incremento del valore dell'argomento *aligned*, nella dichiarazione di *NVMBuffer*, al fine di allocare lo spazio di memoria riservato ai dati della rete ZigBee tra parole di istruzione non utilizzate. Questo argomento è stato trattato nel paragrafo 5.3.2.

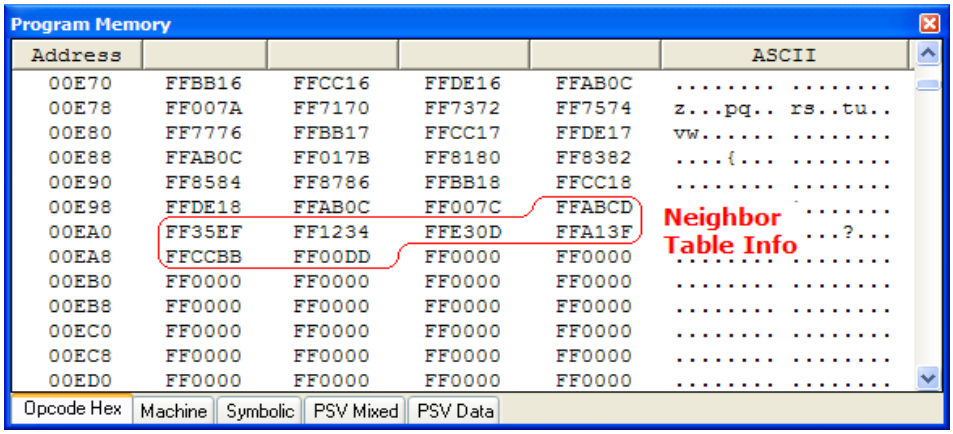

Fig. 5.12.1: Memoria di programma dopo il salvataggio delle informazioni.

Prima di effettuare la lettura si inizializzano a zero tutti i campi della la variabile allocata in RAM, al fine di osservarne l'aggiornamento a lettura ultimata, visibile in Fig. 5.12.2.

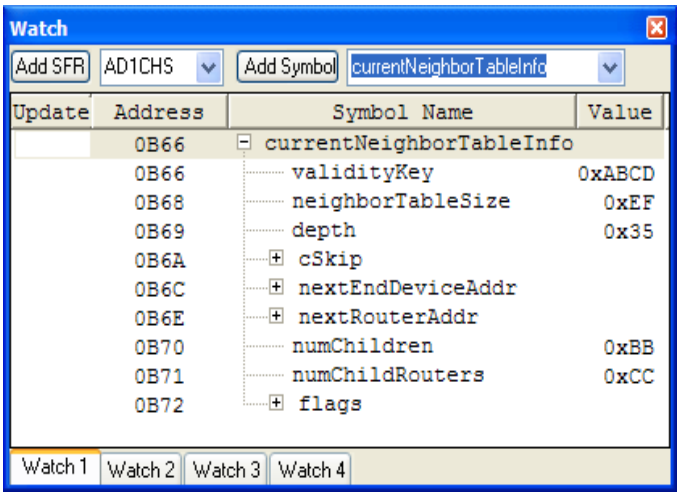

Fig. 5.12.2: Risultati prodotti dalla simulazione.

La figura mostra i valori della struttura currentNeighborTableInfo dopo la lettura, i quali coincidono con quelli a cui la stessa struttura è stata inizializzata prima della scrittura (nel Cod. 5.12.8).

5.13 – Esplorazione di Routing Entry

In seguito a quanto riportato nella sezione 5.4 – Utilizzo delle funzioni di storage, si procede alla ricerca delle funzioni PutRoutingEntry(x, y) e GetRoutingEntry(x, y) definite nel file zNVM.h alle righe 317 e 316 rispettivamente.

Prima di procedere ad esaminare le sopracitate funzioni è opportuno introdurre alcuni tipi di dato che interessano il contesto, iniziando da quelli più semplici e procedendo verso quelli più complessi.

```
typedef enum _ROUTE_STATUS{
    ROUTE_ACTIVE                    = 0x00,
    ROUTE_DISCOVERY_UNDERWAY        = 0x01,
    ROUTE_DISCOVERY_FAILED          = 0x02,
    ROUTE_INACTIVE                  = 0x03
} ROUTE_STATUS;
```

Cod. 5.13.1: Definizione dell'enumerazione ROUTE_STATUS.

Il tipo di dato ROUTE_STATUS, definito come nel riquadro di codice 5.13.1, consiste in una enumerazione utile a definire il valore dello stato di Route. Un'enumerazione nel contesto del linguaggio C è una ridefinizione del tipo di dato intero, pertanto occupa in memoria uno spazio di 2 byte.

```
typedef struct _ROUTING_ENTRY{
    SHORT_ADDR      destAddress;
    ROUTE_STATUS    status;
    SHORT_ADDR      nextHop;
} ROUTING_ENTRY;
```

Cod. 5.13.2: Definizione della struttura ROUTING_ENTRY.

Il tipo di dato ROUTING_ENTRY, definito come nel riquadro di codice 5.13.2, consiste in una struttura adatta a contenere un elemento della tabella di routing. Osservando le definizioni dei vari campi è possibile constatare che lo spazio che la struttura occupa in memoria è di 6 byte.

5.13.1 – Funzione PutRoutingEntry

La funzione in esame si occupa di memorizzare un elemento della tabella di routing all'interno della memoria di programma del

microcontrollore. La sua definizione originale, tratta dall'elenco di Cod. 5.4.1 è visibile nel frammento di codice 5.13.3.

```
#define PutRoutingEntry(x, y)
NVMWrite((NVM_ADDR *)x, (BYTE*)y, sizeof(ROUTING_ENTRY))
```

Cod. 5.13.3: Definizione originale della funzione PutRoutingEntry.

Osservando il codice del frammento 5.13.3 si nota che i parametri da passare come argomento alla chiamata della funzione sono: la locazione in memoria di programma nella quale salvare la entry e il riferimento ad una variabile di tipo ROUTING_ENTRY, contenente i dati da memorizzare. Ovvero il prototipo delle chiamate alla funzione sono tutte del tipo mostrato nel frammento di codice 5.13.4.

```
PutRoutingEntry( pCurrentRoutingEntry, &currentRoutingEntry );
```

Cod. 5.13.4: Prototipo delle chiamate a PutRoutingEntry.

In questo codice la variabile *currentRoutingEntry*, di tipo ROUTING_ENTRY, è utilizzata come cursore delle varie entry della tabella di routing e contiene il valore di quella che deve essere salvata in memoria. La variabile *pCurrentRoutingEntry* rappresenta un puntatore alla locazione nella quale salvare i dati ed è inizializzato ad un elemento dell'array *routingTable*. Tale array è di remapping ma nella dichiarazione originale è stato commesso l'errore di non specificare l'attributo *space(prog)* e pertanto è allocato in RAM anziché in memoria di programma, come visibile nel frammento di codice 5.13.5, inoltre non è stato inizializzato ed essendo costante non può essergli assegnato un valore neppure in una fase successiva.

```
extern ROM ROUTING_ENTRY routingTable[ROUTING_TABLE_SIZE];
```

Cod. 5.13.5: Dichiarazione originale dell'array di remapping.

La costante *ROUTING_TABLE_SIZE* indica il numero totale delle entry della tabella ed è definita al valore 16. Si evince dunque che lo spazio totale richiesto per salvare tutte le Routing Entry è 96 byte, ottenuto

moltiplicando la dimensione di una singola entry – di 6 byte come visto nel paragrafo 5.13 – per il valore di *ROUTING_TABLE_SIZE*. È pertanto possibile procedere alle definizioni dell'offset su NVMBuffer, della dimensione e della variabile di remapping al fine di ottenere i riferimenti alla memoria di programma, analogamente a quanto già visto per le informazioni trattate nei paragrafi precedenti. Tali definizioni sono scritte nel file NVM.h e riportate nel frammento di codice 5.13.6.

```
/** Numero di Routing Entry da salvare in memoria di programma. */
#define ROUTING_TABLE_SIZE 16

/** routingEntryOffset indica quanti byte intercorrono tra la
 *  locazione iniziale di NVMBuffer e la locazione dove ha inizio
 *  la memorizzazione dei Routing Entry.
 */
#define routingEntryOffset    neighborTableInfoOffset +
                              neighborTableInfoSize

/** Dimensione in BYTE dello spazio da allocare in memoria di
 *  programma riservato alle Routing Entry.
 */
#define routingEntryMapSize   sizeof(ROUTING_ENTRY) *
                              ROUTING_TABLE_SIZE

/** Definizione dell'array routingTable. L'elemento i-esimo di
 *  questo array contiene  un puntatore alla locazione in memoria
 *  di programma nella quale deve essere salvata l'i-esima Routing
 *  Entry.
 */
extern ROM BYTE * routingTable[ROUTING_TABLE_SIZE];
```

Cod. 5.13.6: Definizioni nel file NVM.h relative alle informazioni sulle routing entry.

Si osservi che la definizione dell'offset tiene conto dello stesso offset e delle dimensioni relative alla tabella dei vicini. Nel file NVM.c è necessario dichiarare la variabile di remapping, *routingTable*, definita nel file di intestazione, inizializzandola correttamente alle opportune locazioni di NVMBuffer, come visibile nel frammento di codice 5.13.7.

L'inizializzazione fa uso della definizione del relativo offset: in tal modo *routingTable* punta alla prima locazione libera dopo le informazioni relative alla tabella dei vicini.

```
ROM BYTE * routingTable[ROUTING_TABLE_SIZE] = {
      NVMBuffer + routingEntryOffset + 0*sizeof(ROUTING_ENTRY),
      NVMBuffer + routingEntryOffset + 1*sizeof(ROUTING_ENTRY),
      NVMBuffer + routingEntryOffset + 2*sizeof(ROUTING_ENTRY),
      NVMBuffer + routingEntryOffset + 3*sizeof(ROUTING_ENTRY),
      NVMBuffer + routingEntryOffset + 4*sizeof(ROUTING_ENTRY),
      NVMBuffer + routingEntryOffset + 5*sizeof(ROUTING_ENTRY),
      NVMBuffer + routingEntryOffset + 6*sizeof(ROUTING_ENTRY),
      NVMBuffer + routingEntryOffset + 7*sizeof(ROUTING_ENTRY),
      NVMBuffer + routingEntryOffset + 8*sizeof(ROUTING_ENTRY),
      NVMBuffer + routingEntryOffset + 9*sizeof(ROUTING_ENTRY),
      NVMBuffer + routingEntryOffset + 10*sizeof(ROUTING_ENTRY),
      NVMBuffer + routingEntryOffset + 11*sizeof(ROUTING_ENTRY),
      NVMBuffer + routingEntryOffset + 12*sizeof(ROUTING_ENTRY),
      NVMBuffer + routingEntryOffset + 13*sizeof(ROUTING_ENTRY),
      NVMBuffer + routingEntryOffset + 14*sizeof(ROUTING_ENTRY),
      NVMBuffer + routingEntryOffset + 15*sizeof(ROUTING_ENTRY)
};
```

Cod. 5.13.7: Dichiarazione e inizializzazione della variabile di remapping.

5.13.2 - Funzione GetRoutingEntry

La funzione GetRoutingEntry si occupa della lettura degli elementi della tabella di routing dalla memoria di programma del microcontrollore; originariamente è definita come visibile nel frammento di codice 5.13.8, tratto dall'elenco di Cod. 5.4.2.

```
#define GetRoutingEntry(x, y)
NVMRead((BYTE *)x, (ROM void*)y, sizeof(ROUTING_ENTRY))
```

Cod. 5.13.8: Definizione originale della funzione GetRoutingEntry.

Analogamente alla funzione di memorizzazione, accetta due parametri: il primo contiene il riferimento alla locazione in RAM nella quale deve essere trasferito il dato letto; il secondo contiene il riferimento alla locazione in memoria di programma nella quale si trova il record da leggere; ovvero il prototipo delle chiamate a GetRoutingEntry è quello visibile nel frammento di codice 5.13.9, dove la variabile *currentRoutingEntry* è di tipo ROUTING_ENTRY.

```
GetRoutingEntry(&currentRoutingEntry, pCurrentRoutingEntry);
```

Cod. 5.13.9: Prototipo delle chiamate a GetRoutingEntry.

5.13.3 - Considerazioni

Nei precedenti paragrafi sono state presentate le funzioni di salvataggio e di lettura dei record delle Routing Entry verso e dalla memoria di programma; in particolare si è visto il loro comportamento ed i prototipi alle relative chiamate. Nel paragrafo 5.13.1 inoltre, sono stati introdotti: offset, dimensioni e variabile di remapping per permettere la corretta gestione della memoria di programma, al fine di riservarne lo spazio necessario ai vari record. In questa sezione, di tali funzioni saranno presentati gli aggiornamenti delle definizioni.

Secondo quanto visto nel paragrafo 5.13.1, la funzione PutRoutingEntry può essere ridefinita come nel frammento di codice 5.13.10.

```
#define PutRoutingEntry(x, y)
NVMWrite((BYTE*)x, (BYTE*)y, sizeof(ROUTING_ENTRY))
```

Cod. 5.13.10: Definizione aggiornata della funzione PutRoutingEntry.

Rispetto alla definizione originale, riportata nel riquadro di codice 5.13.3, si è mutato solamente il tipo del cast del primo parametro, da NVM_ADDR a BYTE.

Secondo quanto riportato nel paragrafo 5.13.2, la funzione GetRoutingEntry può essere ridefinita come nel frammento di codice 5.13.11.

```
#define GetRoutingEntry(x, y)
NVMRead((BYTE*)y, (BYTE*)x, sizeof(ROUTING_ENTRY))
```

Cod. 5.13.11: Definizione aggiornata della funzione GetRoutingEntry.

Rispetto alla definizione originale, riportata nel riquadro di codice

5.13.8, si è mutato l'ordine dei primi due argomenti (questo cambiamento è la conseguenza dell'inversione dei parametri nella funzione NVMRead, aggiornata nel paragrafo 5.3.3) e il tipo del cast del secondo parametro da void a BYTE.

Le definizioni appena introdotte altro non sono che una mappatura delle funzioni NVMWrite ed NVMRead, e sono tutte inserite nel file NVM.h, insieme alle definizioni di offset, dimensioni e variabile di remapping.

5.13.4 – Codice di esempio

Al fine di testare la validità delle ipotesi e delle considerazioni trattate nei paragrafi 5.13.1, 5.13.2 e 5.13.3, si amplia il progetto creato appositamente (introdotto nel paragrafo 5.5.4) con i frammenti di codice presentati nei paragrafi sopracitati e con una parte di codice, inserita nella funzione main, che utilizza le funzioni di storage relative agli elementi della tabella di routing.

Quest'ultima è visibile nel frammento di codice 5.13.12, nel quale sono state omesse le parti relative alle informazioni trattate precedentemente e contiene: una parte di dichiarazione delle variabili; una parte di inizializzazione in RAM dei dati da scrivere nella memoria di programma; la scrittura delle routing entry; la lettura di queste dalla memoria FLASH e relativo trasferimento del valore in RAM.

Nel codice si dichiarano tre variabili: *routingEntryTable,* un array allocato in RAM contenente tutte le routing entry da salvare in memoria di programma; *pCurrentRoutingEntry,* un puntatore alla locazione nella memoria FLASH il quale sarà inizializzato ad un elemento della array di remapping prima di ogni scrittura; *currentRoutingEntry*, di tipo ROUTING_ENTRY, utilizzata come cursore all'interno dei cicli di scrittura e lettura.

Il primo dei tre cicli *for* provvede all'inizializzazione dei record da salvare. L'inizializzazione è effettuata in modo tale che i campi del

record i-esimo siano impostati come segue: il campo *destAddr* al valore 0xBF00 + *i*; il campo *status* costantemente al valore *ROUTE_INACTIVE*; il campo *nextHop* al valore 0xCD00 + *i*.

```
int main(){
    // DICHIARAZIONE DELLE VARIABILI
    ROUTING_ENTRY routingEntryTable[ROUTING_TABLE_SIZE];
    ROM BYTE * pCurrentRoutingEntry;
    ROUTING_ENTRY currentRoutingEntry;

    // --- Inizializzazione dei valori in RAM:
    for(i=0; i<ROUTING_TABLE_SIZE; i++){
        routingEntryTable[i].destAddress.Val = 0xBF00 + i;
        routingEntryTable[i].status = ROUTE_INACTIVE;
        routingEntryTable[i].nextHop.Val = 0xCD00 + i;
    }

    // --- Scrittura in memoria di programma.
    for(i=0; i<ROUTING_TABLE_SIZE; i++){
        pCurrentRoutingEntry = routingTable[i];
        currentRoutingEntry = routingEntryTable[i];
        PutRoutingEntry(pCurrentRoutingEntry,
                        &currentRoutingEntry);
    }

    // --- Lettura dalla memoria di programma.
    for(i=0; i<ROUTING_TABLE_SIZE; i++){
        pCurrentRoutingEntry = routingTable[i];
        GetRoutingEntry(&currentRoutingEntry,
                        pCurrentRoutingEntry);
    }
```

Cod. 5.13.12: Parte integrante nella funzione main per testare le funzioni di storage delle entry.

Il secondo dei cicli *for* provvede, ad ogni iterazione, a salvare una routing entry. Prima di ogni salvataggio l'iterazione aggiorna la variabile cursore *currentRoutingEntry* e imposta la variabile puntatore *pCurrentRoutingEntry* alla locazione di programma nella quale scrivere i dati, che corrisponde all'i-esimo elemento dell'array di remapping.

Il salvataggio in memoria di programma comporta la scrittura delle locazioni della FLASH come visibile in Fig. 5.13.1: i valori sono memorizzati a partire dall'indirizzo esadecimale 0xEAC, poiché questa è la prima locazione libera dopo la memorizzazione delle informazioni

relative alla tabella dei vicini.

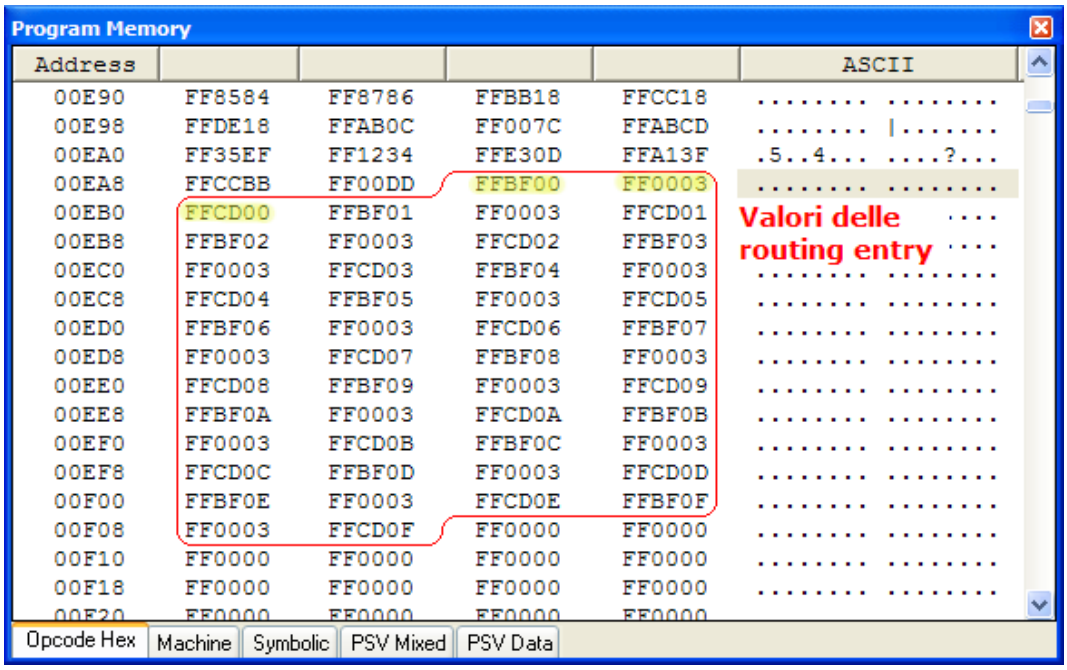

Fig. 5.13.1: Valori delle routing entry salvate nella memoria di programma.

In figura sono evidenziati in giallo i valori relativi alla prima routing entry: la prima parola di istruzione contiene il valore del campo *destAddr*; la seconda contiene il valore del campo *status* e la terza contiene il valore del campo *nextHop*. Come si può vedere dalla figura, i valori coincidono con quelli dei campi corrispondenti.

La funzione di lettura ha il compito di leggere i dati dalla FLASH e di trasferirli in RAM. Per mezzo del simulatore MPLAB SIM integrato nell'ambiente

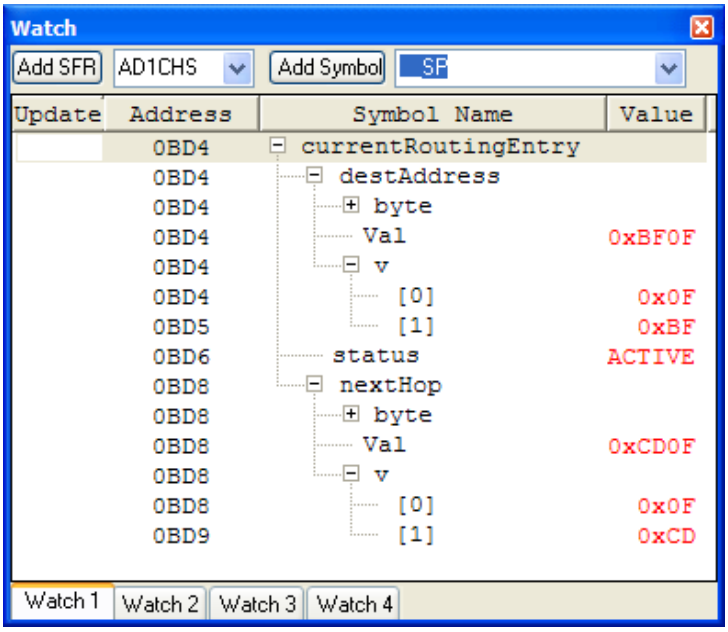

Fig. 5.13.2: Risultati prodotti dalla simulazione.

di sviluppo, è possibile osservare l'aggiornamento della variabile *currentRoutingEntry*, la quale a lettura ultimata assume il valore visibile in Fig. 5.13.2, in accordo con i valori salvati, visibili in Fig. 5.13.1 e con quelli impostati nel codice 5.13.12.

5.14 – Esplorazione di Trust Center Address

In seguito a quanto riportato nella sezione 5.4 – Utilizzo delle funzioni di storage, si procede alla ricerca delle funzioni PutTrustCenterAddress(x) e GetTrustCenterAddress(x) definite nel file zNVM.h alle righe 321 e 319 rispettivamente.

5.14.1 – Funzione PutTrustCenterAddress

La funzione in esame si occupa di memorizzare il Trust Center Address all'interno della memoria di programma del microcontrollore. La sua definizione originale, tratta dall'elenco di Cod. 5.4.1 è visibile nel frammento di codice 5.14.1:

```
#define PutTrustCenterAddress(x)
NVMWrite((NVM_ADDR *)&trustCenterLongAddr, (BYTE *)x,
sizeof(LONG_ADDR))
```

Cod. 5.14.1: Definizione originale della funzione PutTrustCenterAddress.

dove la variabile *trustCenterLongAddr* dovrebbe essere il riferimento alla locazione nella memoria FLASH nella quale sarà memorizzata la chiave di validazione; in realtà nella sua dichiarazione originaria non è specificato l'attributo *space(prog)* e pertanto tale variabile è allocata in RAM.

La funzione accetta un solo argomento: il riferimento alla locazione in RAM nella quale si trova il valore dell'indirizzo da memorizzare. Pertanto il prototipo della sua chiamata è quello mostrato nel riquadro di codice 5.14.2, dove l'unico parametro è di tipo LONG_ADDR. La

ridefinizione della funzione dovrà essere compatibile con tale prototipo, al fine di non dover effettuare alcuna modifica di massa all'interno dello stack.

```
PutTrustCenterAddress(&params.APSME_TRANSPORT_KEY_indication.SrcAd
dr);
```

Cod. 5.14.2: Prototipo della chiamata alla funzione PutTrustCenterAddress.

Dalla definizione di Cod. 5.14.1 e dal tipo del parametro *SrcAddr* è possibile affermare che lo spazio in memoria di programma da riservare all'indirizzo è di 8 byte.

5.14.2 - Funzione GetTrustCenterAddress

La funzione in esame si occupa di leggere l'indirizzo del Trust Center dalla memoria di programma del microcontrollore e di trasferirne il valore in RAM. La sua definizione originale, tratta dall'elenco di Cod. 5.4.2 è visibile nel frammento di codice 5.14.3:

```
#define GetTrustCenterAddress(x)
NVMRead((BYTE *)x, (NVM_ADDR *)&trustCenterLongAddr,
sizeof(LONG_ADDR))
```

Cod. 5.14.3: Definizione originale della funzione GetBindingValidityKey.

La funzione accetta un solo argomento: il riferimento alla locazione in RAM nella quale trasferire il valore dell'indirizzo dalla memoria di programma, pertanto il prototipo della sua chiamata è quello mostrato nel riquadro di codice 5.14.4 ed è del tutto analogo a quello della funzione PutTrustCenterAddress.

```
GetTrustCenterAddress(&params.APSME_UPDATE_DEVICE_request.DestAddr
ess);
```

Cod. 5.14.4: Prototipo della chiamata alla funzione GetTrustCenterAddress.

5.14.3 – Considerazioni

Nei precedenti paragrafi sono state presentate le funzioni di salvataggio e di lettura dell'indirizzo del Trust Center, verso e dalla memoria di programma; in particolare si è visto il loro comportamento ed i prototipi alle relative chiamate. In questa sezione ci si occuperà delle dichiarazioni dei parametri necessari alla corretta gestione della memoria di programma al fine di riservarne lo spazio necessario al Trust Center Address e saranno presentati gli aggiornamenti delle definizioni delle funzioni PutTrustCenterAddress e GetTrustCenterAddress.

```
/** trustCenterAddressOffset indica quanti byte intercorrono tra
 *  la locazione iniziale di NVMBuffer e la locazione dove ha
 *  inizio la memorizzazione del trust center address.
 */
#define trustCenterAddressOffset routingEntryOffset +
routingEntryMapSize

/** Dimensione in BYTE dello spazio da allocare in memoria di
 *  programma riservato al trust center address.
 */
#define trustCenterAddressMapSize sizeof(LONG_ADDR)

/** Definizione della variabile trustCenterLongAddr. Questa
 *  variabile contiene un puntatore alla locazione in memoria di
 *  programma nella quale deve essere salvato il trust center
 *  address.
 */
extern ROM BYTE * trustCenterLongAddr;
```

Cod. 5.14.5: Definizione dei parametri per la memorizzazione del Trust Center Address.

Osservando la Fig. 5.13.1 si nota che la prima locazione libera in memoria di programma è all'indirizzo 0xF0C. Per garantire la scrittura dell'indirizzo in tale locazione è necessario dichiararne l'offset tenendo conto di quello della tabella di routing e delle relative dimensioni. Inoltre è possibile procedere alla definizione della dimensione della chiave, in quanto, come osservato nel paragrafo 5.14.1, lo spazio di memoria richiesto è di 2 byte. Nel file NVM.h è necessario definire la variabile di remapping, attraverso la parola chiave *extern*, al fine di poterne fare riferimento all'interno del file Main.c. Tali definizioni sono

visibili nel frammento di codice 5.14.5.

La dichiarazione della variabile di remapping, avviene nel file NVM.c ed è inizializzata alle opportune locazioni di memoria di programma grazie all'offset dell'indirizzo, come visibile nel frammento di codice 5.14.6.

```
ROM BYTE * trustCenterLongAddr = NVMBuffer +
trustCenterAddressOffset;
```

Cod. 5.14.6: Dichiarazione e inizializzazione della variabile di remapping del Trust Center Address.

Una volta completate le definizioni dei parametri per il salvataggio dell'indirizzo nella memoria di programma è possibile procedere alla ridefinizione delle funzioni di lettura e scrittura.

Secondo quanto visto nel paragrafo 5.14.1, la funzione PutTrustCenterAddress può essere ridefinita come nel frammento di codice 5.14.7.

```
#define PutTrustCenterAddress(x)
NVMWrite((BYTE*)trustCenterLongAddr, (BYTE*)x, sizeof(LONG_ADDR));
```

Cod. 5.14.7: Definizione aggiornata della funzione PutTrustCenterAddress.

Rispetto alla definizione originale, riportata nel riquadro di codice 5.14.1, si è omesso il simbolo "&" avanti al primo parametro. Questa modifica si rende necessaria in quanto la variabile di remapping è stata dichiarata come puntatore.

Secondo quanto riportato nel paragrafo 5.14.2, la funzione GetTrustCenterAddress può essere ridefinita come nel frammento di codice 5.14.8.

```
#define GetTrustCenterAddress(x)
NVMRead((BYTE*)trustCenterLongAddr, (BYTE*)x, sizeof(LONG_ADDR));
```

Cod. 5.14.8: Definizione aggiornata della funzione GetTrustCenterAddress.

Rispetto alla definizione originale, riportata nel riquadro di codice 5.14.3 si è variato il tipo del cast del secondo parametro da NVM_ADDR a

BYTE.

Gli aggiornamenti non comportano alcuna modifica al prototipo delle chiamate, pertanto nel codice dello stack non è necessario effettuare alcuna ulteriore modifica.

5.14.4 - Codice di esempio

Al fine di testare la validità delle ipotesi e delle considerazioni trattate nei paragrafi 5.14.1, 5.14.2 e 5.14.3, si amplia il progetto creato appositamente (introdotto nel paragrafo 5.5.4) con i frammenti di codice presentati nei paragrafi sopracitati e con una parte di codice, inserita nella funzione main, che utilizza le funzioni di storage relative al Trust Center Address.

Tale codice è visibile nel frammento 5.14.9, nel quale sono state omesse le parti relative alle informazioni trattate precedentemente e contiene: una parte di dichiarazione delle variabili; la scrittura dell'indirizzo in memoria di programma; la lettura di questo dalla memoria FLASH.

```
int main(){
    // DICHIARAZIONE DELLE VARIABILI
    LONG_ADDR trustCenterAddressValue =
              {{0xAB,0xCD,0xEF,0xA0,0xB1,0xC2,0xD3,0xE4}};

    // --- Scrittura in memoria di programma.
    PutTrustCenterAddress(&trustCenterAddressValue);

    // --- Lettura dalla memoria di programma.
    for(i=0; i<8; i++){
            trustCenterAddressValue.v[i] = 0x00;
    }
    GetTrustCenterAddress(&trustCenterAddressValue);

    return 0;
}
```

Cod. 5.14.9: Parte integrante nella funzione main per testare le funzioni di storage dell'indirizzo.

In tale codice si dichiara una sola variabile: *trustCenterAddressValue* di

tipo LONG_ADDR, inizializzandola al valore dell'indirizzo da salvare.

Il salvataggio in memoria di programma comporta la scrittura delle locazioni della FLASH come visibile in Fig. 5.14.1: i valori sono

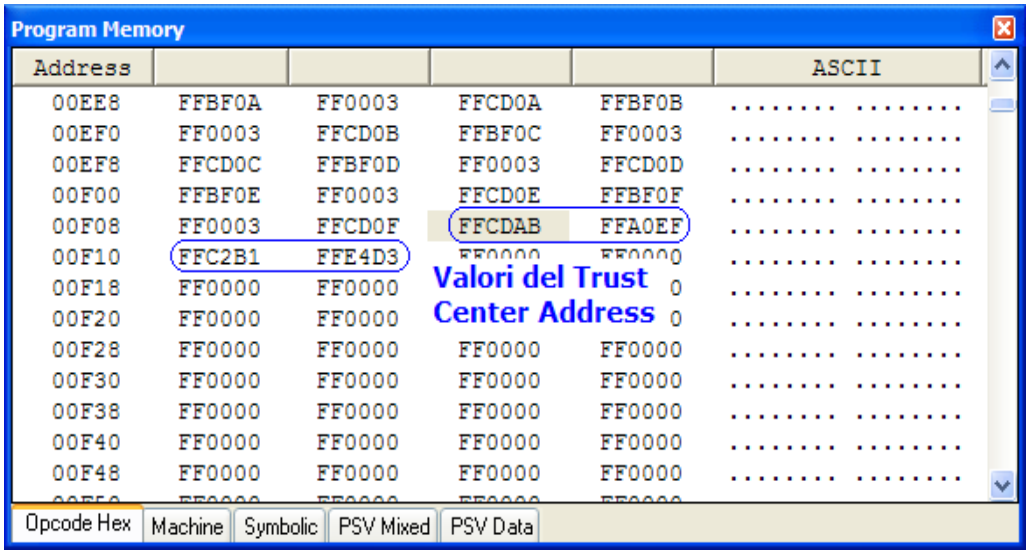

Fig. 5.14.1: Valore del Trust Center Address salvato nella memoria di programma.

memorizzati a partire dall'indirizzo esadecimale 0xF0C, poiché questa è la prima locazione libera dopo la memorizzazione della tabella di routing. Tale allineamento, come osservato nel paragrafo 5.14.3, è ottenuto grazie alla gestione degli indirizzamenti tramite offset e dimensioni relativi ad ogni informazione da salvare in memoria. Confrontando i valori della figura con quelli della variabile *trustCenterAddress* del codice 5.14.9, si può affermare che la scrittura avviene correttamente.

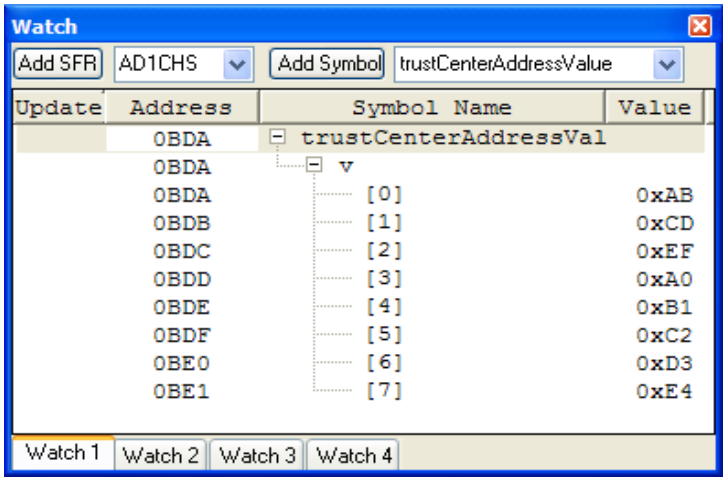

Fig. 5.14.2: Risultati ottenuti con il simulatore MPLAB SIM.

La funzione di lettura ha il compito di leggere i dati dalla FLASH e di trasferirli in RAM. Per mezzo

del simulatore MPLAB SIM, integrato nell'ambiente di sviluppo, è possibile osservare l'aggiornamento della variabile *trustCenterAddress*, la quale a lettura ultimata assume il valore visibile in Fig. 5.14.2, in accordo con il valore salvato, visibile in Fig. 5.14.1 e con quello impostato nel codice 5.14.9.

5.15 – Esplorazione di Network Active Key Number

In seguito a quanto riportato nella sezione 5.4 – Utilizzo delle funzioni di storage, si procede alla ricerca delle funzioni PutNwkAtciveKeyNumber(x) e GetNwkAtciveKeyNumber(x) definite nel file zNVM.h alle righe 324 e 323 rispettivamente.

5.15.1 – Funzione PutNwkActiveKeyNumber

La funzione in esame si occupa di memorizzare il numero della chiave di rete all'interno della memoria di programma del microcontrollore. La sua definizione originale, tratta dall'elenco di Cod. 5.4.1 è visibile nel frammento di codice 5.15.1:

```
#define PutNwkActiveKeyNumber(x)
NVMWrite((NVM_ADDR *)&nwkActiveKeyNumber, (BYTE *)x, 1)
```

Cod. 5.15.1: Definizione originale della funzione PutNwkActiveKeyNumber.

dove la variabile *nwkActiveKeyNumber* dovrebbe essere il riferimento alla locazione nella memoria FLASH nella quale sarà memorizzato il numero della chiave di rete; in realtà nella sua dichiarazione originaria non è specificato l'attributo *space(prog)* e pertanto tale variabile è allocata in RAM.

La funzione accetta un solo argomento: il riferimento alla locazione in RAM nella quale si trova il valore dell'indirizzo da memorizzare.

Pertanto il prototipo della sua chiamata è quello mostrato nel riquadro di codice 5.15.2, dove l'argomento *ActiveKeyIndex* è di tipo BYTE. La ridefinizione della funzione dovrà essere compatibile con tale prototipo, al fine di non dover effettuare alcuna modifica di massa all'interno dello stack.

```
PutNwkActiveKeyNumber(&ActiveKeyIndex);
```

Cod. 5.15.2: Prototipo della chiamata alla funzione PutNwkActiveKeyNumber.

Dalla definizione di Cod. 5.15.1 e dal tipo del parametro *ActiveKeyIndex* è possibile affermare che lo spazio in memoria di programma da riservare all'indirizzo è di un byte, ma siccome l'architettura del microcontrollore è a 16 bit sarà necessario riservarne due.

5.15.2 – Funzione GetNwkActiveKeyNumber

La funzione in esame si occupa di leggere il numero della chiave di rete dalla memoria di programma del microcontrollore e di trasferirne il valore in RAM. La sua definizione originale, tratta dall'elenco di Cod. 5.4.2 è visibile nel frammento di codice 5.15.3:

```
#define GetNwkActiveKeyNumber(x)
NVMRead(( BYTE *)x, (ROM void *)&nwkActiveKeyNumber, 1)
```

Cod. 5.15.3: Definizione originale della funzione GetActiveKeyNumber.

La funzione accetta un solo argomento: il riferimento alla locazione in RAM nella quale trasferire il valore della chiave dalla memoria di programma, pertanto il prototipo della sua chiamata è quello mostrato nel riquadro di codice 5.15.4 ed è del tutto analogo a quello della funzione PutNwkActiveKeyNumber.

```
GetNwkActiveKeyNumber(&ActiveKeyIndex);
```

Cod. 5.15.4: Prototipo della chiamata alla funzione GetActiveKeyNumber.

5.15.3 – Considerazioni

Nei precedenti paragrafi sono state presentate le funzioni di salvataggio e di lettura della chiave di rete, verso e dalla memoria di programma; in particolare si è visto il loro comportamento ed i prototipi alle relative chiamate. In questa sezione ci si occuperà delle dichiarazioni dei parametri necessari alla corretta gestione della memoria di programma al fine di riservarne lo spazio necessario alla chiave di validazione e saranno presentati gli aggiornamenti delle definizioni delle funzioni PutNwkActiveKeyNumber e GetNwkActiveKeyNumber.

Osservando la Fig. 5.14.1 si nota che la prima locazione libera in memoria di programma è all'indirizzo 0xF14. Per garantire la scrittura della chiave di rete in tale locazione è necessario dichiararne l'offset tenendo conto di quello del Trust Center Address e delle relative dimensioni. Inoltre è possibile procedere alla definizione della dimensione della chiave di rete, in quanto, come osservato nel paragrafo 5.15.1, lo spazio di memoria richiesto è di 2 byte. Nel file NVM.h è necessario definire la variabile di remapping, attraverso la parola chiave *extern*, al fine di poterne fare riferimento all'interno del file Main.c. Tali definizioni sono visibili nel frammento di codice 5.15.5.

```
/** nwkActiveKeyNumberOffset indica quanti byte intercorrono tra
 *  la locazione iniziale di NVMBuffer e la locazione dove ha
 *  inizio la memorizzazione della Network Active Key.
 */
#define nwkActiveKeyNumberOffset    trustCenterAddressOffset +
                                    trustCenterAddressMapSize

/** Dimensione in BYTE dello spazio da allocare in memoria di
 *  programma riservato  alla Network Active Key.
 */
#define nwkActiveKeyNumberMapSize sizeof(WORD)

/** Definizione della variabile nwkActiveKeyNumber. Questa
 *  variabile contiene un puntatore alla locazione in memoria di
 *  programma nella quale deve essere salvata la Network Active
 *  Key.
 */
extern ROM BYTE * nwkActiveKeyNumber;
```

Cod. 5.15.5: Definizione dei parametri per la memorizzazione della chiave di rete.

La dichiarazione della variabile di remapping, avviene nel file NVM.c ed è inizializzata alle opportune locazioni di memoria di programma grazie all'offset della chiave di rete, come visibile nel frammento di codice 5.15.6.

```
ROM BYTE * nwkActiveKeyNumber =     NVMBuffer +
                                    nwkActiveKeyNumberOffset;
```

Cod. 5.15.6: Dichiarazione e inizializzazione della variabile di remapping della chiave di rete.

Una volta completate le definizioni dei parametri per il salvataggio della chiave nella memoria di programma è possibile procedere alla ridefinizione delle funzioni di lettura e scrittura.

Secondo quanto visto nel paragrafo 5.15.1, la funzione PutNwkActivekeyNumber può essere ridefinita come nel frammento di codice 5.15.7.

```
#define PutNwkActiveKeyNumber(x)
NVMWrite((BYTE*)nwkActiveKeyNumber, (BYTE*)x, sizeof(BYTE));
```

Cod. 5.15.7: Definizione aggiornata della funzione PutNwkActiveKeyNumber.

Rispetto alla definizione originale, riportata nel riquadro di codice 5.15.1, si è omesso il simbolo "&" avanti al primo parametro. Questa modifica si rende necessaria in quanto la variabile di remapping è stata dichiarata come puntatore. Inoltre la quantità dei byte da scrivere è stata modificata per ottenere lo stesso valore in modo più elegante.

Secondo quanto riportato nel paragrafo 5.15.2, la funzione GetNwkActiveKeyNumber può essere ridefinita come nel frammento di codice 5.15.8.

```
#define GetNwkActiveKeyNumber(x)
NVMRead((BYTE*)nwkActiveKeyNumber, (BYTE*)x, sizeof(BYTE));
```

Cod. 5.15.8: Definizione aggiornata della funzione GetNwkActiveKeyNumber.

Rispetto alla definizione originale, riportata nel riquadro di codice 5.15.3 si è variato il tipo del cast del secondo parametro da void a BYTE e,

analogamente alla funzione PutNwkActiveKeyNumber, si è introdotto un modo più elegante per indicare la quantità dei byte da leggere.

Gli aggiornamenti non comportano alcuna modifica al prototipo delle chiamate, pertanto nel codice dello stack non è necessario effettuare alcuna ulteriore modifica.

5.15.4 - Codice di esempio

Al fine di testare la validità delle ipotesi e delle considerazioni trattate nei paragrafi 5.15.1, 5.15.2 e 5.15.3, si amplia il progetto creato appositamente (introdotto nel paragrafo 5.5.4) con i frammenti di codice presentati nei paragrafi sopracitati e con una parte di codice, inserita nella funzione main, che utilizza le funzioni di storage relative al numero della chiave di rete.

Tale codice è visibile nel frammento 5.15.9, nel quale sono state omesse le parti relative alle informazioni trattate precedentemente e contiene: una parte di dichiarazione delle variabili; la scrittura dell'indirizzo in memoria di programma; la lettura di questo dalla memoria FLASH.

```
int main(){
    // DICHIARAZIONE DELLE VARIABILI
    BYTE temp_byte;

    // --- Scrittura in memoria di programma.
    temp_byte = 0xCF;
    PutNwkActiveKeyNumber(&temp_byte);

    // --- Lettura dalla memoria di programma.
    temp_byte = 0;
    GetNwkActiveKeyNumber(&temp_byte);

    return 0;
}
```

Cod. 5.15.9: Parte integrante nella funzione main per testare le funzioni di storage della chiave di rete.

In tale codice si dichiara una sola variabile: *temp_byte* di tipo BYTE, inizializzandola al valore dell'indirizzo da salvare.

Il salvataggio in memoria di programma comporta la scrittura delle locazioni della FLASH come visibile in Fig. 5.15.1: il valore è

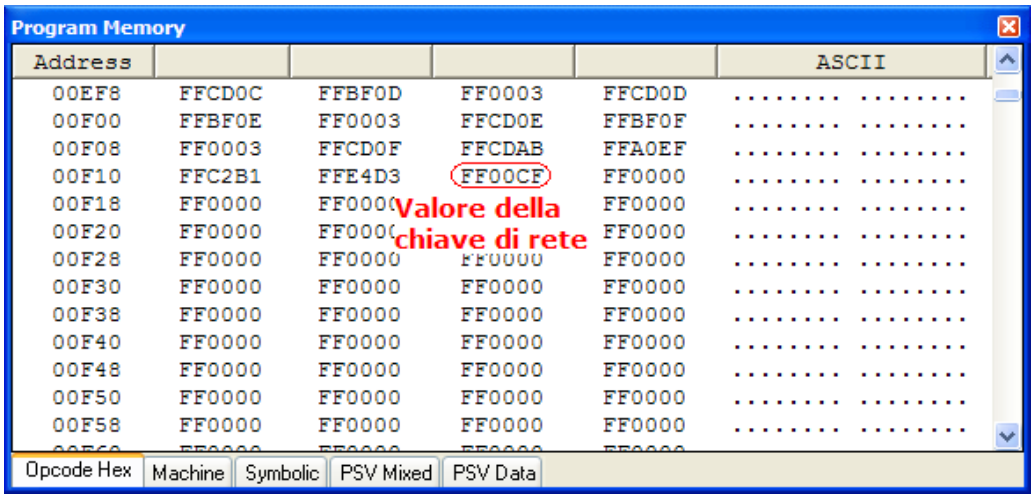

Fig. 5.15.1: Valore della chiave di rete salvato nella memoria di programma.

memorizzato all'indirizzo esadecimale 0xF14, poiché questa è la prima locazione libera dopo la memorizzazione del Trust Center Address. Tale allineamento, come osservato nel paragrafo 5.15.3, è ottenuto grazie alla gestione degli indirizzamenti tramite offset e dimensioni relativi ad ogni informazione da salvare in memoria. Confrontando i valori della figura con quelli della variabile *temp_byte* nel codice 5.15.9, si può affermare che la scrittura avviene correttamente.

La funzione di lettura ha il compito di leggere i dati dalla FLASH e di trasferirli in RAM. Per mezzo del simulatore MPLAB SIM, integrato nell'ambiente di sviluppo, è possibile osservare l'aggiornamento della variabile *temp_byte*, la quale a lettura ultimata assume il valore visibile in Fig. 5.15.2, in accordo con il valore salvato, visibile in Fig. 5.15.1 e con quello impostato nel codice.

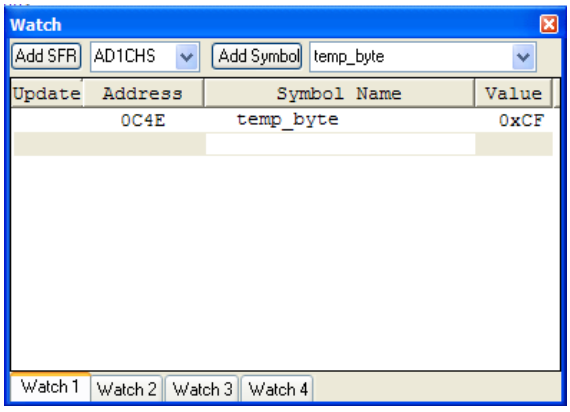

Fig. 5.15.2: Risultati prodotti dal simulatore.

5.16 – Esplorazione di Network Key Info

In seguito a quanto riportato nella sezione 5.4 – Utilizzo delle funzioni di storage, si procede all'esplorazione delle funzioni PutNwkKeyInfo(x, y) e GetNwkKeyInfo(x, y) definite nel file zNVM.h alle righe 327 e 326 rispettivamente.

Al fine di apprendere il funzionamento delle funzioni di storage, risulta utile esaminare alcune definizioni di tipo di dato, iniziando da quelle più semplici e procedendo verso quelle più complesse.

```
typedef struct _KEY_VAL{
    BYTE v[16];
} KEY_VAL;
```

Cod. 5.16.1: Definizione del tipo di dato KEY_VAL.

Il tipo di dato KEY_VAL, definito come nel frammento di codice 5.16.1, consiste in una struttura adatta a contenere il valore di una chiave di rete. L'unico campo della struttura è l'array *v* di sedici elementi di tipo BYTE. Si evince che il valore di una chiave di rete occupa in memoria uno spazio di 16 byte.

```
typedef struct _NETWORK_KEY_INFO{
      KEY_VAL     NetKey;
      WORD_VAL    SeqNumber;
} NETWORK_KEY_INFO;
```

Cod. 5.16.2: Definizione del tipo NETWORK_KEY_INFO.

Il tipo di dato NETWORK_KEY_INFO, definito come nel frammento di codice 5.16.2, consiste in una struttura adatta a contenere le informazioni sulla chiave di rete. Dalla sua definizione è possibile constatare che lo spazio in memoria occupato da una variabile di questo tipo è di 18 byte.

5.16.1 – Funzione PutNwkKeyInfo

La funzione PutNwkKeyInfo, definita originariamente come nel frammento di codice 5.16.3 (tratto dall'elenco di Cod. 5.4.1), si occupa

della memorizzazione delle informazioni di una chiave di rete all'interno della memoria di programma del microcontrollore.

```
#define PutNwkKeyInfo(x, y)
NVMWrite((NVM_ADDR *)x, (BYTE *)y, sizeof(NETWORK_KEY_INFO))
```

Cod. 5.16.3: Definizione originale della funzione PutNwkKeyInfo.

Osservando il codice del frammento 5.16.3 si nota che i parametri da passare come argomento alla chiamata della funzione sono: la locazione in memoria di programma nella quale salvare le informazioni e una variabile di tipo NETWORK_KEY_INFO, contenente i dati da memorizzare. Ovvero le chiamate a PutNwkKeyInfo sono tutte del tipo mostrato nel frammento di codice 5.16.4.

```
PutNwkKeyInfo(&networkKeyInfo, &currentNetworkKeyInfo);
```

Cod. 5.16.4: Prototipo delle chiamate a PutNwkKeyInfo.

In questo codice la variabile *currentNetworkKeyInfo*, di tipo NETWORK_KEY_INFO, è utilizzata come cursore e contiene il valore delle informazioni relative a una chiave di rete che devono essere salvate in memoria. La variabile *networkKeyInfo* è l'array di remapping, ma nella dichiarazione originale è stato commesso l'errore di non specificare l'attributo *space(prog)* e pertanto è allocato in RAM anziché in memoria di programma, come visibile nel frammento di codice 5.16.5.

```
extern ROM NETWORK_KEY_INFO networkKeyInfo[NUM_NWK_KEYS];
```

Cod. 5.16.5: Dichiarazione originale dell'array di remapping.

La costante NUM_NWK_KEYS rappresenta il numero di chiavi di rete di cui salvare le informazioni, il suo valore è 2.

Si evince dunque che lo spazio totale richiesto per salvare tutte le informazioni relative alle chiavi è 36 byte, ottenuto moltiplicando la dimensione di una singola chiave – di 18 byte come visto nel paragrafo 5.16 – per il valore di NUM_NWK_KEYS. È pertanto possibile

procedere alle definizioni dell'offset su NVMBuffer, della dimensione e della variabile di remapping al fine di ottenere i riferimenti alla memoria di programma, analogamente a quanto già visto per le informazioni trattate nei paragrafi precedenti. Tali definizioni sono scritte nel file NVM.h e riportate nel frammento di codice 5.16.6.

```
/** Numero di chiavi di rete di cui salvare le informazioni in
 *  memoria di programma.
 */
#define NUM_NWK_KEYS 2

/** nwkKeyInfoOffset indica quanti byte intercorrono tra la
 *  locazione iniziale di NVMBuffer e la locazione dove ha inizio
 *  la memorizzazione delle informazioni delle chiavi di rete.
 */
#define nwkKeyInfoOffset      nwkActiveKeyNumberOffset +
                              nwkActiveKeyNumberMapSize

/** Dimensione in BYTE dello spazio da allocare in memoria di
 *  programma riservato alle informazioni sulle chiavi di rete.
 */
#define nwkKeyInfoMapSize sizeof(NETWORK_KEY_INFO) * NUM_NWK_KEYS

/** Definizione dell'array networkKeyInfo. L'elemento i-esimo di
 *  questo  array contiene un puntatore alla locazione in memoria
 *  di programma nella quale devono essere salvate le informazioni
 *  relative alla i-esima chiave di rete.
 */
extern ROM BYTE * networkKeyInfo[NUM_NWK_KEYS];
```

Cod. 5.16.6: Definizioni nel file NVM.h relative alle informazioni sulle chiavi di rete.

Si osservi che la definizione dell'offset tiene conto dello stesso offset e della dimensione del numero delle chiavi di rete. Nel file NVM.c è necessario dichiarare la variabile di remapping, *networkKeyInfo*, già definita nel file di intestazione, inizializzandola correttamente alle opportune locazioni di NVMBuffer, come visibile nel frammento di codice 5.16.7.

```
ROM BYTE * networkKeyInfo[NUM_NWK_KEYS] = {
      NVMBuffer + nwkKeyInfoOffset + 0*sizeof(NETWORK_KEY_INFO),
      NVMBuffer + nwkKeyInfoOffset + 1*sizeof(NETWORK_KEY_INFO)
};
```

Cod. 5.16.7: Dichiarazione e inizializzazione della variabile di remapping delle chiavi di rete.

L'inizializzazione fa uso della definizione del relativo offset: in tal modo *networkKeyInfo* punta alla prima locazione libera dopo il numero della chiave di rete attiva.

5.16.2 - Funzione GetNwkKeyInfo

La funzione GetNwkKeyInfo si occupa della lettura dei record delle informazioni relative alla chiave di rete dalla memoria di programma del microcontrollore; originariamente è definita come visibile nel frammento di codice 5.16.8, tratto dall'elenco di Cod. 5.4.2.

```
#define GetNwkKeyInfo(x, y)
NVMRead((BYTE *)x,(ROM void *)y, sizeof(NETWORK_KEY_INFO))
```

Cod. 5.16.8: Definizione originale della funzione GetNwkKeyInfo.

Analogamente alla funzione di memorizzazione, accetta due parametri: il primo contiene il riferimento alla locazione in memoria di programma nella quale si trova il record da leggere; il secondo contiene il riferimento alla locazione in RAM nella quale deve essere trasferito il dato letto; ovvero il prototipo delle chiamate a GetNwkKeyInfo è quello visibile nel frammento di codice 5.16.9, dove la variabile *currentNetworkKeyInfo* è di tipo NETWORK_KEY_INFO, come visto nel paragrafo 5.16.1.

```
GetNwkKeyInfo(&currentNetworkKeyInfo, &networkKeyInfo[i]);
```

Cod. 5.16.9: Prototipo delle chiamate a GetBindingRecord.

5.16.3 - Considerazioni

Nei precedenti paragrafi sono state presentate le funzioni di salvataggio e di lettura delle informazioni relative alle chiavi di rete verso e dalla memoria di programma; in particolare si è visto il loro comportamento ed i prototipi alle relative chiamate. Nel paragrafo 5.16.1 inoltre, sono stati introdotti: offset, dimensioni e variabile di remapping per

permettere la corretta gestione della memoria di programma, al fine di riservarne lo spazio necessario ai vari record. In questa sezione, di tali funzioni saranno presentati gli aggiornamenti delle definizioni.

Secondo quanto visto nel paragrafo 5.16.1, la funzione PutNetworkKeyInfo può essere ridefinita come nel frammento di codice 5.16.10.

```
#define PutNwkKeyInfo(x, y)
NVMWrite((BYTE*)x, (BYTE*)y, sizeof(NETWORK_KEY_INFO))
```

Cod. 5.16.10: Definizione aggiornata della funzione PutNwkKeyInfo.

Rispetto alla definizione originale, riportata nel riquadro di codice 5.16.3 non è stata apportata alcuna modifica.

Secondo quanto riportato nel paragrafo 5.16.2, la funzione GetNwkKeyInfo può essere ridefinita come nel frammento di codice 5.16.11.

```
#define GetNwkKeyInfo(x, y)
NVMRead((BYTE*)y, (BYTE*)x, sizeof(NETWORK_KEY_INFO))
```

Cod. 5.16.11: Definizione aggiornata della funzione GetNwkKeyInfo.

Rispetto alla definizione originale, riportata nel riquadro di codice 5.16.8, si è mutato l'ordine dei primi due argomenti (questo cambiamento è la conseguenza dell'inversione dei parametri nella funzione NVMRead, aggiornata nel paragrafo 5.3.3) e il tipo del cast del secondo parametro da void a BYTE.

Le definizioni appena introdotte altro non sono che una mappatura delle funzioni NVMWrite ed NVMRead, e sono tutte inserite nel file NVM.h, insieme alle definizioni di offset, dimensioni e variabile di remapping.

5.16.4 - Codice di esempio

Al fine di testare la validità delle ipotesi e delle considerazioni trattate

nei paragrafi 5.16.1, 5.16.2 e 5.16.3, si amplia il progetto creato appositamente (introdotto nel paragrafo 5.5.4) con i frammenti di codice presentati nei paragrafi sopracitati e con una parte di codice, inserita nella funzione main, che utilizza le funzioni di storage relative alle informazioni sulle chiavi di rete.

Quest'ultima è visibile nel frammento di codice 5.16.12, nel quale sono state omesse le parti relative alle informazioni trattate precedentemente e contiene: una parte di dichiarazione delle variabili; una parte di inizializzazione in RAM dei dati da scrivere nella memoria di programma; la scrittura delle informazioni; la lettura di queste dalla memoria FLASH e relativo trasferimento del valore in RAM.

Nel codice si dichiara una variabile: *currentNetworkKeyInfo* di tipo NETWORK_KEY_INFO contenente tutte le informazioni da salvare in memoria di programma.

Il salvataggio in memoria di programma comporta la scrittura delle locazioni della FLASH come visibile in Fig. 5.16.2: i valori sono memorizzati a partire dall'indirizzo esadecimale 0xF16, poiché questa è la prima locazione libera dopo la memorizzazione del numero della chiave di rete attiva. Tale allineamento si riesce ad ottenere grazie alla gestione degli indirizzamenti tramite offset e dimensioni per ogni informazione da salvare in memoria. In figura sono evidenziati in giallo i valori relativi alle informazioni sulla prima chiave di rete: le prime otto parole di istruzione contengono i valori del campo *NetKey*; la nona contiene il valore del campo *SeqNumber;* come si può vedere dalla figura coincidono con quelli dei campi impostati nel

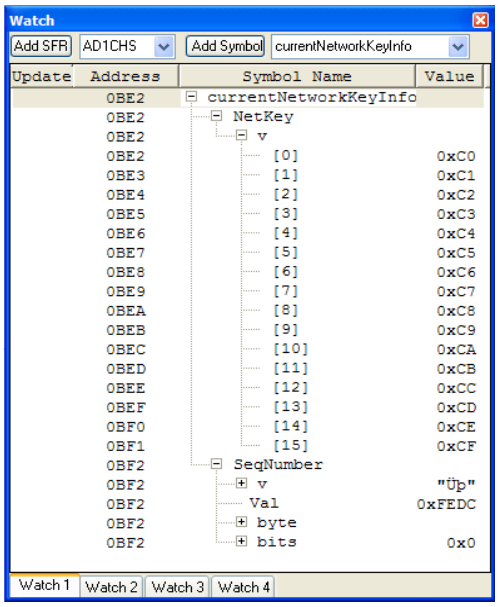

Fig. 5.16.1: Risultati della simulazione.

codice 5.16.12.

```
int main(){
    NETWORK_KEY_INFO currentNetworkKeyInfo;

    // --- Inizializzazione dei valori in RAM:
    currentNetworkKeyInfo.NetKey.v[0] = 0xB0;
    currentNetworkKeyInfo.NetKey.v[1] = 0xB1;
    currentNetworkKeyInfo.NetKey.v[2] = 0xB2;
    currentNetworkKeyInfo.NetKey.v[3] = 0xB3;
    currentNetworkKeyInfo.NetKey.v[4] = 0xB4;
    currentNetworkKeyInfo.NetKey.v[5] = 0xB5;
    currentNetworkKeyInfo.NetKey.v[6] = 0xB6;
    currentNetworkKeyInfo.NetKey.v[7] = 0xB7;
    currentNetworkKeyInfo.NetKey.v[8] = 0xB8;
    currentNetworkKeyInfo.NetKey.v[9] = 0xB9;
    currentNetworkKeyInfo.NetKey.v[10] = 0xBA;
    currentNetworkKeyInfo.NetKey.v[11] = 0xBB;
    currentNetworkKeyInfo.NetKey.v[12] = 0xBC;
    currentNetworkKeyInfo.NetKey.v[13] = 0xBD;
    currentNetworkKeyInfo.NetKey.v[14] = 0xBE;
    currentNetworkKeyInfo.NetKey.v[15] = 0xBF;
    currentNetworkKeyInfo.SeqNumber.Val = 0xCDEF;
    currentNetworkKeyInfo.NetKey.v[0] = 0xC0;
    currentNetworkKeyInfo.NetKey.v[1] = 0xC1;
    currentNetworkKeyInfo.NetKey.v[2] = 0xC2;
    currentNetworkKeyInfo.NetKey.v[3] = 0xC3;
    currentNetworkKeyInfo.NetKey.v[4] = 0xC4;
    currentNetworkKeyInfo.NetKey.v[5] = 0xC5;
    currentNetworkKeyInfo.NetKey.v[6] = 0xC6;
    currentNetworkKeyInfo.NetKey.v[7] = 0xC7;
    currentNetworkKeyInfo.NetKey.v[8] = 0xC8;
    currentNetworkKeyInfo.NetKey.v[9] = 0xC9;
    currentNetworkKeyInfo.NetKey.v[10] = 0xCA;
    currentNetworkKeyInfo.NetKey.v[11] = 0xCB;
    currentNetworkKeyInfo.NetKey.v[12] = 0xCC;
    currentNetworkKeyInfo.NetKey.v[13] = 0xCD;
    currentNetworkKeyInfo.NetKey.v[14] = 0xCE;
    currentNetworkKeyInfo.NetKey.v[15] = 0xCF;
    currentNetworkKeyInfo.SeqNumber.Val = 0xFEDC;
    // --- Scrittura in memoria di programma.
    PutNwkKeyInfo(networkKeyInfo[0], &currentNetworkKeyInfo);
    PutNwkKeyInfo(networkKeyInfo[1], &currentNetworkKeyInfo);
    // --- Lettura dalla memoria di programma.
    GetNwkKeyInfo(&currentNetworkKeyInfo, networkKeyInfo[0]);
    GetNwkKeyInfo(&currentNetworkKeyInfo, networkKeyInfo[1]);

    return 0;
}
```

Cod. 5.16.12: Parte integrante nella funzione main per testare le funzioni delle chiavi di rete.

La funzione di lettura ha il compito di leggere i dati dalla FLASH e di trasferirli in RAM. Per mezzo del simulatore MPLAB SIM integrato nell'ambiente di sviluppo, è possibile osservare l'aggiornamento della variabile *currentNetworkKeyInfo*, la quale a lettura ultimata della seconda chiave assume il valore visibile in Fig. 5.16.1, in accordo con i valori salvati, visibili in Fig. 5.16.2 e con quelli impostati nel codice 5.16.12.

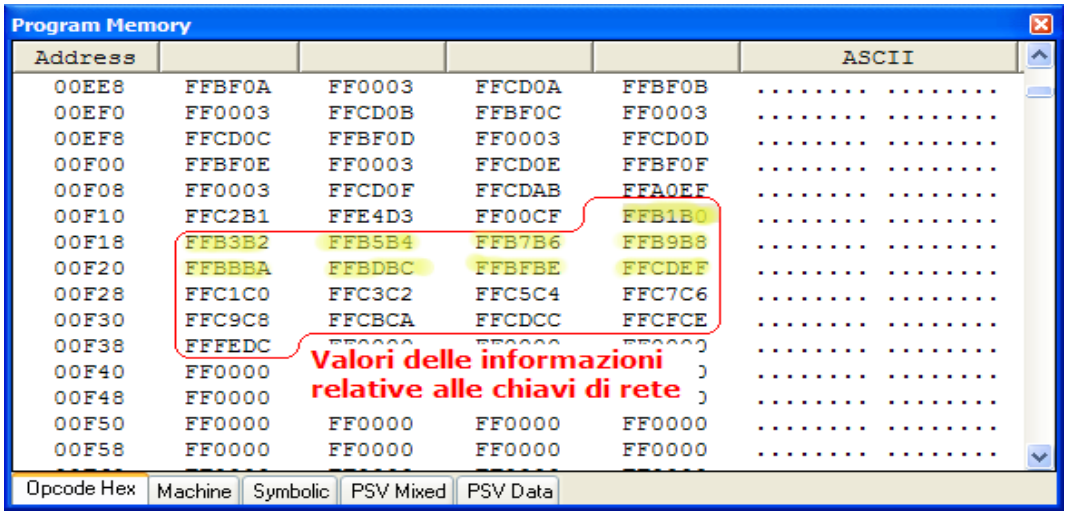

Fig. 5.16.2: Valori dei record di binding salvati nella memoria di programma.

5.17 – Esplorazione di GroupAddress

In seguito a quanto riportato nella sezione 5.4 – Utilizzo delle funzioni di storage, si procede all'esplorazione delle funzioni PutGroupAddress(x, y) e GetGroupAddress(x, y) definite nel file zNVM.h alle righe 330 e 329 rispettivamente.

Al fine di apprendere il funzionamento delle funzioni di storage, risulta utile esaminare la definizione del tipo di dato GROUP_ADDRESS_RECORD, definito come nel frammento di codice 5.17.1, il quale consiste in una struttura di due campi: *GroupAddress* di tipo SHORT_ADDR e l'array EndPoint di 8 elementi di tipo BYTE.

Una variabile di questo tipo occupa in memoria uno spazio di 10 byte.

```
#define MAX_GROUP_END_POINT 8

typedef struct _GROUP_ADDRESS_RECORD
{
    SHORT_ADDR      GroupAddress;
    BYTE            EndPoint[MAX_GROUP_END_POINT];
} GROUP_ADDRESS_RECORD;
```

Cod. 5.17.1: Definizione del tipo di dato GROUP_ADDRESS_RECORD.

5.17.1 – Funzione PutGroupAddress

La funzione PutGroupAddress, definita originariamente come nel frammento di codice 5.17.2 (tratto dall'elenco di Cod. 5.4.1), si occupa della memorizzazione di un gruppo di indirizzi all'interno della memoria di programma del microcontrollore.

```
#define PutGroupAddress(x, y)
NVMWrite((NVM_ADDR *)x, (BYTE *)y, sizeof(GROUP_ADDRESS_RECORD))
```

Cod. 5.17.2: Definizione originale della funzione PutGroupAddress.

Osservandone la definizione si nota che i parametri da passare come argomento alla chiamata della funzione sono: la locazione in memoria di programma nella quale salvare il gruppo e una variabile di tipo GROUP_ADDRESS_RECORD, contenente i dati da memorizzare. Ovvero le chiamate a PutGroupAddress sono tutte del tipo mostrato nel frammento di codice 5.17.3.

```
PutGroupAddress(  pCurrentGroupAddressRecord,
                  &currentGroupAddressRecord);
```

Cod. 5.17.3: Prototipo delle chiamate a PutGroupAddress.

In questo codice la variabile *currentGroupAddressRecord*, di tipo GROUP_ADDRESS_RECORD, è utilizzata come cursore e contiene il valore delle informazioni relative a una chiave di rete che devono essere salvate in memoria. La variabile pC*urrentGroupAddressRecord* è un puntatore alla locazione di programma nella quale scrivere i dati,

ed è inizializzato ad un elemento dell'array di remapping. Tale array, *apsGroupAddressTable*, è dichiarato originariamente senza specificare l'attributo *space(prog)* e pertanto è allocato in RAM anziché in memoria di programma, come visibile nel frammento di codice 5.17.4.

```
ROM BYTE * apsGroupAddressTable[MAX_GROUP];
```

Cod. 5.17.4: Dichiarazione originale dell'array di remapping.

La costante rappresenta il numero di gruppi di indirizzi da salvare ed è impostato al valore 8.

```
/** Numero gruppi di indirizzi da salvare in memoria di programma.
*/
#define MAX_GROUP 8

/** groupAddressOffset indica quanti byte intercorrono tra la
 *  locazione iniziale di NVMBuffer e la locazione dove ha inizio
 *  la memorizzazione dei gruppi di indirizzi.
 */
#define groupAddressOffset nwkKeyInfoOffset + nwkKeyInfoMapSize

/** Dimensione in BYTE dello spazio da allocare in memoria di
 *  programma riservato ai gruppi di indirizzi.
 */
#define groupAddressMapSize   sizeof(GROUP_ADDRESS_RECORD) *
                              MAX_GROUP

/** Definizione dell'array apsGroupAddressTable. L'elemento
 *  i-esimo di questo array contiene un puntatore alla locazione
 *  in memoria di programma nella quale deve essere salvato l'i-
 *  esimo gruppo di indirizzi.
 */
extern ROM BYTE * apsGroupAddressTable[MAX_GROUP];
```

Cod. 5.17.5: Definizioni nel file NVM.h relative alle informazioni sulle chiavi di rete.

Si evince dunque che lo spazio totale richiesto per salvare tutti i gruppi di indirizzi è 80 byte, ottenuto moltiplicando la dimensione di un singolo gruppo – di 10 byte come visto nel paragrafo 5.17 - per il valore di MAX_GROUP. È pertanto possibile procedere alle definizioni dell'offset su NVMBuffer, della dimensione e della variabile di remapping al fine di ottenere i riferimenti alla memoria di programma, analogamente a quanto già visto per le informazioni trattate nei

paragrafi precedenti. Tali definizioni sono scritte nel file NVM.h e riportate nel frammento di codice 5.17.5.

Si osservi che la definizione dell'offset tiene conto dello stesso offset e delle dimensioni delle informazioni relative alle chiavi di rete. Nel file NVM.c è necessario dichiarare la variabile di remapping, *apsGroupAddressTable*, già definita nel file di intestazione, inizializzandola correttamente alle opportune locazioni di NVMBuffer, come visibile nel frammento di codice 5.17.6.

```
ROM BYTE * apsGroupAddressTable[MAX_GROUP] = {
  NVMBuffer + groupAddressOffset + 0*sizeof(GROUP_ADDRESS_RECORD),
  NVMBuffer + groupAddressOffset + 1*sizeof(GROUP_ADDRESS_RECORD),
  NVMBuffer + groupAddressOffset + 2*sizeof(GROUP_ADDRESS_RECORD),
  NVMBuffer + groupAddressOffset + 3*sizeof(GROUP_ADDRESS_RECORD),
  NVMBuffer + groupAddressOffset + 4*sizeof(GROUP_ADDRESS_RECORD),
  NVMBuffer + groupAddressOffset + 5*sizeof(GROUP_ADDRESS_RECORD),
  NVMBuffer + groupAddressOffset + 6*sizeof(GROUP_ADDRESS_RECORD),
  NVMBuffer + groupAddressOffset + 7*sizeof(GROUP_ADDRESS_RECORD)
};
```

Cod. 5.17.6: Dichiarazione e inizializzazione della variabile di remapping dei gruppi di indirizzi.

L'inizializzazione fa uso della definizione del relativo offset: in tal modo *apsGroupAddressTable* punta alla prima locazione libera dopo le informazioni relative alle chiavi di rete.

5.17.2 - Funzione GetGroupAddress

La funzione GetGroupAddress si occupa della lettura dei gruppi di indirizzi dalla memoria di programma del microcontrollore; originariamente è definita come visibile nel frammento di codice 5.17.7, tratto dall'elenco di Cod. 5.4.2.

```
#define GetGroupAddress(x, y)
NVMRead((BYTE *)x, (ROM void *)y, sizeof(GROUP_ADDRESS_RECORD))
```

Cod. 5.17.7: Definizione originale della funzione GetNwkKeyInfo.

Analogamente alla funzione di memorizzazione, accetta due parametri: il primo contiene il riferimento alla locazione in memoria di programma

nella quale si trova il record da leggere; il secondo contiene il riferimento alla locazione in RAM nella quale deve essere trasferito il dato letto; ovvero il prototipo delle chiamate a GetNwkKeyInfo è quello visibile nel frammento di codice 5.17.8, dove la variabile *currentGroupAddress* è di tipo GROUP_ADDRESS_RECORD, come visto nel paragrafo 5.17.1.

```
GetGroupAddress(  &currentGroupAddressRecord,
                  pCurrentGroupAddressRecord);
```

Cod. 5.17.8: Prototipo delle chiamate a GetBindingRecord.

5.17.3 – Considerazioni

Nei precedenti paragrafi sono state presentate le funzioni di salvataggio e di lettura dei gruppi di indirizzi verso e dalla memoria di programma; in particolare si è visto il loro comportamento ed i prototipi alle relative chiamate. Nel paragrafo 5.17.1 inoltre, sono stati introdotti: offset, dimensioni e variabile di remapping per permettere la corretta gestione della memoria di programma, al fine di riservarne lo spazio necessario ai vari gruppi. In questa sezione, di tali funzioni saranno presentati gli aggiornamenti delle definizioni.

Secondo quanto visto nel paragrafo 5.17.1, la funzione PutGroupAddress può essere ridefinita come nel frammento di codice 5.17.9.

```
#define PutGroupAddress(x, y)
NVMWrite((BYTE*)x, (BYTE*)y, sizeof(GROUP_ADDRESS_RECORD))
```

Cod. 5.17.9: Definizione aggiornata della funzione PutGroupAddress.

Rispetto alla definizione originale, riportata nel riquadro di codice 5.17.2 si è variato il tipo del cast del primo parametro da NVM_ADDR a BYTE.

Secondo quanto riportato nel paragrafo 5.17.2, la funzione GetGroupAddress può essere ridefinita come nel frammento di codice 5.17.10.

```
#define GetGroupAddress(x, y)
NVMRead((BYTE*)y, (BYTE*)x, sizeof(GROUP_ADDRESS_RECORD))
```

Cod. 5.17.10: Definizione aggiornata della funzione GetGroupAddress.

Rispetto alla definizione originale, riportata nel riquadro di codice 5.17.7, si è mutato l'ordine dei primi due argomenti (questo cambiamento è la conseguenza dell'inversione dei parametri nella funzione NVMRead, aggiornata nel paragrafo 5.3.3) e il tipo del cast del secondo parametro da void a BYTE.

Le definizioni appena introdotte altro non sono che una mappatura delle funzioni NVMWrite ed NVMRead, e sono tutte inserite nel file NVM.h, insieme alle definizioni di offset, dimensioni e variabile di remapping.

5.17.4 - Codice di esempio

Al fine di testare la validità delle ipotesi e delle considerazioni trattate nei paragrafi 5.17.1, 5.17.2 e 5.17.3, si amplia il progetto creato appositamente (introdotto nel paragrafo 5.5.4) con i frammenti di codice presentati nei paragrafi sopracitati e con una parte di codice, inserita nella funzione main, che utilizza le funzioni di storage relative ai gruppi di indirizzi.

Quest'ultima è visibile nel frammento di codice 5.17.11, nel quale sono state omesse le parti relative alle informazioni trattate precedentemente e contiene: una parte di dichiarazione delle variabili; una parte di inizializzazione in RAM dei dati da scrivere nella memoria di programma; la scrittura dei gruppi di indirizzi; la lettura di questi dalla memoria FLASH e relativo trasferimento del valore in RAM.

Nel codice si dichiarano tre variabili: l'array *groupAddressTable* contenente tutti i gruppi di indirizzi da salvare in memoria; *currentGroupAddressRecord* di tipo GROUP_ADDRESS_RECORD utilizzata come cursore tra i gruppi da scrivere; *pCurrentGroupAddressRecord*, un

puntatore alla locazione in memoria di programma nella quale memorizzare il gruppo di indirizzo corrente, tale puntatore è inizializzato prima di ogni scrittura ad un elemento dell'array di remapping.

```
int main(){
    // DICHIARAIZIONE DELLE VARIBILI
    GROUP_ADDRESS_RECORD groupAddressTable[MAX_GROUP];
    GROUP_ADDRESS_RECORD currentGroupAddressRecord;
    ROM BYTE * pCurrentGroupAddressRecord;

    // --- Inizializzazione dei valori in RAM:
    for(i=0; i<MAX_GROUP; i++){
        groupAddressTable[i].GroupAddress.Val = 0xEB00 + i;
        groupAddressTable[i].EndPoint[0] = 0x20 + i;
        groupAddressTable[i].EndPoint[1] = 0x30 + i;
        groupAddressTable[i].EndPoint[2] = 0x40 + i;
        groupAddressTable[i].EndPoint[3] = 0x50 + i;
        groupAddressTable[i].EndPoint[4] = 0x60 + i;
        groupAddressTable[i].EndPoint[5] = 0x70 + i;
        groupAddressTable[i].EndPoint[6] = 0x80 + i;
        groupAddressTable[i].EndPoint[7] = 0x90 + i;
    }

    // --- Scrittura in memoria di programma.
    for(i=0; i<MAX_GROUP; i++){
        pCurrentGroupAddressRecord = apsGroupAddressTable[i];
        currentGroupAddressRecord = groupAddressTable[i];
        PutGroupAddress(pCurrentGroupAddressRecord,
                        &currentGroupAddressRecord);
    }

    // --- Lettura dalla memoria di programma.
    for(i=0; i<MAX_GROUP; i++){
        pCurrentGroupAddressRecord = apsGroupAddressTable[i];
        GetGroupAddress(&currentGroupAddressRecord,
        pCurrentGroupAddressRecord);
    }

    return 0;
}
```

Cod. 5.17.11: Parte integrante nella funzione main per testare le funzioni dei gruppi di indirizzi.

Il salvataggio in memoria di programma comporta la scrittura delle locazioni della FLASH come visibile in Fig. 5.17.1: i valori sono memorizzati a partire dall'indirizzo esadecimale 0xF3A, poiché questa è la prima locazione libera dopo la memorizzazione delle informazioni

relative alle chiavi di rete. Tale allineamento si riesce ad ottenere grazie alla gestione degli indirizzamenti tramite offset e dimensioni per ogni informazione da salvare in memoria. In figura sono evidenziati in giallo i valori relativi al primo gruppo: la prima parola di istruzione contiene i valori del campo *GroupAddressVal*; le successive contengono i valori del campo *EndPoint*; come si può vedere dalla figura tali valori coincidono con quelli impostati nel codice 5.17.11.

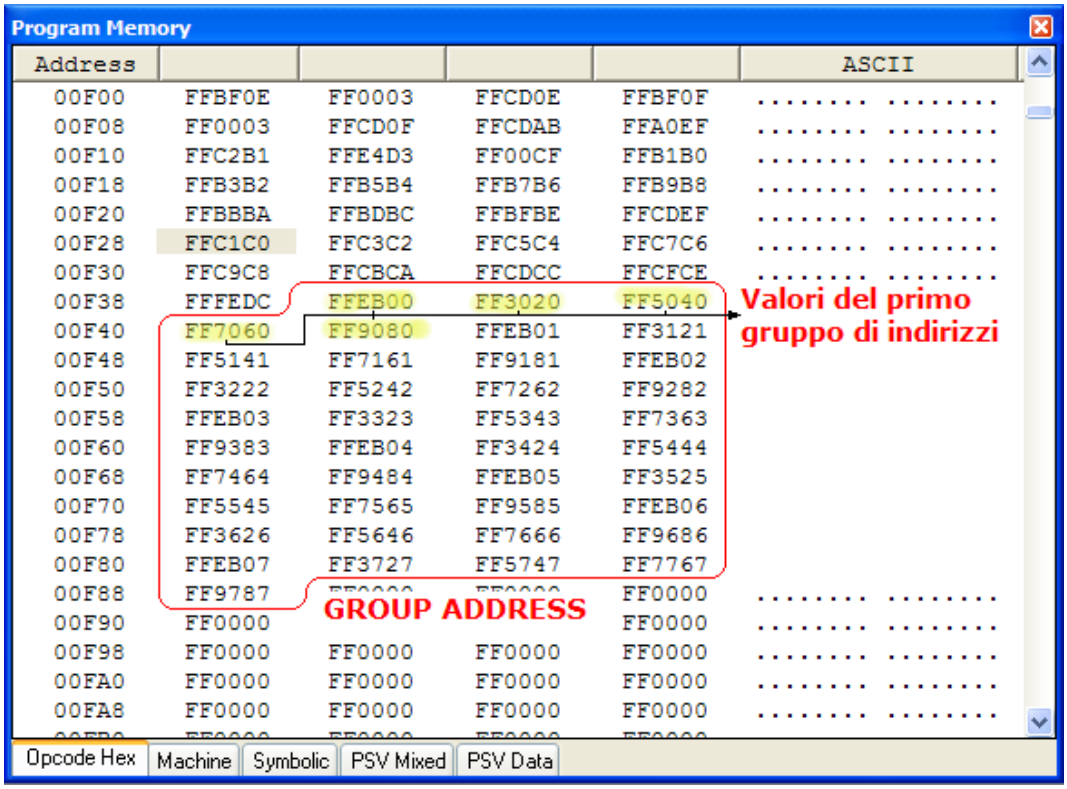

Fig. 5.17.1: Valori dei record dei gruppi di indirizzi salvati nella memoria di programma.

La funzione di lettura ha il compito di leggere i dati dalla FLASH e di trasferirli in RAM. Per mezzo del simulatore MPLAB SIM integrato nell'ambiente di sviluppo, è possibile osservare l'aggiornamento della variabile *currentGroupAddress*, la quale a lettura ultimata assume il valore visibile in Fig. 5.17.2, in accordo con i valori salvati, visibili in Fig. 5.17.1 e con quelli impostati nel codice 5.17.11.

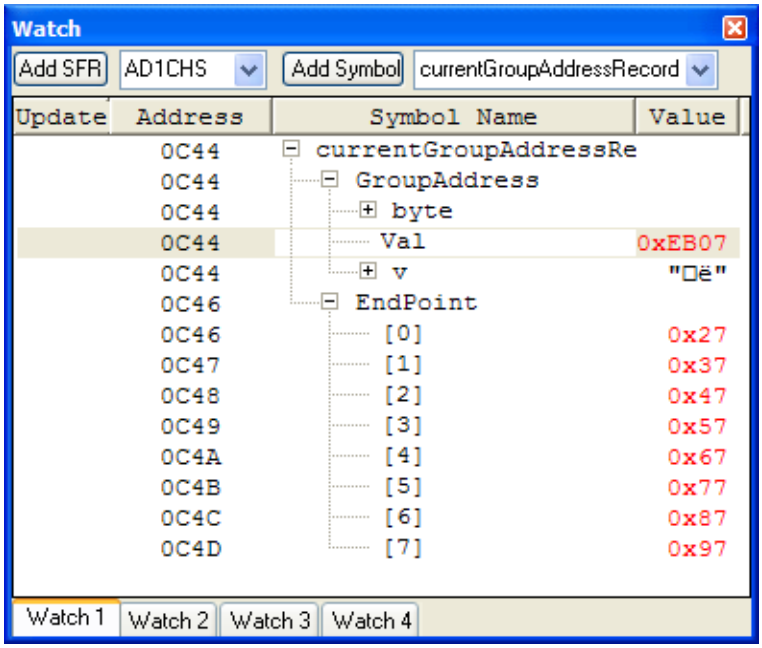

Fig. 5.17.2: Risultati prodotti dalla simulazione.

Cap. 6: Conclusioni

Il lavoro svolto in questa tesi è stato rivolto alla risoluzione di alcuni problemi riguardanti l'integrazione dello stack TCP/IP e dello stack ZigBee nella stessa unità a microcontrollore.

Nella fase iniziale del lavoro sono stati individuati i possibili problemi, in particolare si è riscontrato che il problema principale per la coesistenza dei due stack protocollari è causato da un conflitto relativo all'utilizzo del bus SPI da parte delle risorse fisiche connesse ai due diversi stack.

Per risolvere questo conflitto è necessaria una diversa ripartizione delle risorse hardware, in particolare della memoria esterna utilizzata da entrambi gli stack protocollari. In questo modo sarà possibile in un secondo passo, che costituirà la naturale prosecuzione di questa tesi, separare fisicamente le connessioni dei moduli di comunicazione relativi alla PicTail Ethernet e alla daughter card picdemZ al fine di destinare il bus SPI-1 alla prima, che funziona da master, ed il bus SPI-2 al secondo modulo e alla memoria EEPROM esterna, in quanto entrambe funzionano da slave.

Concludendo, il contributo principale apportato da questo lavoro di tesi concerne la migrazione dello stack ZigBee dalla memoria esterna alla memoria interna di programma, oltre alla fondamentale parte di diagnostica iniziale, ed il rilascio di un progetto aggiornato da integrare a quello principale, già sviluppato in precedenza, relativo alla creazione di un nodo NCAP della rete di sensori.

Ringraziamenti

Grazie ai miei genitori, Giovanni e Maria, perché mi permettono di vivere più che dignitosamente e perché costituiscono un importante sostegno, anche morale, nei momenti di difficoltà.

Grazie a mia sorella, Angela, per la sua allegria ma soprattutto per la sua generosità e per il suo affetto, che non mi è mai mancato.

Grazie a Rossella, perché mi sopporta e mi supporta, per il suo affetto, i suoi sorrisi, il suo rispetto, la sua lealtà e per tutte le sue infinite doti.

Grazie ai miei amici, recenti e di infanzia, per tutte le esperienze condivise e per il loro supporto.

Grazie a Augusta, Bruno, Antonella, Massimo, Paolo, Irene, Monica, Mauro, Maria, Walter per la loro calorosa accoglienza e condivisione di momenti allegri.

Grazie a Giovanni e Lorenzo perché la convivenza mi ha reso più maturo e responsabile.

Grazie ai colleghi con i quali ho condiviso l'esperienza di laboratorio nella preparazione di questo lavoro di tesi, in particolare a Pierpaolo per il suo fondamentale aiuto e ad Alberto per i suoi preziosi consigli.

Grazie alla Prof.ssa Ing. Giada Giorgi per avermi accolto in questo gruppo e avermi permesso di partecipare alla

realizzazione di un progetto per me molto interessante.

Grazie a tutti coloro i quali mi sono stati vicino in questi anni per il raggiungimento di questo traguardo e con i quali posso condividere questa gioia.