Università degli studi di Catania

Progetto di Reti per L'automazione Industriale

anno accademico 2012/ 2013

Laura Pafumi
Marco Rapisarda

Abract

Il lavoro oggetto di questa relazione è stato svolto in due fasi:

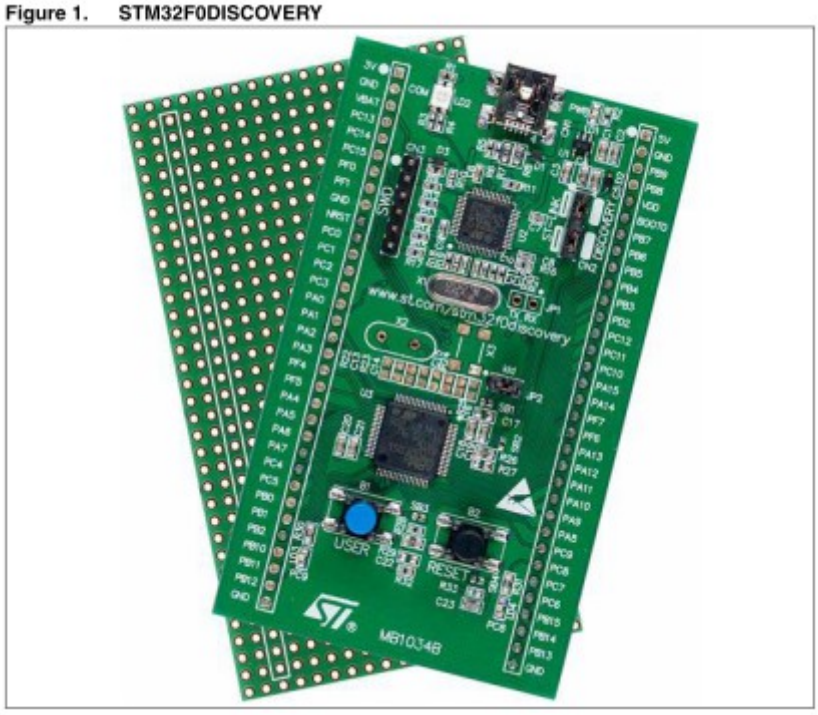
1) *Test e caratterizzazione dei dispositivi utilizzati (in particolare dei transceiver nRF24L01)*
2) *Implementazione di un protocollo di comunicazione di livello MAC di tipo CSMA.*

1 Dispositivi e strumenti utilizzati

Sono stati utilizzati:

- 4 stm32f0-discovery [1]
- 4 nRF24L01 [2]
- 1 XBee - USB Board 990002

Figure 1. STM32F0DISCOVERY



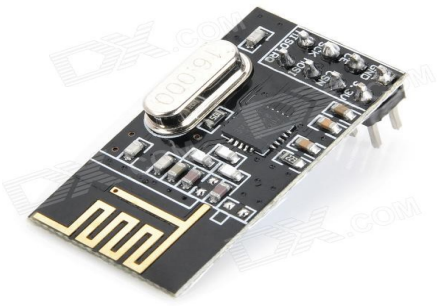


Figure 2: nRF24L01

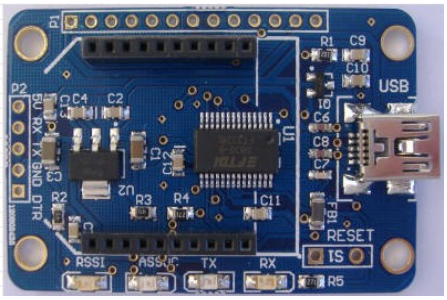


Figure 3: XBee - USB Board

1.1 Strumenti richiesti:

- Windows OS (XP, Vista, 7)
- cavetto USB tipo A to Mini-B USB

1.2 Ambiente di sviluppo/altri:

Come ambiente di sviluppo è stato scelto IAR Embedded Workbench [3].
Per poter leggere i dati tramite seriale abbiamo utilizzato il programma “Terminal” [4]

2 Il tranceiver nRF24l01

Il nRF24L01 è un ricetrasmettitore wireless a chip singolo a 2,4 GHz con embedded baseband protocol engine (Enhanced ShockBurst ™), progettato per operare a potenza ultra bassa. Il transceiver è progettato per il funzionamento nella banda di frequenza ISM compresa tra i 2.400 e i 2,4835 Ghz.

Il nRF24L01 viene configurato e gestito attraverso una Serial Peripheral Interface (SPI.) .
Una coda FIFO garantisce un flusso regolare di dati tra il microcontrollore e il tranceiver.
Il front-end radio utilizza la modulazione GFSK e i suoi parametri sono configurabili dall'utente (canale di frequenza, potenza di uscita e la velocità di trasmissione dati).

Il data rate è configurabile a 1 e 2Mbps. La combinazione dell'alta velocità di trasferimento dati con la modalità di alimentazione a risparmio energetico consente al transceiver di essere molto utile per progetti a basso costo energetico.

2.1 Enhanced ShockBurst ™

Enhanced ShockBurst™ utilizza ShockBurst ™ (che verrà discusso nel paragafo 2.1) per la gestione automatica dei pacchetti e del timer. Durante la trasmissione, ShockBurst ™ assembla il pacchetto nel trasmettitore per l'invio.
Durante la ricezione, ShockBurst ™ cerca costantemente un indirizzo valido nel segnale demodulato e, una volta trovato, processa il resto del pacchetto e lo convalida utilizzando CRC. Se il pacchetto è valido il payload viene spostato nella FIFO RX.

La gestione automatica delle transazioni del pacchetto in caso di auto-ack ed auto-ritrasmissioni abilitate, funziona nel modo seguente:

- L'utente avvia la transazione trasmettendo un pacchetto di dati dal PTX (trasmettitore) al PRX (ricevitore).
- Enhanced ShockBurst ™ imposta automaticamente la modalità di ricezione nel PTX al fine di aspettare il pacchetto di ACK.
- Se il pacchetto viene ricevuto dal PRX, Enhanced ShockBurst ™ assembla automaticamente e trasmette un pacchetto di riconoscimento (ACK) al PTX prima di tornare alla modalità di ricezione.
- Se il PTX non riceve il pacchetto di ACK entro un tempo prestabilito, il pacchetto originale verrà ritrasmesso e il PTX verrà impostato nuovamente nella modalità di ricezione per l'attesa del nuovo ACK.

La modalità Enhanced Shock Burst™ è altamente configurabile; è possibile configurare parametri come il numero massimo di ritrasmissione e il ritardo tra una ritrasmissione e la successiva.
Tutte le operazioni automatiche vengono fatte senza coinvolgimento del microcontrollore.

2.1.1. Enhanced Shockburst™ packet format

Il pacchetto contiene un campo di preambolo, un campo di indirizzo, un campo di controllo, il payload e un campo CRC (Figura 3).

- **Preambolo:** Il preambolo è una sequenza di bit utilizzato per rilevare i livelli di 0 e 1 nel ricevitore. Il preambolo è lungo un byte
  e può avere come valore 01010101 o 10101010 : se il primo bit è = 1 il preambolo verrà automaticamente impostato a 10101010 mentre se il primo bit è a 0, il preambolo verrà automaticamente impostata su 01010101.

- **Indirizzo:** Questo è l'indirizzo per il ricevitore. L' indirizzo assicura i pacchetti corretti vengano rilevati dal ricevitore. Il campo può essere configurato per essere 3, 4 o, 5 byte tramite il registro AW.

- **Campo di controllo:** Il campo di controllo contiene un campo di 6 bit contenente la lunghezza del payload, un campo PID di 2 bit (Packet Identity) e 1 bit utilizzato per il flag NO_ACK.

- **Payload:** il payload può avere lunghezza minima di 0 e massima di 32 byte

- **CRC:** Il CRC è il meccanismo di rilevamento di errore nel pacchetto. Esso può essere di 1 o 2 byte e viene calcolato sull'indirizzo, sul campo di controllo e sul payload.
  Il polinomio usato dal CRC ad 1 byte è X^8 + X^2 + X + 1, con 0xFF come valore iniziale.
  Il polinomio nel caso di impostazione a 2 byte è invece X^16 + X^12 + X^5 + 1, con 0xFFFF come valore iniziale.
  Se il CRC fallisce i pacchetti vengono rifiutati.

2.1.2 Gestione automatica dei pacchetti

Enhanced ShockBurst™ usa ShockBurst™ per la gestione automatica dei pacchetti. Le funzioni presenti sono:

- **Lunghezza statica e dinamica del payload:** L'Enhanced ShockBurst ™ offre due alternative per la gestione della lunghezza del payload.
  L'alternativa di default è la lunghezza di payload statico. In questo caso tutti i

pacchetti tra un trasmettitore e un ricevitore hanno la stessa lunghezza. Nel caso in cui, invece, si imposti una lunghezza dinamica del payload, è possibile inviare pacchetti con lunghezza differente. Ciò significa che per un sistema con diverse lunghezze dei pacchetti, non è necessario effettuare lo scale della lunghezza.

- **Assemblamento automatico del pacchetto:** preambolo, indirizzo, controllo del pacchetto, payload e CRC vengono assemblati in maniera automatica prima che il pacchetto venga trasmesso.

- **Validazione automatica del pacchetto**
- **Diassemblamento automatico del pacchetto**

3 HowTo: connessione e programmazione di trasmettitore e ricevitore (versione base)

3.1 Connessione tra stm32f0-discovery e nRF24L01

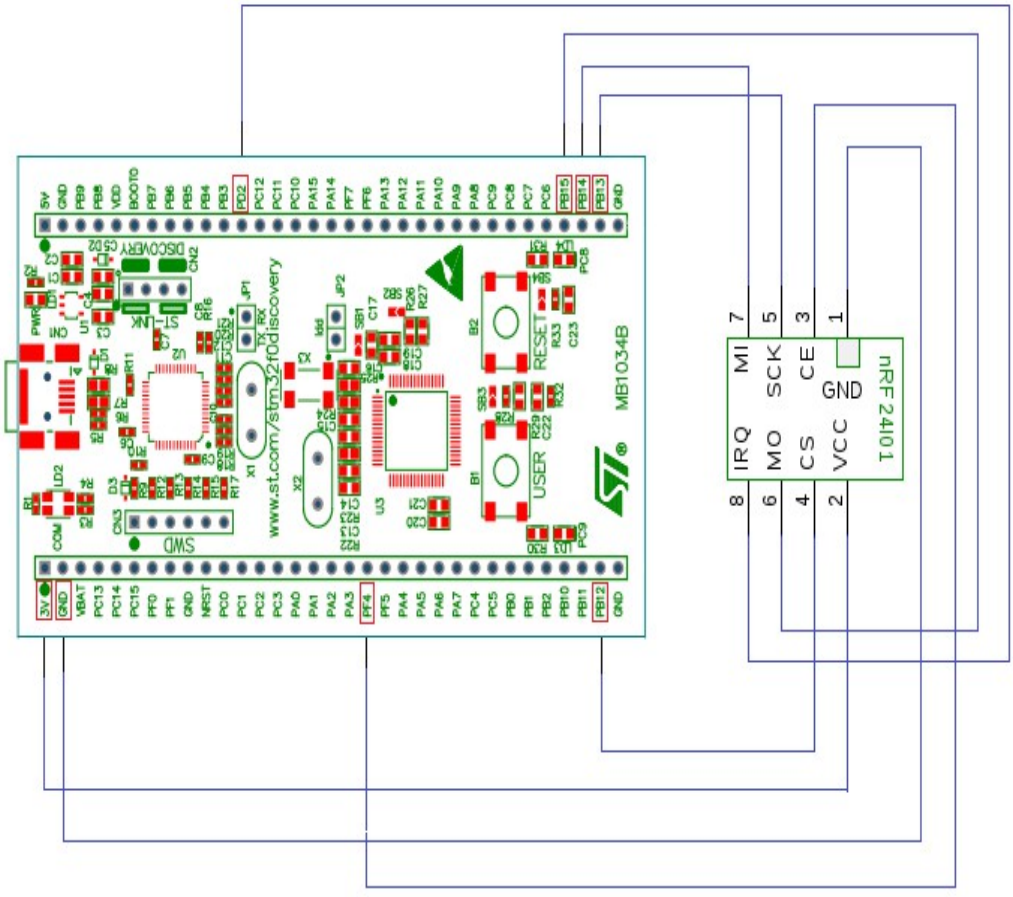


Figure 4: connessione tra i due dispositivi

I due dispositivi sono stati collegati tra loro come mostrato in Figura 3 e nella tabella sottostante.

PIN stm32	PIN nRF24L01
3V	**VCC** - Power Supply (+1.9V - +3.6V DC)
GND	**GND**
PF4 - EVENTOUT	**CE** - Chip Enable Activates RX or TX mode
PB12 - 2_NSS	**CS** - SPI Chip Select
PD2 – 3_ETR	**IRQ** - Maskable interrupt pin. Active low
PB15 – 2_MOSI	**MO** - SPI Slave Data Input
PB14 – 2_MISO	**MI** - SPI Slave Data Output, with tri-state option
PB13 - 2_SCK	**SCK** - SPI Clock

Tabella 1: i PIN utilizzati e la loro modalità per la connessione stm32f/nRF24L01

3.2 Librerie necessarie

Sono state incluse le seguenti librerie:
Le prime 2 sono le librerie del microcontrollore STM32f0, mentre la terza è un header creato appositamente per contenere le definizioni di tutti i comandi ed i registri,utilizzati per il funzionamento dell'nRF24L01, che sono presenti nel datasheet[6].

```
#include "stm32f0xx.h”
#include "stm32f0_discovery.h"
#include "nRF24L01.h"
```

3.3 Abilitazione dei PIN

Il seguente codice mostra come abilitare e configurare correttamente le porte GPIO relative ai pin dell'stm32f0 utilizzati, al fine di permettere la comunicazione tra i due dispositivi secondo quanto scritto nella Tabella 1.

```
void SPI2_Init(void)
{

//SPI & GPIO struct
SPI_InitTypeDef SPI_InitStructure;
GPIO_InitTypeDef     GPIO_InitStructure;

//Enable SPI & GPIO clock
RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI2, ENABLE);
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOB |
RCC_AHBPeriph_GPIOD |  RCC_AHBPeriph_GPIOF , ENABLE);

// MI, MO, SCK
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13 |GPIO_Pin_14 |
GPIO_Pin_15;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_DOWN;
GPIO_Init(GPIOB, &GPIO_InitStructure);
```

```
GPIO_PinAFConfig(GPIOB, GPIO_PinSource13, GPIO_AF_0);
GPIO_PinAFConfig(GPIOB, GPIO_PinSource14, GPIO_AF_0);
GPIO_PinAFConfig(GPIOB, GPIO_PinSource15, GPIO_AF_0);

//CS
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
GPIO_Init(GPIOB, &GPIO_InitStructure);

//CE
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
GPIO_Init(GPIOF, &GPIO_InitStructure);

//IRQ
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
GPIO_Init(GPIOD, &GPIO_InitStructure);


/* SPI configuration */
SPI_I2S_DeInit(SPI2);
SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;
SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low;
SPI_InitStructure.SPI_CPHA = SPI_CPHA_1Edge;
SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_2;
SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
SPI_InitStructure.SPI_CRCPolynomial = 7;
SPI_Init(SPI2, &SPI_InitStructure);
SPI_SSOutputCmd(SPI2, ENABLE);

/* Enable SPI2   */
SPI_Cmd(SPI2, ENABLE);
}
```

E' importante fare alcune semplici ma fondamentali osservazioni sulle configurazioni:
La prima operazione da fare è quella di abilitare il clock delle periferiche che si devono configurare, pena mancato funzionamento del dispositivo.

Come è facile notare, in tutti i casi il numero di pin della configurazione, corrisponde con il numero riportato nella tabella soprastante. L'unica eccezione è quella relativa al pin PF4 che collega il pin CE. In questo caso, nella configurazione viene selezionato "GPIO_Pin_0", questo perchè la porta GPIOF dell'stm32f0 è composta da soli 4 pin PF[4:7] che in fase di programmazione corrispondono ai pin 0-4. In tutti gli altri casi il nome da

utilizzare corrisponde esattamente al nominativo che troviamo riportato nella board.

In generale,per quanto riguarda la configurazione dei pin bisogna tener presente che esistono diverse modalità in base alla funzione che si decide di utilizzare. Un determinato pin può essere infatti configurato come ingresso(IN),uscita(OUT) o in alternate function(AF). Nel nostro caso era necessario il più delle volte utilizzare l'alternate function e quindi il primo passo è stato quello di abilitare le porte in tale modalità (anche se alcuni pin specifici non necessitano di tale configurazione).

Un'altra importante configurazione riguarda il type. In questo caso bisogna controllare tramite il datasheet se è necessario configurare un pin in modalità di pullUp,pullDown o floating

Nel caso degli stm32f0 la modalità di floating corrisponde alla configurazione NOPULL.

Per quanto riguarda infine l'SPI, la configurazione utilizzata risulta essere abbastanza standard. L'unico parametro su cui abbiamo agito per effettuare i test è stato il baud rate prescaler. La configurazione di tale parametro, infatti, permette di variare le condizioni operative relative alla velocità dell'interfaccia. Per evitare che la periferica creasse un collo di bottiglia nella comunicazione tra il transceiver e il microcontrollore si è scelto di utilizzare il prescaler in modo che la velocità fosse massimizzata.

Nonostante quest'ultima osservazione possa risultare ovvia, bisogna prestare molta attenzione ai collegamenti fisici tra i due dispositivi. Dato che il transceiver non possiede alcun led che ci faccia rendere conto della sua operatività, potrebbe capitare facilmente che dei collegamenti fisici non corretti, possano precludere un corretto funzionamento nonostante una corretta configurazione facendo perdere tempo alla ricerca di errori di programmazione non presenti.

3.4 Funzioni di supporto:

Per permettere la corretta comunicazione tra i transceiver, abbiamo implementato le seguenti funzioni di supporto:

```
//legge un registro attraverso l'interfaccia SPI
unsigned char SPI2_readReg(unsigned char reg)

//legge un buffer(es.il pacchetto); il terzo parametro indica la
dimensione del buffer da leggere
unsigned char SPI2_readBuf(unsigned char reg,
                                    unsigned char *pBuf,
                                    unsigned char byte)

//scrive e contemporaneamente legge (a causa del funzionamento
dell'SPI) un registro
unsigned char SPI2_readWriteReg(unsigned char reg,
                                   unsigned char value)
```

```
//scrive e legge un byte sull'SPI
uint8_t SPI2_readWrite(uint8_t byte)

//scrive un buffer(es.il pacchetto)
unsigned char SPI2_writeBuf(unsigned char reg, unsigned char
*pBuf, unsigned char bytes);
```

3.5 TX_Mode

La seguente funzione permette di settare tutti i parametri necessari per l'invio di un pacchetto:

```
void TX_Mode(unsigned char* addr,
                    unsigned char addr_leng,
                    unsigned char pload_leng,
                    unsigned char aa,
                    unsigned char rxaddr,
                    unsigned char set_retr,
                    unsigned char rfch,
                    unsigned char rfset,
                    unsigned char conf)
{
CE_L(); //GPIO_ResetBits(GPIOF, GPIO_Pin_0)

//scrittura nel registro dell'indirizzo di trasmissione e
di ricezione
SPI2_writeBuf(WRITE_REG + TX_ADDR, addr, addr_leng);
SPI2_writeBuf(WRITE_REG + RX_ADDR_P0, addr, addr_leng);

//Enable 'Auto Acknowledgment'
SPI2_readWriteReg(WRITE_REG + EN_AA, aa);

//Enabled RX Addresses
SPI2_readWriteReg(WRITE_REG + EN_RXADDR, rxaddr);

//Setup ritrasmissione automatica
SPI2_readWriteReg(WRITE_REG + SETUP_RETR, set_retr);

//Selezione del canale (0-125)
SPI2_readWriteReg(WRITE_REG + RF_CH, rfch);

//Selezione della velocità di trasmissione (1 o 2 Mbps) e
dell'output power
SPI2_readWriteReg(WRITE_REG + RF_SETUP, rfset);

//scrittura nel registo di configurazione degli altri
parametri necessari
SPI2_readWriteReg(WRITE_REG + CONFIG, conf);

CE_H(); //GPIO_SetBits(GPIOF, GPIO_Pin_0)
}
```

Prima di impostare tutti i parametri è necessario resettare CE e poi settarlo nuovamente. Vediamo un attimo nel dettaglio i vari registri[6] e comandi:

- **TX_ADDR**: contiene l'indirizzo di trasmissione e viene utilizzato solo dal trasmettitore.
- **RX_ADDR_PX:** numero di byte in RX payload nella data pipeX (X=1...5) . E' possibile utilizzare fino a 5 data pipe. L'argomento verrà discusso meglio più avanti, si veda il paragrafo 5.1.
- **EN_AA (**EnhancedShockBurst™**): permette di abilitare l'a**uto acknowledgment' su una o più delle 5 data pipe
- **EN_RXADDR**: Abilita l'indirizzo per la ricezione degli ack su una o più delle 5 data pipe (deve essere uguale all'indirizzo di trasmissione)
- **RF_CH**: canale di trasmissione
- **RF_SETUP**: velocità e potenza di trasmissione
- **CONFIG**: tra le varie funzionalità permette di abilitare/disabilitare il CRC
- **WRITE_REG** è il comando W_REGISTER[7] : definito come 001A AAAA permette di scrivere nel registro AAAAA

Subito dopo aver settato tutti i parametri necessari è possibile inviare il pacchetto scrivendo il payload nel buffer di trasmissione:

```
SPI2_writeBuf(WR_TX_PLOAD, packet, TX_PLOAD_WIDTH);
```

e pulire l'interrupt flag TX_DS nel registro STATUS [6]:

```
SPI2_readWriteReg(WRITE_REG+STATUS,(SPI2_readReg(READ_REG+STATUS)));
```

SE IL FLAG TX_DS NON VIENE PULITO DOPO OGNI INVIO IL TRANSCEIVER NON SARA' IN GRADO DI INVIARE NUOVAMENTE

- **WR_TX_PLOAD** è il comando che permette di scrivere il TX-payload partendo dal byte 0 definito come 1010 0000 [7]
- **TX_PLOAD_WIDTH** è la lunghezza del payload (1-32 bytes)
- **READ_REG** è il comando R_REGISTER, duale al W_REGISTER: è definito come 00A AAAA che permette di leggere il contenuto del registro AAAAA.

3.6 RX_Mode:

E' funzione duale alla TX_Mode e permette di settare tutti i parametri necessari per permettere al transceiver di ricevere.

```
void RX_Mode(unsigned char *addr,
             unsigned char addr_leng,
             unsigned char aa,
             unsigned char rxaddr,
             unsigned char rfch,
             unsigned char pload_leng,
             unsigned char rfset,
             unsigned char conf)
```

```
{
CE_L();  //GPIO_ResetBits(GPIOF, GPIO_Pin_0)

//scrittura dell'indirizzo di ricezione
SPI2_writeBuf(WRITE_REG + RX_ADDR_P0, addr, addr_leng);

//Enable 'Auto Acknowledgment'
SPI2_readWriteReg(WRITE_REG + EN_AA, aa);

//Enabled RX Addresses
SPI2_readWriteReg(WRITE_REG + EN_RXADDR, rxaddr);

//Selezione del canale (0-125)
SPI2_readWriteReg(WRITE_REG + RF_CH, rfch);

//Number of bytes in RX payload in data pipe 0
SPI2_readWriteReg(WRITE_REG + RX_PW_P0, pload_leng);

//Selezione della velocità di trasmissione (1 o 2 Mbps) e
dell'output power
SPI2_readWriteReg(WRITE_REG + RF_SETUP, rfset);

/Scrittura nel registo di configurazione degli altri
parametri necessari
SPI2_readWriteReg(WRITE_REG + CONFIG, conf);

CE_H();  //GPIO_SetBits(GPIOF, GPIO_Pin_0)
}
```

La maggior parte registri utilizzati sono gli stessi discussi nel paragrafo 3.5.
Anche nella modalità di ricezione, prima di impostare tutti i parametri, è necessario resettare CE e poi settarlo nuovamente.
In questo semplice esempio stiamo utilizzando solo il data pipe0, quindi l'indirizzo di ricezione viene scritto nel registro `RX_ADDR_P0[6]`.

I VALORI DEI REGISTRI COMUNI CON LA FUNZIONE TX_MODE DEVONO ESSERE GLI STESSI!

E' possibile leggere il payload scrivendo semplicemente:

```
SPI2_readBuf(RD_RX_PLOAD,rx_buf,TX_PLOAD_WIDTH);
```

dove, RD_RX_PLOAD è il registro contenente i dati ricevuti[6]. Ovviamente, utilizzando una tecnica di polling su tale buffer senza gli opportuni controlli, potrebbe capitare che alcuni pacchetti non vengano letti o che, se nessun pacchetto è stato ricevuto,si leggano dati inconsistenti dal registro che potrebbero erroneamente essere scambiati per pacchetti validi.

Un approccio più sicuro è quello di utilizzare la tecnica dell'interrupt, la cui configurazione verrà discussa successivamente

AL CONTRARIO DEL REGISTRO RD_RX_PLOAD, IL REGISTRO WR_TX_PLOAD UTILIZZATO DAL TRASMETTITORE NON PUO'ESSERE LETTO!

3.6 Trasmettitore e Ricevitore

Ecco un semplice codice per il trasmettitore:

```
int main(void)

     {
     unsigned char TX_ADDRESS[TX_ADR_WIDTH]=
     {0x34,0x43,0x10,0x10,0x01};
     unsigned char TX_ADR_WIDTH=5;
     unsigned char TX_PLOAD_WIDTH=9;
     unsigned char pacchetto[TX_PLOAD_WIDTH]=”pacchetto”;

     //Init dei PIN  (paragrafo 3.3)
     SPI2_Init();

     //pulizia della coda di trasmissione da eventuali
       pacchetti rimasti [7]
      SPI2_readWriteReg(FLUSH_TX,0);

     //Set dei parametri - enanched shock burst 1Mbps
      TX_Mode(TX_ADDRESS,
       TX_ADR_WIDTH,
       TX_PLOAD_WIDTH,
       0x01,
       0x01,
       0x1a,
       40,
       0x07,
       0x0e);

     while(1)
      {
        //scrittura nel buffer
        SPI2_writeBuf(WR_TX_PLOAD, pacchetto, TX_PLOAD_WIDTH);
        //pulizia del flag TX_DS
       SPI2_readWriteReg(WRITE_REG+STATUS,
                         SPI2_readReg(READ_REG+STATUS)));
     };

   }
```

e un semplice Ricevitore:

```
int main(void)
      {

      unsigned char TX_ADDRESS[TX_ADR_WIDTH]=
      {0x34,0x43,0x10,0x10,0x01};
      unsigned char TX_ADR_WIDTH=5;
      unsigned char TX_PLOAD_WIDTH=9;
      unsigned char rx_buf[TX_PLOAD_WIDTH];


      //Init dei PIN  (paragrafo 3.3)
      SPI2_Init();
      //Set dei parametri - enanched shock burst 1Mbps
      RX_Mode
      (TX_ADDRESS,
      TX_ADR_WIDTH,
      0x01,
      0x01,
      40,
      TX_PLOAD_WIDTH,
      0x07,
      0x0f);


while(1)
       {
         SPI2_readBuf(RD_RX_PLOAD,rx_buf,TX_PLOAD_WIDTH);
         WaitAWhile();
      };

   }
```

3.6. Abilitazione dell'interrupt

Come già anticipato, esiste la possiblità di controllare un interrupt nel caso in cui si verifichino determinate situazioni nel transceiver. Nel registro CONFIG, sono presenti 3 bit che vengono utilizzati come maschere per l'abilitazione e la disabilitazione(di default sono abilitati) di tale interrupt. Queste maschere permettono semplicemente di riflettere sul pin IRQ, i valori di determinati registri, sottoforma di segnali attivi bassi.
In particolar modo, i registri su cui si può abilitare l'interrupt sono:

- TX_DS: avvenuto invio di un pacchetto in modalità trasmissione (nel caso in cui è abilitato l'auto-ack, questo evento si verifica esclusivamente alla ricezione dell'ack e non nel momento in cui viene avviata la trasmissione)

- RX_DR: avvenuta ricezione di un pacchetto in modalità ricezione

- MAX_RT: numero massimo di ritrasmissioni(in seguito a mancata ricezione di ACK) in trasmissione con auto-ack abilitato

Per poter utilizzare l'interrupt, bisogna innanzitutto abilitare una linea di interrupt in fase di configurazione. Alla configurazione illustrata nel paragrafo 3.3, bisogna aggiungere alcune righe di codice che permettono di abilitare, settare e collegare al pin apposito l'interfaccia di interrupt:

```
EXTI_InitTypeDef   EXTI_InitStructure;
NVIC_InitTypeDef   NVIC_InitStructure;

 /* Enable SYSCFG clock */
RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);

/* Connect EXTI Line to the pin */
SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOD, EXTI_PinSource2);

/* Configure the EXTI line */
EXTI_InitStructure.EXTI_Line = EXTI_Line2;
EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;
EXTI_InitStructure.EXTI_LineCmd = ENABLE;
EXTI_Init(&EXTI_InitStructure);

/* Enable and set the EXTI Interrupt */
NVIC_InitStructure.NVIC_IRQChannel = EXTI2_3_IRQn ;
NVIC_InitStructure.NVIC_IRQChannelPriority = 0x01;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);
```

Una volta abilitato l'interrupt è possibile gestirlo tramite l'handler relativo al canale che è stato configurato:

```
void EXTI2_3_IRQHandler(void)
    {
          if(EXTI_GetITStatus(EXTI_Line2) != RESET)
    {
        //Handle the interrupt

        EXTI_ClearITPendingBit(EXTI_Line2);
    }

}
```

In fase di configurazione bisogna fare attenzione a settare correttamente la linea EXTI relativa al pin scelto per la lettura dell'interrupt e l'IRQ channel nelle impostazioni dell'NVIC. Inoltre risulta importante fare attenzione alla priorità da assegnare (compresa tra 0 e 3, in cui un valore minore significa maggiore priorità) in modo da evitare che si verifichino situazioni indesiderate (bisogna infatti considerare che il main del programma è impostato ad una priorità minore rispetto agli interrupt). Nell'implementazione dell'handler, bisogna inoltre fare attenzione a non dimenticarsi di effettuare il clear pending bit, onde

evitare che il programma resti bloccato senza poter eseguire l'interrupt successivo.
NB: E' capitato(in modalità ricezione) che nonostante fossero state configurate le maschere del registro CONFIG in modo da garantire che l'interrupt si verificasse solo in caso di avvenuta ricezione di un pacchetto, è stato necessario controllare il registro RX_DR a causa di situazioni in cui si è verificato l'evento nonostante nessuna ricezione. Si consiglia quindi di effettuare comunque le verifiche dei registri interessati (anche in caso di trasmissione). Infine, per poter sbloccare la ricezione del successivo interrupt risulta necessario effettuare un reset del registro STATUS, come già precedentemente accennato.

3.7 Connessione tra stm32f0-discovery e XBee - USB Board

Per poter effettuare buona parte dei nostri test, al fine di non avere ritardi aggiuntivi che potevano dare dei risultati non veritieri, invece di utilizzare il semplice printf in modalità di debug durante l'esecuzione del codice, si è preferito stampare i risultati ottenuti, solo alla fine dell'esecuzione dei test, inviandoli al computer tramite una connessione seriale. Poiché i nostri portatili non disponevano di ingresso seriale, abbiamo dovuto utilizzare il dispositivo Xbee – USB Board 990002 che effettua una conversione del segnale seriale a quello USB.
Per connettere tra loro i due dispositivi abbiamo usato la USART1 e abbiamo collegato i cavi nel seguente modo:

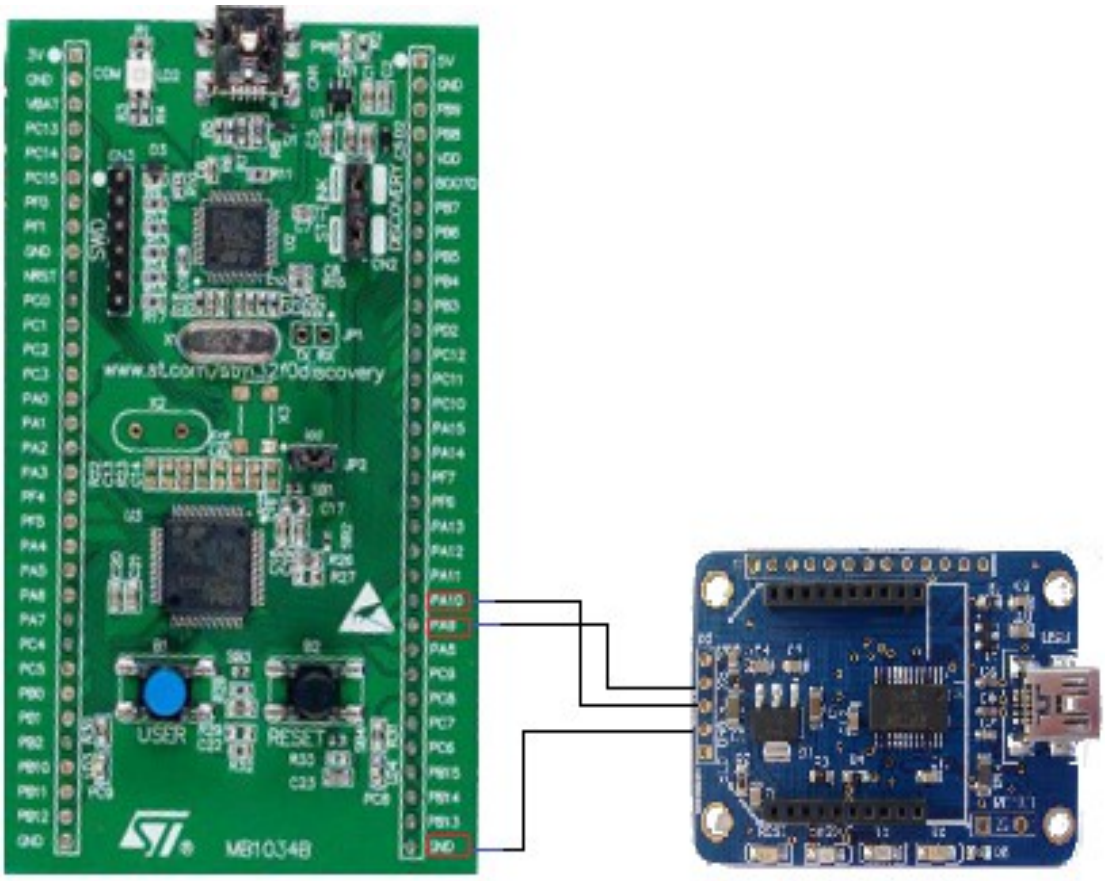


Figure 5: connessione stm32 - Xbee USB Board

- PA9 from the STM32 (USART1_TX) connesso al pin RXD dell'Xbee – USB Board
- PA10 from the STM32 (USART1_RX) connesso al pin TXD dell'Xbee – USB Board
- GND del STM32 connesso al GND dell'Xbee – USB Board

Per poter leggere le informazioni ricevute è stato implementato un semplice programma utilizzando Visual Studio e C#, con un'interfaccia grafica in WPF.

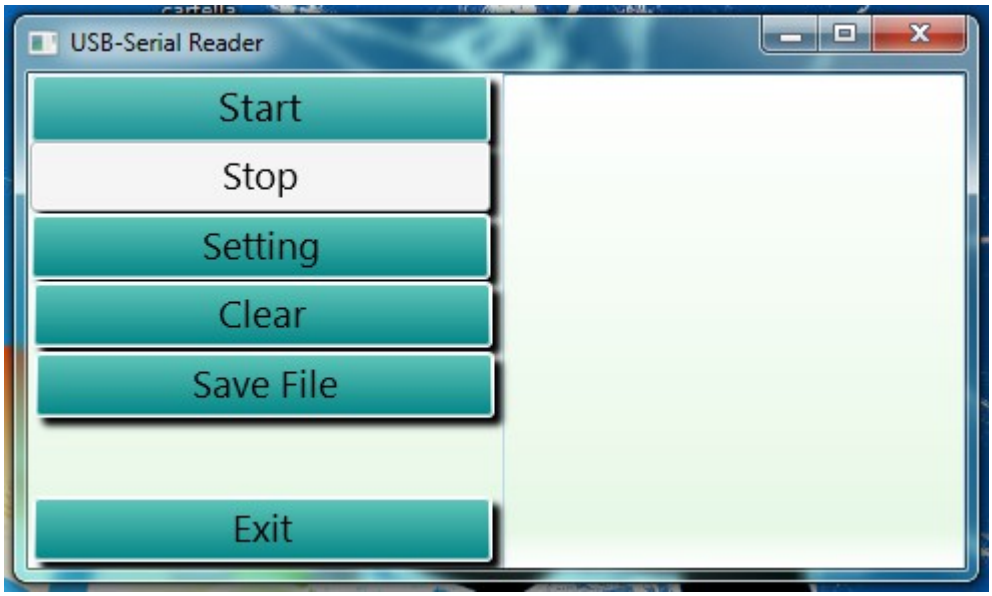


Figure 6: USB Reader

Tuttavia, dopo aver implementato tale programma abbiamo trovato Terminal.exe [4], e nonostante la nostra implementazione funzionasse perfettamente, abbiamo preferito utilizzare quest'ultimo poiché dotato di più funzionalità invece che modificare la nostra implementazione poiché non era questo il nostro obiettivo finale.

4 Test Effettuati

Al fine di valutare le prestazioni del transceiver sono stati effettuati diversi tipi di test che possono essere riassunti nelle seguenti categorie:

- Test relativi alla velocità di comunicazione tramite SPI e delle funzioni principalmente utilizzate.
- Test per calcolare il minimo delay necessario per il corretto invio/ricezione dei pacchetti
- Test di trasmissione in presenza di interferenza da parte di un altro trasmettitore su canali di comunicazione adiacenti
- Test di distanza

4.1 Velocità comunicazione SPI e delle funzioni principalmente utilizzate

Tali test sono stati effettuati utilizzando una funzione che implementa uno stopwatch che effettua il calcolo della durata di tempo di una determinata operazione.
Eseguando singolarmente i test per le diverse operazioni, i risultati ottenuti sono stati i seguenti:

- velocità scrittura buffer(es.scrittura pacchetto)
  - 20byte: 72-74 us = 2,2Mbps
  - 32byte: 112-114 us = 2,26Mbps
  - 10byte: 39-40 us = 2Mbps

- velocità readWritereg(8bit+8bit): 15-18 us = 888Kbps
- velocità readReg(8 bit): 8-10 us = 800Kbps
- readReg(8 bit)+assegnazione: 8-10 us = 800Kbps
- velocità ciclo di invio (inserimento,reset registro,lettura coda,if piena+incremento):
  - 20byte: 94-96 us= 1,6Mbps
  - 32byte: 134-136 us = 1,8Mbps
  - 10byte: 61-62 us = 1,29Mbps
- velocità ciclo di invio (inserimento,reset registro):
  - 32byte: 127-129 us = 1,98Mbps
  - 20byte: 86-89 us = 1,79Mbps
  - 10byte: 53-56 us = 1,42Mbps
- SPI_Init(): 47-49 us
- Tx_Mode(): 271-272 us
- Rx_Mode(): 160-162 us
- setRf_channel(): 10-11 us
- read(): 62-64 us
- velocità lettura buffer (es.lettura pacchetto)
  - 32byte: 112-115 us = 2,2Mbps
  - 20byte: 72-75 us = 2,1Mbps
  - 10byte: 39-41 us = 2Mbps

4.1 Calcolo del minimo delay necessario per il corretto invio/ricezione dei pacchetti

Questi test sono stati effettuati in modalità ShockBurst ™, utilizzando un pacchetto di lunghezza 5byte (address) + 2byte (crc) + payload (di lunghezza variabile).
Per poter ottenere dei risultati corretti abbiamo fatto uso dell'interrupt e di un clock. L'idea di base è stata quella di suddividere il tempo in intervalli di lunghezza T e inviare un pacchetto all'inizio di ogni intervallo. Se entro il termine del tempo T riuscivamo a ricevere correttamente l'interrupt di trasmissione era possibile ridurre l'intervallo di un valore delta fino a trovare la lunghezza minima dell'intervallo all'interno del quale era possibile inviare un pacchetto correttamente.
Nonostante il test citato è stato quello che ci ha permesso di ottenere i risultati più precisi,nello stesso ambito sono stati effettuati anche altri test che hanno portato il più delle volte a risultati del tutto simili e che ci hanno dato conferma della veridicità dei

valori ottenuti.
Innanzitutto è stato effettuato un test in cui veniva inserito un nuovo pacchetto all'interno dell'handler dell'interrupt ogni volta che un pacchetto veniva inviato. Utilizzando uno stopwatch abbiamo quindi calcolato l'intervallo di tempo che intercorreva tra due successivi interrupt che stava ad indicare il momento di inserimento del pacchetto e il suo invio.
Un altro tipo di test effettuato prevedeva di effettuare iterativamente l'inserimento di un pacchetto, l'attesa di un delay e il controllo della coda di trasmissione. Diminuendo di volta in volta il delay abbiamo cercato il minimo delay necessario affinchè non si verificasse mai che la coda di trasmissione risultasse completamente occupata, in modo da verificare se la velocità di inserimento nel buffer fosse maggiore rispetto alla velocità di invio del transceiver.

I risultati ottenuti (relativi al primo test citato) sono stati i seguenti:

- **MODALITA' 2Mb**

  Payload lenght = 10 byte
  Min Delay: 254 us
  Goodput: 314,96 Mbps
  Throughput: 535,43 Mbps

  Payload lenght =32 byte
  Min Delay: 407 us
  Goodput: 628,9 Mbps
  Throughput: 766,58 Mbps

- **MODALITA' 1Mb**

  Payload lenght = 10 byte
  Min Delay: 328 us
  Goodput: 243,9 Mbps
  Throughput: 414,6 Mbps

  Payload lenght = 32 byte
  Min Delay: 570 us
  Goodput: 449,1 Mbps
  Throughput: 547,37 Mbps

4.2 Trasmissione in presenza di interferenza da parte di un altro trasmettitore su canali di comunicazione adiacenti

Sono stati utilizzati pacchetti con payload di 32 byte con lunghezza totale di 39 byte. I test sono stati effettuati utilizzando 3 transceiver in modalità ShockBurst ™ :

- un transceiver in trasmissione sul canale x
- un transceiver in ricezione sul canale x
- un transceiver in trasmissione sul canale x± y

L'obiettivo di questi test è stato quello di trovare il valore minimo di y al fine di avere una corretta ricezione di tutti i pacchetti inviati sul canale x.

Abbiamo scoperto che, al fine di avere i risultati desiderati, il transceiver “disturbatore” deve trasmettere, nella modalità a 1 Mbps a una distanza minima di 2 canali mentre nella modalità a 2Mbps ad una distanza minima di 3 canali
Di seguito, nel dettaglio, i risultati ottenuti:

- **Modalità di trasmissione 1Mbps:**
  - 1 CANALE DI DISTANZA: valore medio ottenuto 47,77%
    - test su campioni di 50 pacchetti:
      - percentuale massima di pacchetti ricevuti: 90,9%
      - percentuale minima di pacchetti ricevuti: 22,83%
      - media: 45%
    - test su campioni di 100 pacchetti:
      - percentuale massima di pacchetti ricevuti: 66,6%
      - percentuale minima di pacchetti ricevuti: 22,9%
      - media: 49,16%
  - 2 CANALI DI DISTANZA (test su campioni di 50 pacchetti)
    - percentuale massima di pacchetti ricevuti: 100%
    - percentuale minima di pacchetti ricevuti: 98%
    - media: 99,7%
  - 3 CANALI DI DISTANZA (test su campioni di 50 pacchetti):
    - percentuale massima di pacchetti ricevuti: 100%
    - percentuale minima di pacchetti ricevuti: 96,15%
    - media: 99,6%
  - 5 CANALI DI DISTANZA:
    - percentuale massima di pacchetti ricevuti: 100%
    - percentuale minima di pacchetti ricevuti: 100%
    - media: 100%

- **Modalità di trasmissione 2Mbps:**
  - 1 CANALE DI DISTANZA (test su campioni di 50 pacchetti)
    - percentuale massima di pacchetti ricevuti: 36,76%
    - minimo: 34,96
    - media: 35,95%
  - 2 CANALI DI DISTANZA (test su campioni di 50 pacchetti):
    - percentuale massima di pacchetti ricevuti: 100%
    - minimo: 63,3%
    - media: 87,6%
  - 3 CANALI DI DISTANZA:
    - percentuale massima di pacchetti ricevuti: 100%

- percentuale minima di pacchetti ricevuti: 100%
- media: 100%

4.2 Test di distanza

Nonostante il prezzo ridotto dei transceiver, la distanza massima di trasmissione è risultata abbastanza soddisfacente. I dispositivi sono in grado di trasmettere e ricevere correttamente tutti i pacchetti, utilizzando il delay minimo trovato nei test 4.1, a una distanza di circa 20m in ambiente chiuso e in luoghi fortemente rumorosi. E' stato inoltre constatato che a distanze maggiori(comprese tra i 50 e i 100 metri) i transceiver continuano a comunicare ma con una frequenza di successo di trasmissione inferiore.

5. Implementazione di un protocollo MAC di tipo CSMA

5.1 Multiricevitore [8]

Prima di procedere alla realizzazione del protocollo, è risultato necessario capire determinati concetti relativi alla capacità del transceiver (in modalità di ricezione) di comportarsi da multiricevitore per ricevere "contemporaneamente" da diversi dispositivi.
La configurazione di multiricevitore, è una caratteristica utilizzata in modalità di ricezione che contiene un set di 6 data pipes parallele con indirizzo univoco.
Una data pipe è un canale logico nel canale fisico di ricezione. Ogni data pipe ha il proprio indirizzo fisico nell' nRF24L01.

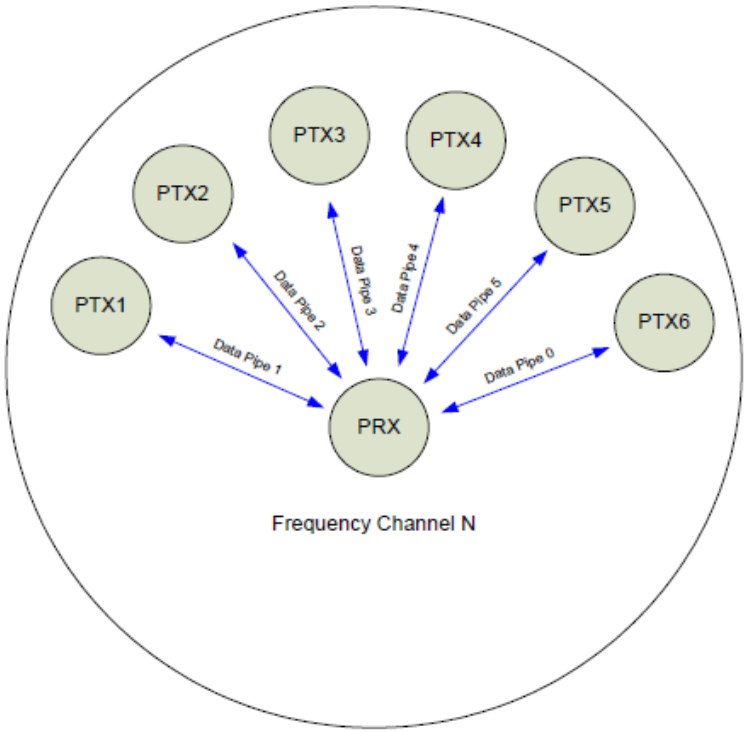


Figure 7: multiricevitore

Le seguenti configurazioni sono comuni a tutte le data pipes:

- abilitazione/disabilitazione del CRC (che è sempre abilitato se stiamo utilizzando l'Enhanced ShockBurst™)
- sceda di codifica CRC
- lunghezza dell'indirizzo di ricezione
- canale
- Air data rate

Le data pipes possono essere abilitate settando l'opportuno bit nel registro EN_RXADDR.
Di default solo le data pipe 1 e 0 sono abilitate.
L'indirizzo di ogni data pipe viene configurato nel registro RX_ADDR_PX [6]

Ogni data pipe ha un indirizzo configurabile di lunghezza massima 5 byte. La data pipe 0 ha un unico indirizzo a 5 byte.
Le data pipes 1-5, invece, condividono i 4 byte più significativi dell'indirizzo. Il LSB deve essere unico per tutte le 6 pipes.
La figura mostra un esempio di indirizzamento delle 6 data pipe

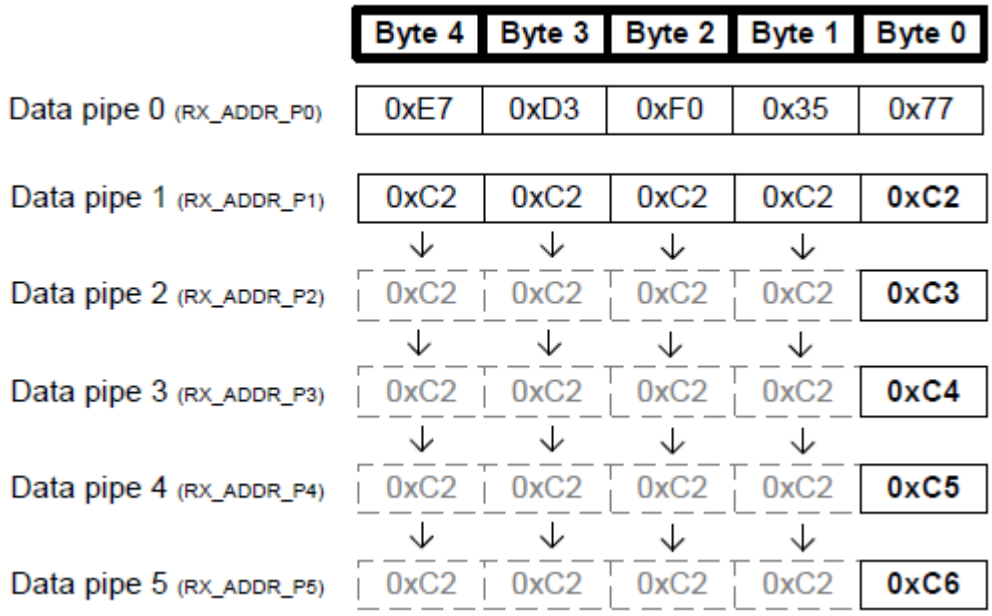


Figure 8: Indirizzamento data pipe 0-5

RX_ADDR_PX (x=2..5) ha dimensione pari a 7 bit, quindi in fase di configurazione del ricevitore è necessario scrivere solo il valore del byte diverso dall'indirizzo della data pipe 1

5.2 L'algoritmo

L'ultima parte del progetto prevedeva la realizzazione e il test di un protocollo MAC di tipo CSMA (Carrier Sense Multiple Access). Come già detto, infatti, tramite il transceiver nrf24l01 si ha la possibilità di effettuare l'ascolto della portante per verificare se in un determinato istante temporale il canale di comunicazione risulta essere occupato. Tramite la lettura del registro CD(Carrier Detect) del dispositivo, verrà infatti verificato se nel canale risulta essere presente un segnale con una potenza superiore a

-65db. Il transceiver purtoppo non da la possiblità di configurare tale soglia e quindi si è costretti ad utilizzare tale valore in qualsiasi ambiente in cui si intende utilizzare il dispositivo.

IL REGISTRO E' LEGGIBILE CORRETTAMENTE SOLO SE IL TRANSCEIVER E' IMPOSTATO IN MODALITA' RICEZIONE (DA MINIMO 258us)

Al fine di semplificare le verifiche sulla validità dell'algoritmo, si è scelto di creare una rete che prevede una topologia in cui tutti i nodi eccetto uno saranno configurati da trasmettitore e cercheranno di inviare secondo le regole stabilite dall'algoritmo(che verrà implementato esclusivamente su questi) ad un singolo ricevitore che resterà in ascolto sul medesimo canale in attesa di pacchetti.

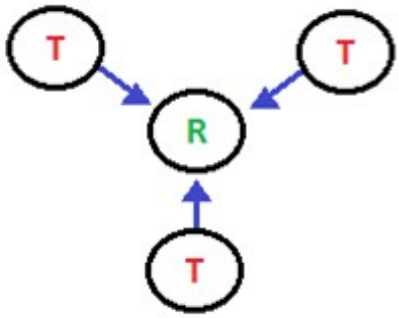


Figure 8: Trasmettitori e Ricevitore

Lo schema nella pagina seguente , rappresenta il funzionamento globale dell'algoritmo (lato trasmettitore) che è stato implementato.
Come è possibile notare, l'algoritmo si suddivide in 3 diverse fasi:

- generazione dei pacchetti
- invio dei pacchetti
- gestione delle collisioni

Il primo passo effettuato è stato quello di creare un sistema che permettesse in automatico di generare i pacchetti che successivamente verranno inviati e che lavorasse in maniera asincrona ed autonoma dal resto del protocollo. Il generatore dei pacchetti, quindi, è stato implementato all'interno del file “utils.c” sfruttando una delle periferiche timer della board (TIM2) per creare degli intervalli di tempo con una frequenza media prefissata, che vengono gestiti attraverso il canale degli interrupt. Dato che tale periferica (così come il Systick) una volta impostato creerà degli interrupt ad intervalli regolari, mentre il nostro obiettivo era quello di avere una frequenza di generazione media prefissata ma non costante, ad ogni interrupt la periferica viene reimpostata con una diversa frequenza (all'interno di una determinato range di valori) generata tramite un apposita funzione che creerà un numero random.
Ogni volta che si verifica l'interrupt, si prevede quindi la creazione di un pacchetto che sarà inserito nella coda dei dati da inviare e la reimpostazione del timer.

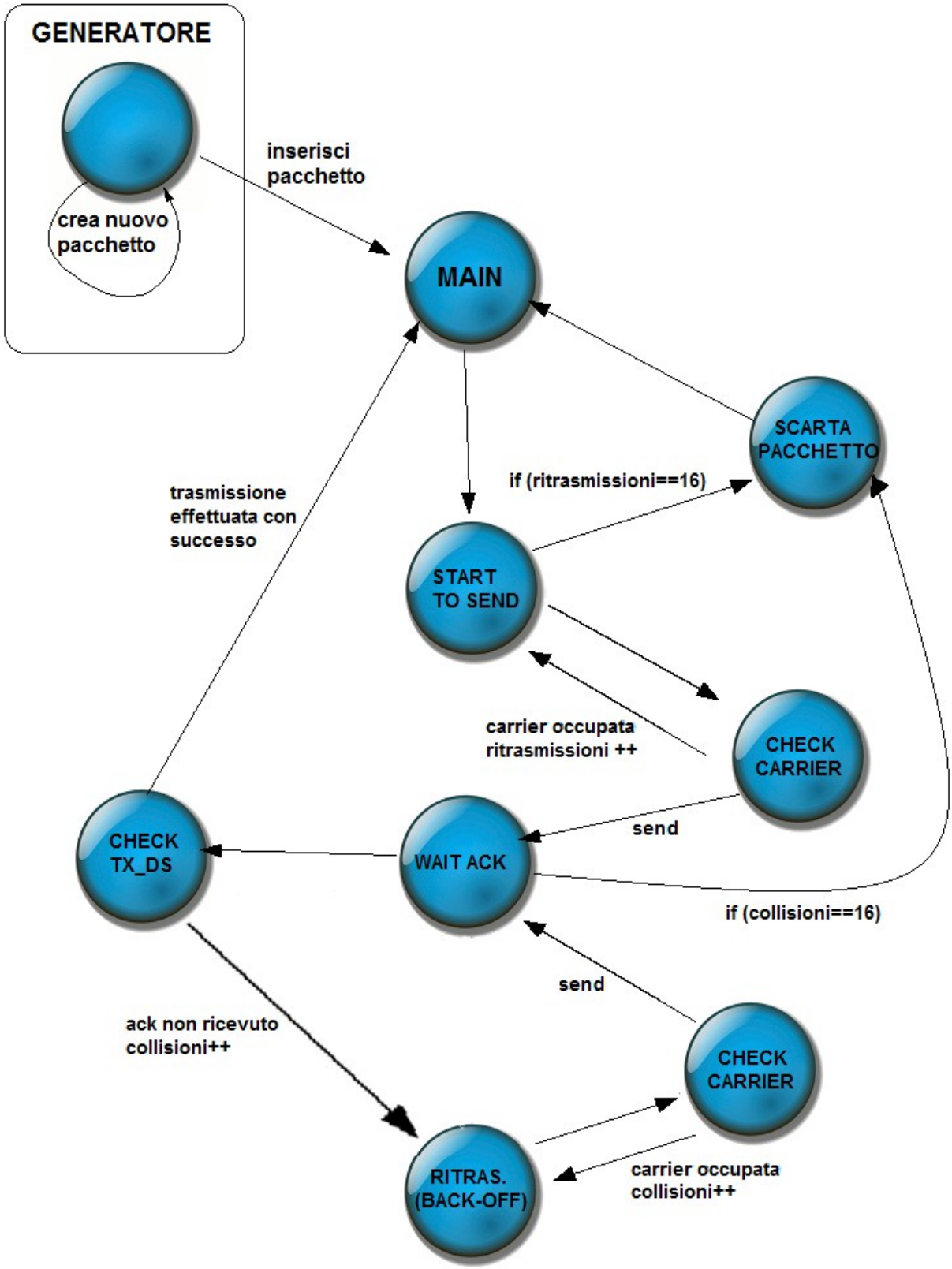


Figure 9: Schema a blocchi dell'algoritmo

```
void TIM2_IRQHandler(void)
       {
        if (TIM_GetITStatus(TIM2, TIM_IT_Update) != RESET)
        {
            elem=setPacketInfo(SOURCE,DEST,NAME,VALUE);
            addPacketElem(elem);

            srand(n_seed);
            n_seed=next_seed(n_seed);
            generator_delay= generator_fix_range + ((rand() %
                  generator_variable_range + 1));
            generator_init(generator_delay);

            TIM_ClearITPendingBit(TIM2, TIM_IT_Update);
        }
      }
```

La stessa funzione per la generazione dei numeri random, verrà utilizzata inoltre per la fase dell'algoritmo relativa all'invio e alla gestione delle collisioni.
Solitamente, il seed utilizzato nelle funzioni random viene generato tramite il timestamp corrente. Dato che i tempi di generazione e invio si aggirano nell'ordine dei microsecondi, mentre i timestamp hanno una precisione dei millisecondi, è stato creato un metodo sostitutivo a quello tradizionale, che prevede l'utilizzo iterativo della stessa funzione random per creare di volta in volta un nuovo seed.

Essendo la funzione pseudocasuale risulta importante che ogni trasmettitore in fase di configurazione abbia un numero di seed iniziale univoco per evitare che tutti abbiano la stessa sequenza di seed.

```
#define next_seed(s)  ((rand() % 4294947295 + 20000))
```

```
int rand(void)
      {
       next = next * 1103515245 + 12345;
       return (unsigned int)(next/65536) % 32768;
      }

void srand(unsigned int seed)
      {
       next = seed;
      }
```

Dato che i pacchetti verranno inviati uno per volta e il tempo necessario per l'intero procedimento può avere bisogno di un tempo superiore alla frequenza di generazione, abbiamo bisogno di una struttura che ci permetta di conservare temporaneamente i dati generati. A tale scopo è stata creata una struttura all'interno del file “init.c” con le

caratteristiche classiche della coda e le relative funzioni per la sua gestione (inserimento,prelievo tramite la regola FIFO, eliminazione e verifica di coda piena e vuota).

```
static QUEUE_TYPE queue[ARRAY_SIZE];
      static size_t head;
      static size_t tail;

      /* Funzioni per la gestione della Coda */
      void addPacketElem(QUEUE_TYPE elem);
      void deleteHead();
      QUEUE_TYPE getPacketElem(void);
      uint8_t is_empty(void);
      uint8_t is_full(void);
```

Il payload del pacchetto di livello fisico, è stato strutturato appositamente per contenere una struttura relativa al nostro livello MAC, che prevede 4 campi che potranno essere utilizzati in una applicazione reale:

- indirizzo del trasmettitore
- indirizzo del ricevitore
- nome della variabile
- valore della variabile

```
typedef struct Data{
      uint16_t source;
      uint16_t dest;
      uint16_t name;
      uint32_t value;
      }Data;
```

Una volta che i pacchetti sono pronti per essere inviati, si può procedere con il tentativo di invio. Nel main del programma, in seguito alla fase di inizializzazione del dispositivo in modalità di trasmissione (il dispositivo viene inoltre impostato in modalità enanched shock burst, auto ack abilitato e auto ritrasmissione disabilitata), vengono effettuati i seguenti controlli:

- Presenza di dati da inviare
- Variabile di lock sbloccata

Dato che, come detto in precedenza, i pacchetti vengono inviati uno alla volta, si è deciso di utilizzare una variabile che serve da semaforo per evitare che vengano gestiti più pacchetti contemporaneamente.

```
          while(1)
          {

            if(lock==0 && (!is_empty()) && stop==0)
            {
              lock=1;
              startSend();
            }
          }
```

Una volta prelevato (ma non cancellato) un dato dalla coda, la prima operazione da effettuare è quella di verificare se il canale è occupato oppure si ha l'immediata possibilità di tentare l'invio. Come detto in precedenza la verifica del canale viene effettuata tramite le lettura del registro CD.

```
     uint8_t checkAndSend(){

          SPI2_readWriteReg(WRITE_REG + CONFIG, 0x01);
          RX_Mode(TX_ADDRESS2,TX_ADR_WIDTH2,0x00,0x00,100,
           TX_PLOAD_WIDTH2,0x07,0x03);
          MIOS32_DELAY_Wait_uS(258);

          if(SPI2_readReg(READ_REG+CD)==0){
           SPI2_readWriteReg(WRITE_REG + CONFIG, 0x00);

          TX_Mode(TX_ADDRESS,TX_ADR_WIDTH,TX_PLOAD_WIDTH,0x00
          ,0x01,0x00,100,0x07,0x5e);

           MIOS32_DELAY_Wait_uS(20);
           CE_H();

           waitAck();
           return 1;
          }

          SPI2_readWriteReg(WRITE_REG + CONFIG, 0x00);

          TX_Mode(TX_ADDRESS,TX_ADR_WIDTH,TX_PLOAD_WIDTH,0x00,
          0x01,0x00,100,0x07,0x5e);
          return 0;
          }
```

Nel caso in cui non sono presenti interferenze nel canale, si procede immediatamente all'invio del dato. In caso contrario, si procederà con una fase in cui sono previsti 16 tentativi di ritrasmissione. Tra un tentativo di ritrasmissione ed il successivo, viene prima atteso un intervallo di tempo random all'interno di un determinato range di valori, sfruttando una funzione di delay (impostata con un ritardo composto dalla somma di un tempo fisso e un tempo random creato con la funzione random precedentemente illustrata).

```
     uint8_t startSend()
          {
           CE_L();
           SPI2_writeStruct(getPacketElem());

           MIOS32_DELAY_Wait_uS(retrasmission_delay_fix_range);

           if(checkAndSend()==1)
           {
            return 0;
          }
          else
          {
          while(retrasmissions!=16)
          {
           srand(n_seed);
           n_seed=next_seed(n_seed);

           MIOS32_DELAY_Wait_uS(retrasmission_delay_fix_range +
(rand()%          retrasmission_delay_range));

           if(checkAndSend()==1)
             return 0;

           retrasmissions++;
           }
          }
           cleanAndRestart(0);
           return 0;
          }
```

Se al termine dei 16 tentativi non si è riusciti ad inviare il dato, quest'ultimo viene scartato (cancellato definitivamente dalla coda) e si procede con il pacchetto successivo. Se al contrario si riesce ad inviare, si procede con la terza ed ultima fase dell'algoritmo. Quest'ultima fase, prevede l'attesa (un tempo fisso preimpostato) della ricezione dell'ack da parte del ricevitore. La ricezione dell'ack coincide con l'avvenuta ricezione dell'interrupt che inizialmente è stato impostato per essere eseguito nel momento in cui si verifica la modifica del registro TX_DS(a differenza della modalità shork burst,il registro non verrà modificato nel momento dell'invio ma bensì alla ricezione dell'ack).
Nel caso in cui entro il tempo stabilito viene ricevuto l'ack, il pacchetto viene considerato inviato con successo, l'algoritmo giunge al termine e si procede col dato successivo.
Se invece non viene ricevuto nessun ack allo scadere del delay, ci si potrebbe ritrovare in una delle seguenti situazioni:

- Pacchetto perso

- Ack perso
- Collisione (del pacchetto o dell'ack)

Qualunque sia il caso in cui ci troviamo, la situazione viene gestita nel medesimo modo tramite un algoritmo di backoff esponenziale. Il procedimento prevede che ogni volta che viene verificata una collisione, si attende un intervallo di tempo proporzionale al numero di collisioni verificatesi fino a quel momento prima di tentare il reinvio (per un totale massimo di 16 tentativi). Solitamente tale algoritmo viene utilizzato in situazioni in cui i sistemi sono di tipo slotted time, scegliendo un tempo di attesa random di M slot temporali compresi tra 1 e 2^N(con N numero di collisioni). Nel nostro caso, verrà utilizzata la stessa formula ma invece di utilizzare slot temporali fissi, verrà utilizzato come nel caso precedente un intervallo random all'interno di un range di valori.

```
uint8_t waitAck()
      {
        uint8_t inserted=0;

        while(collisions!=16)
        {
          MIOS32_DELAY_Wait_uS(waitAck_delay);

       if(ack==1)
       {
        cleanAndRestart(1);
        return 0;
       }
       else
       {
        CE_L();

        if(inserted==0)
        {
            SPI2_readWriteReg(FLUSH_TX,0);
            SPI2_writeStruct(getPacketElem());
        }

        srand(n_seed);
        n_seed=next_seed(n_seed);

        // BACKOFF ESPONENZIALE
        collRand=rand()%(2^(collisions+1));
        MIOS32_DELAY_Wait_uS(collision_delay*collRand);

        inserted=checkRetrasmission();
       }

       collisions++;

      }

      cleanAndRestart(2);
      return 0;

      }
```

5.3 I test

I test di valutazione del protocollo prevedono il calcolo medio dei seguenti parametri, al variare della velocità media di generazione dei pacchetti e dei tempi di ritrasmissione:

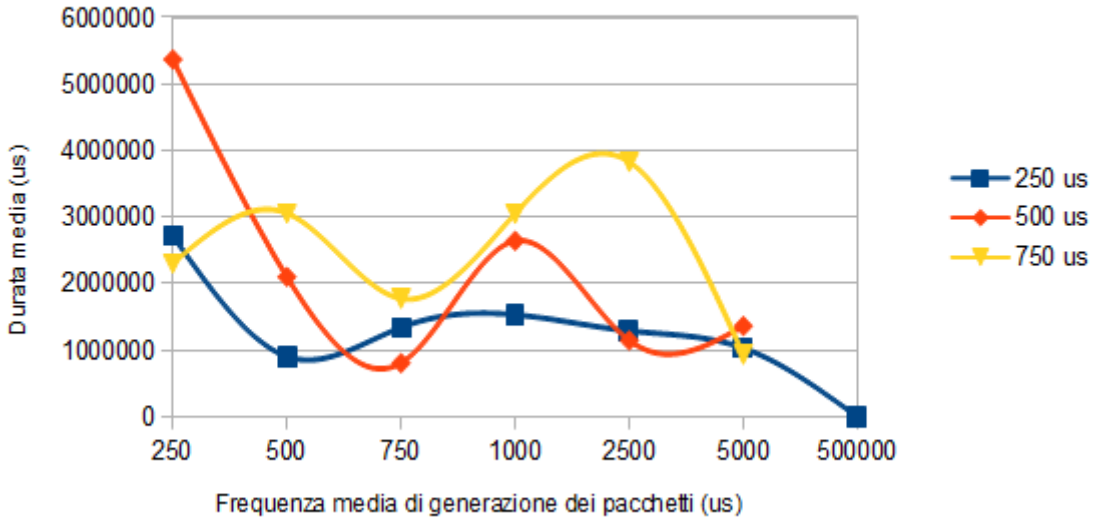
- tempi generazione-gestione del pacchetto (dalla creazione all'avvio della fase di invio)
- tempi generazione-invio
- tempi generazione-ricezione ack e throughput associato
- tempi inizio gestione-invio
- tempi inizio gestione-ricezione ack e throughput associato
- tempi invio-ricezione ack e throughput associato
- numero medio di tentativi di ritrasmissione
- numero medio di collisioni
- percentuale di pacchetti correttamente inviati

Per effettuare tali test, è stato creato un file in cui vengono inizializzate le strutture che conserveranno i dati sotto forma di timestamp. Il sistema prevede la possibilità di attivare e disattivare la modalità di test, tramite la variabile DEBUG(1=debug attivato,0=debug disattivato) nel file "init.h".
Il timestamp viene generato tramite due periferiche timer(TIM6 e TIM3) inizializzate nel file "stopwatch.c". Dato che ogni timer può essere impostato con una risoluzione massima di 1 microsecondo e riesce a contare fino ad un massimo di 65535(0xFFFF , 16bit) sono stati inizializzati 2 diversi contatori, il primo che conta da 0 a 65535 con frequenza di un microsecondo e che successivamente si resetta e comincia nuovamente da 0, e il secondo che tramite un interrupt generato ogni 65535 microsecondi incrementa di uno una variabile count. La somma di entrambi i valori ottenuti dai contatori (con count moltiplicato per 65535) ci permetterà di avere un timestamp interno con una buona precisione,tramite il quale saranno effettuati i test.

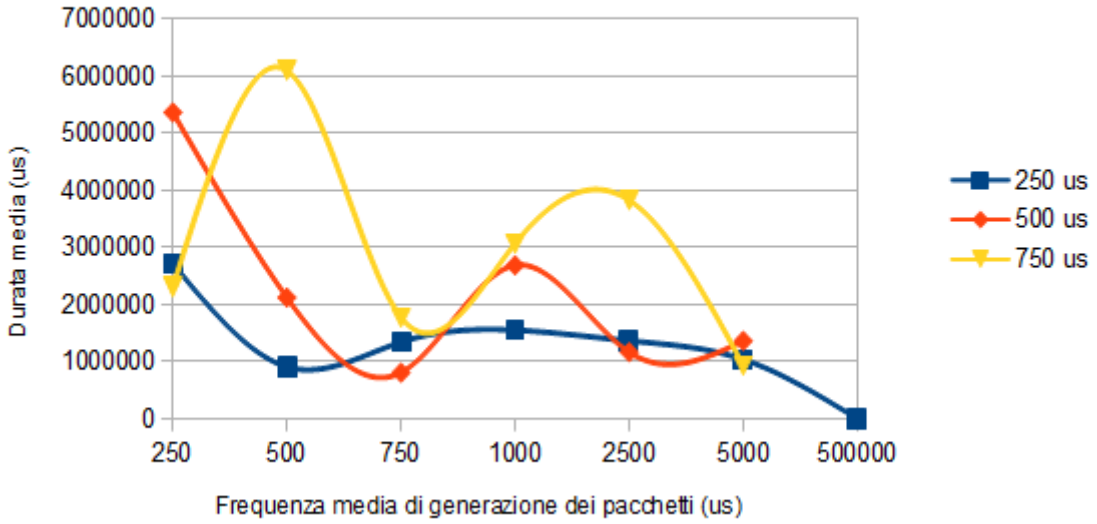
I seguenti grafici mostrano i risultati ottenuti eseguendo i test con una configurazione di 3 trasmettitori ed un ricevitore, posti ad una distanza di circa 20 metri ciascuno in ambiente interno rumoroso. I dispositivi sono stati inoltre configurati nella modalità ad 1Mbps ed i pacchetti inviati avevano una dimensione di 153bit ciascuno(1byte preambolo, 5byte indirizzo, 10byte payload, 9bit campo di controllo, 2byte CRC).

Tutti i grafici mostrano tre curve, ognuna delle quali caratterizza una diversa durata media di attesa (espressa in microsecondi) tra un tentativo di ritrasmissione ed un altro (sia nel caso di ritrasmissione iniziale che nel caso di collisioni, da applicare al backoff come spiegato precedentemente)

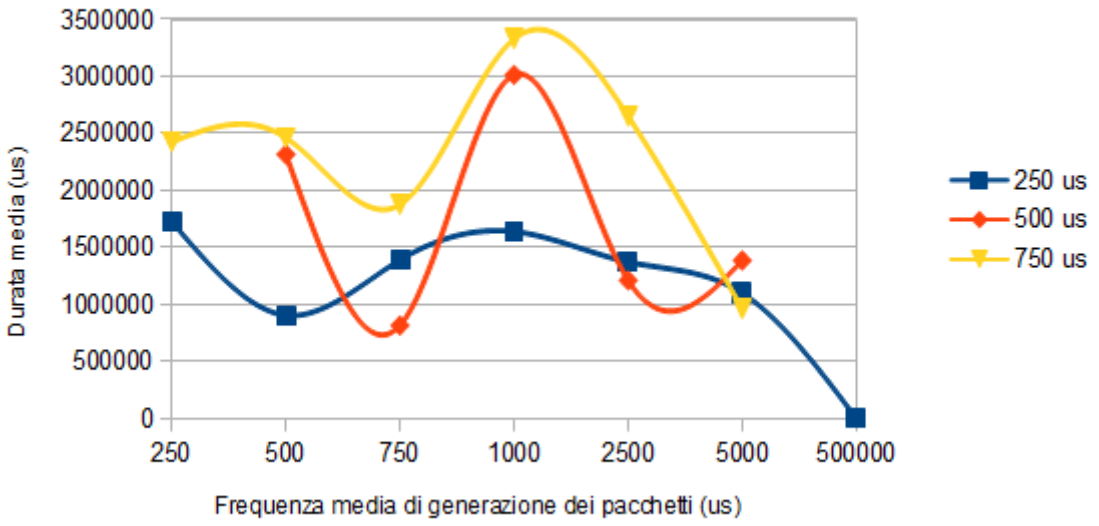
Ritardo medio tra generazione e gestione del pacchetto



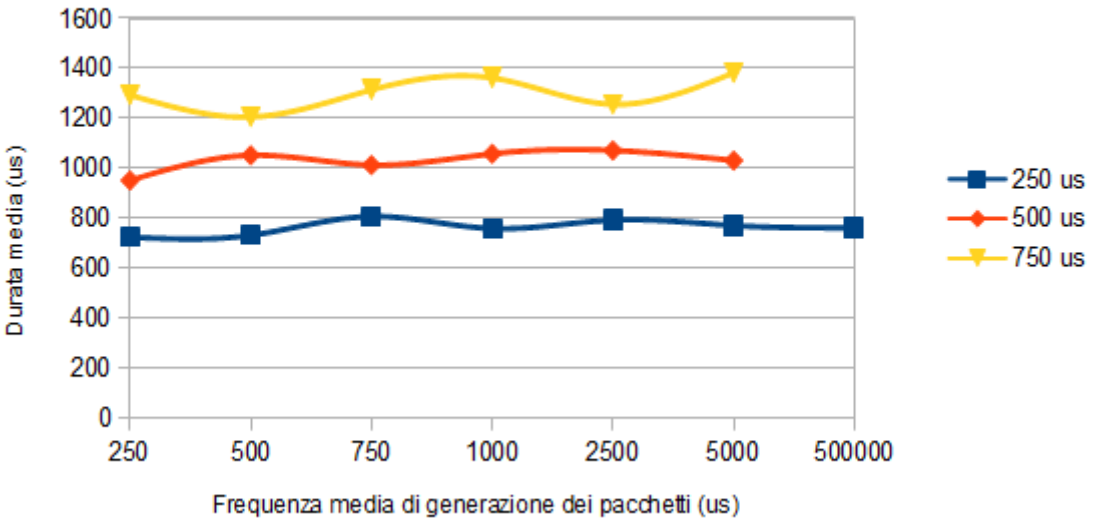
Ritardo medio tra generazione del pacchetto e invio

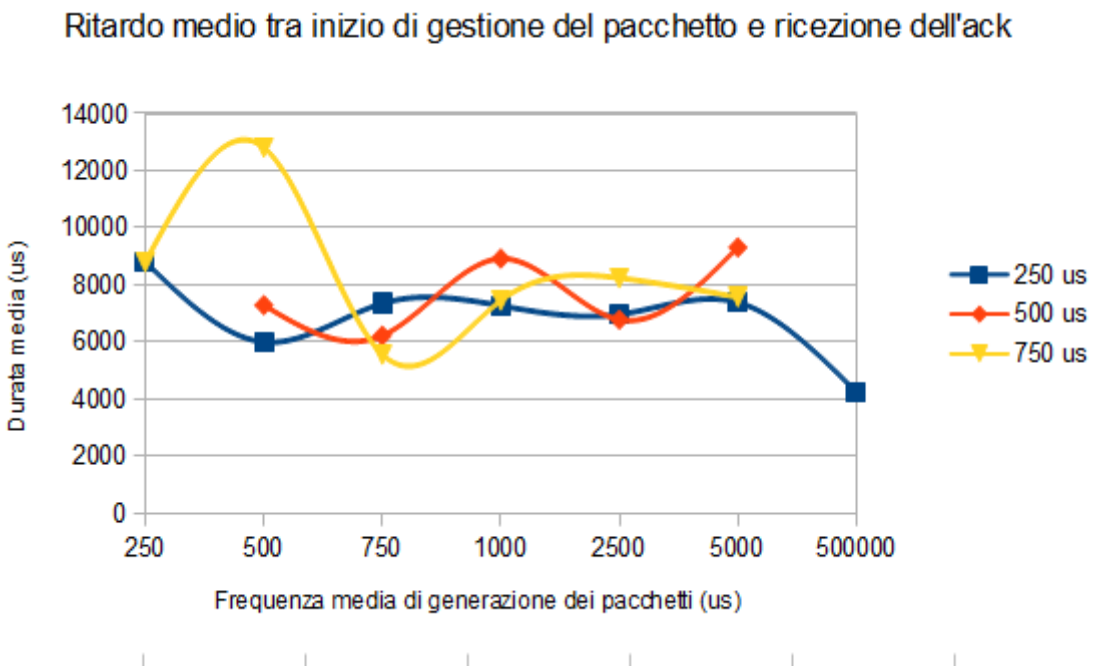


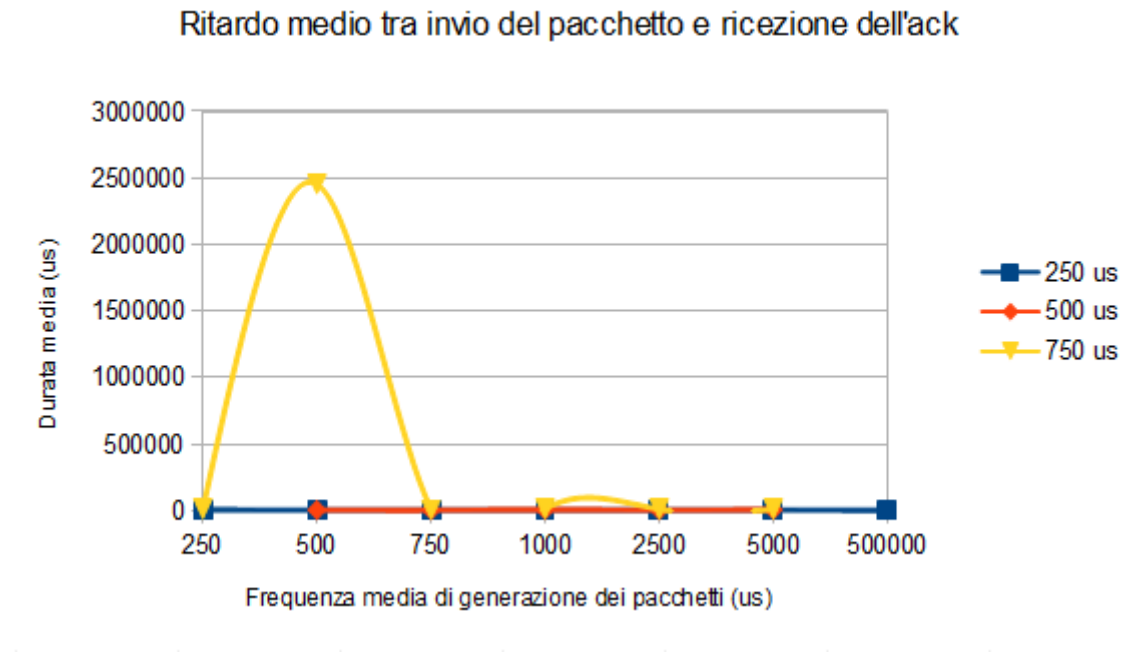
Ritardo medio tra generazione del pacchetto e ricezione dell'ack

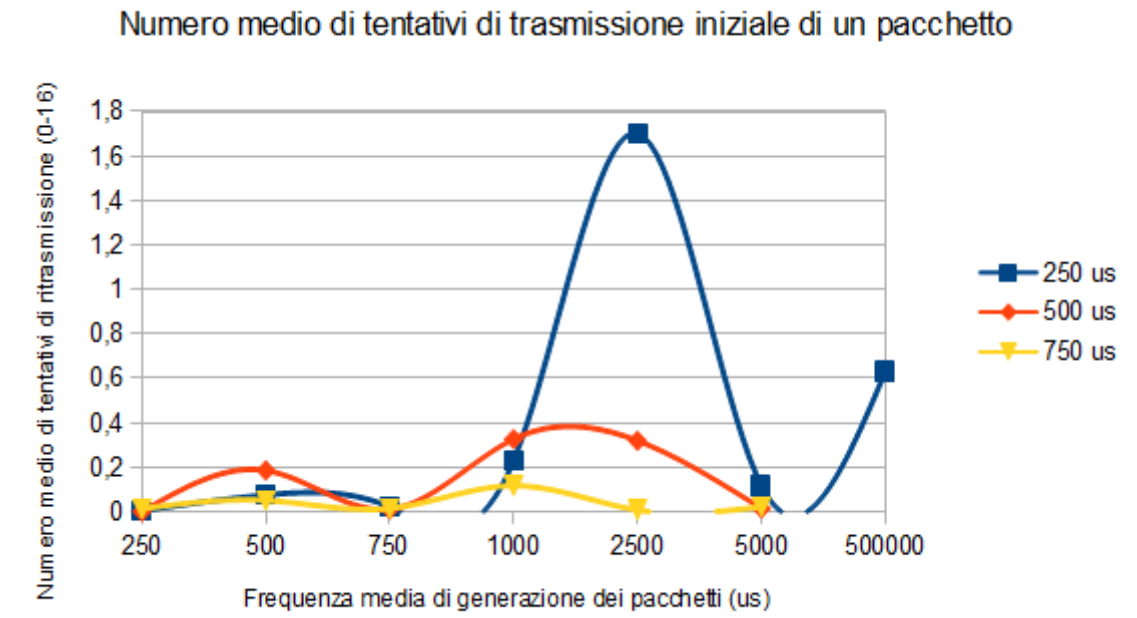


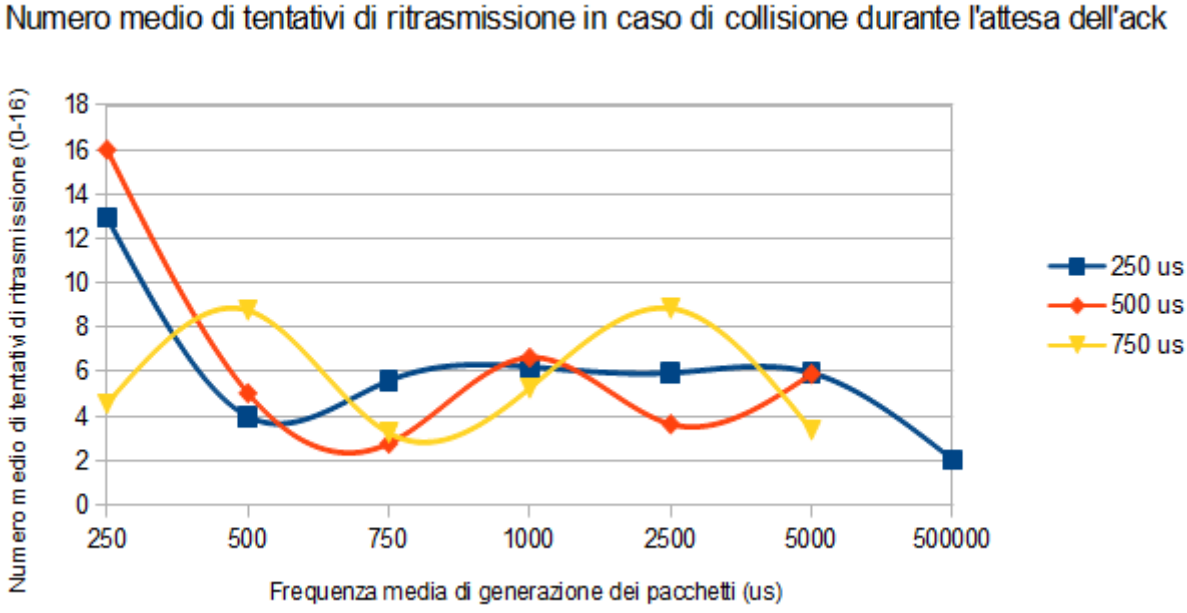
Ritardo medio tra inizio di gestione del pacchetto ed invio

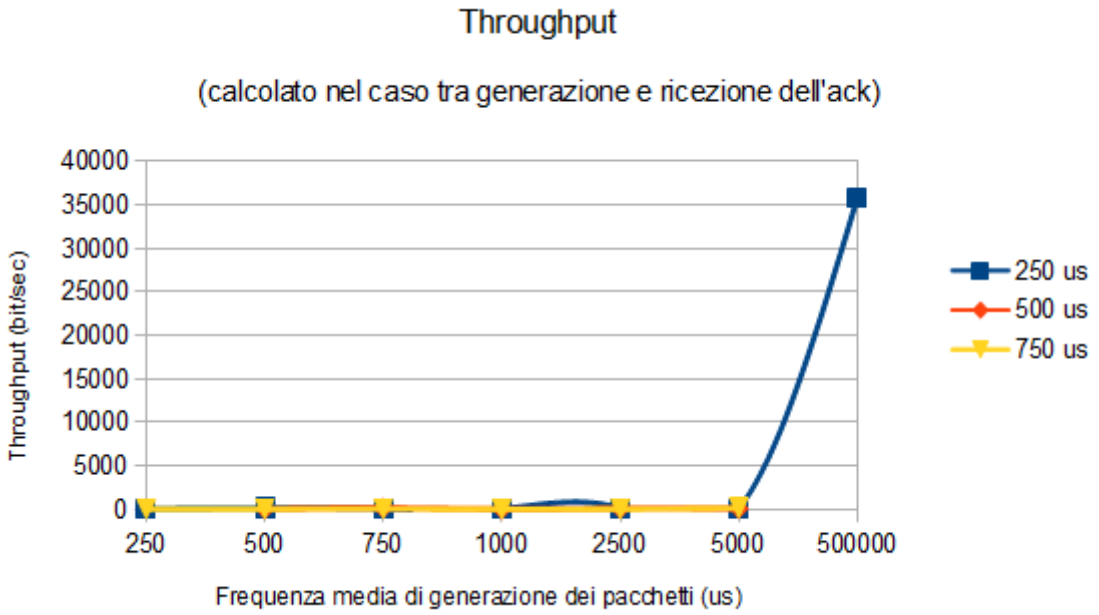


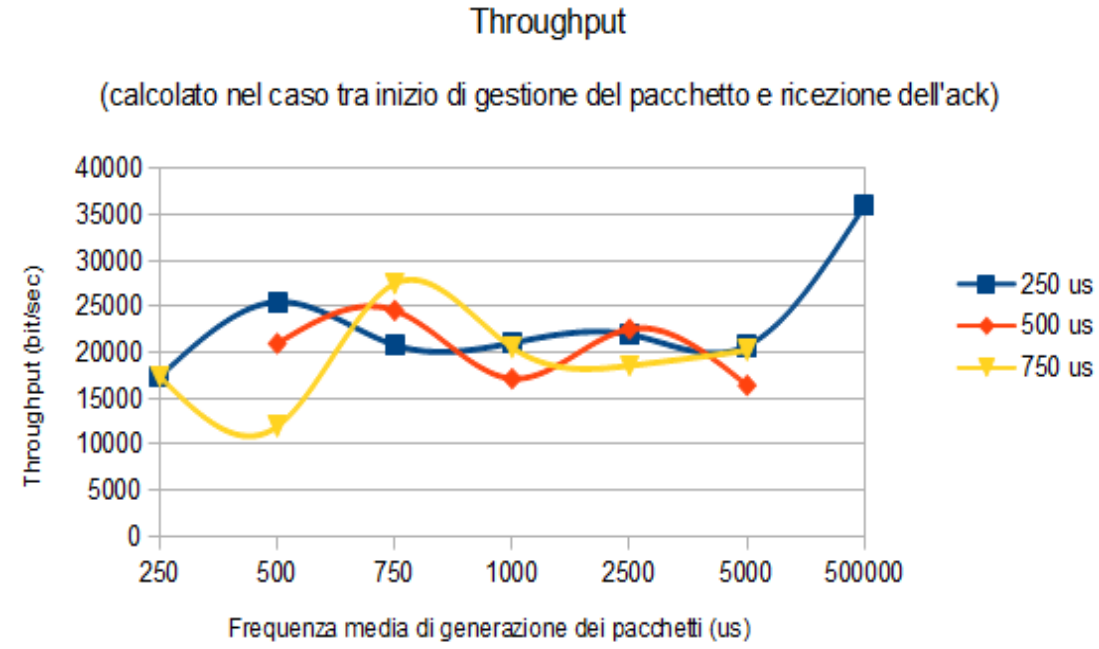

Ritardo medio tra inizio di gestione del pacchetto e ricezione dell'ack
Durata media (us)
14000
12000
10000
8000
6000
4000
2000
0
250
500
750
1000
2500
5000
500000
Frequenza media di generazione dei pacchetti (us)
250 us
500 us
750 us

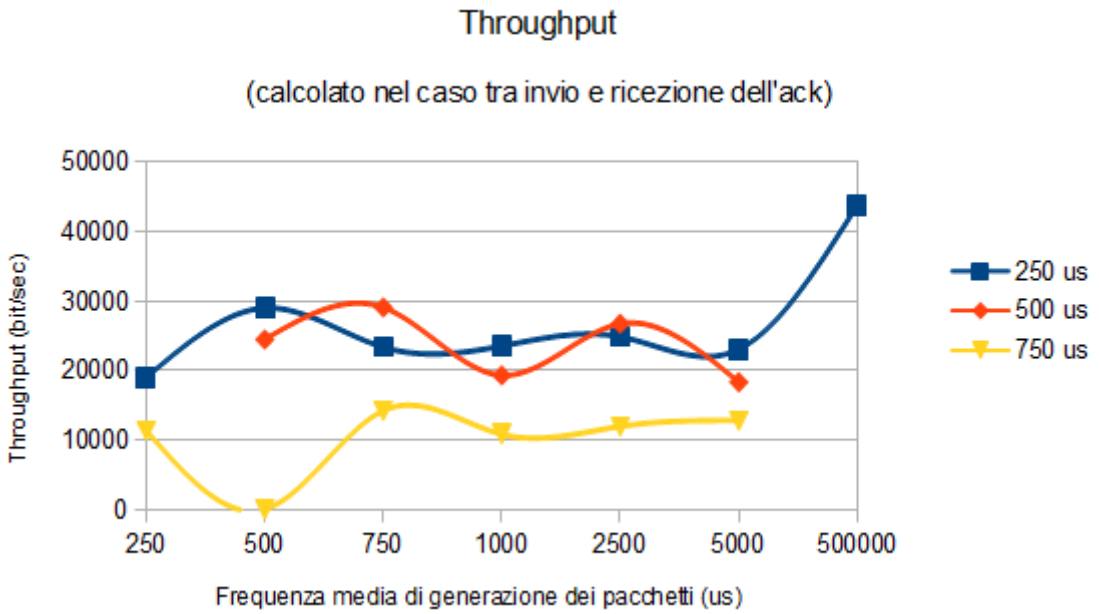

Ritardo medio tra invio del pacchetto e ricezione dell'ack
Durata media (us)
3000000
2500000
2000000
1500000
1000000
500000
0
250
500
750
1000
2500
5000
500000
Frequenza media di generazione dei pacchetti (us)
250 us
500 us
750 us

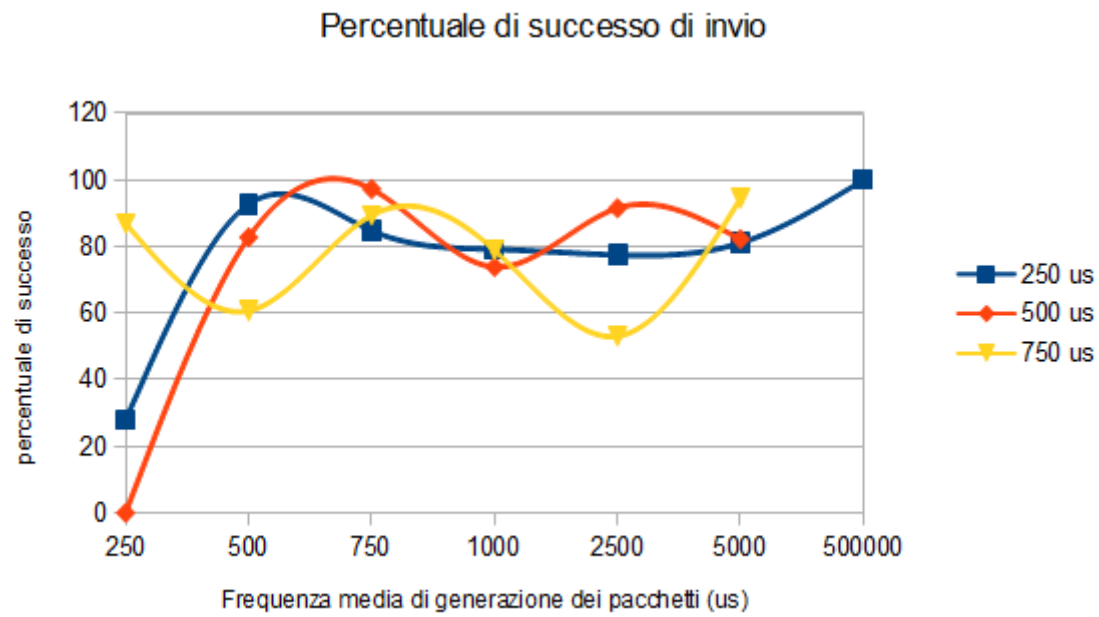

Numero medio di tentativi di trasmissione iniziale di un pacchetto
Numero medio di tentativi di ritrasmissione (0-16)
1,8
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
250
500
750
1000
2500
5000
500000
Frequenza media di generazione dei pacchetti (us)
250 us
500 us
750 us


Numero medio di tentativi di ritrasmissione in caso di collisione durante l'attesa dell'ack
Numero medio di tentativi di ritrasmissione (0-16)
18
16
14
12
10
8
6
4
2
0
250
500
750
1000
2500
5000
500000
Frequenza media di generazione dei pacchetti (us)
250 us
500 us
750 us


Throughput
(calcolato nel caso tra generazione e ricezione dell'ack)
Throughput (bit/sec)
40000
35000
30000
25000
20000
15000
10000
5000
0
250
500
750
1000
2500
5000
500000
Frequenza media di generazione dei pacchetti (us)
250 us
500 us
750 us


Throughput
(calcolato nel caso tra inizio di gestione del pacchetto e ricezione dell'ack)
Throughput (bit/sec)
40000
35000
30000
25000
20000
15000
10000
5000
0
250
500
750
1000
2500
5000
500000
Frequenza media di generazione dei pacchetti (us)
250 us
500 us
750 us





In conclusione possiamo dire che, nonostante i risultati ottenuti non hanno soddisfatto del tutto le tempistiche attese, ed hanno mostrato risultati diversi rispetto ai tempi medi calcolati in fase di caratterizzazione del dispositivo, bisogna considerare che l'ambiente in cui sono stati eseguiti i test risultava essere abbastanza rumoroso. Bisogna ricordare inoltre che i test iniziali erano stati effettuati in modalità shock burst, mentre per lo sviluppo del protocollo ci siamo basati sulla modalità enanched shock burst. In questa modalità, come abbiamo visto, verrà sicuramente introdotto un overhead dovuto alle costanti transazioni dei dispositivi da ricevitore a trasmettitore e viceversa al fine di inviare e ricevere l'ack, e al calcolo e la verifica di CRC e campo di controllo. Come abbiamo già notato, si aggiunge anche un ulteriore fattore di overhead, dovuto al fatto che per

eseguire la lettura della portante, dobbiamo effettuare due ulteriori switch tra le due modalità (TX → RX → TX). Infine, vista l'impossibilità di far partire tutti i nodi contemporaneamente , potrebbe capitare che il nodo su cui vengono effettuati i test inizi a trasmettere in un istante in cui la rete sia già in fase di saturazione. Si può quindi concludere che effettuando i test in condizioni diverse i risultati potrebbero mostrare netti miglioramenti rispetto a quelli da noi ottenuti, nonostante siano stati effettuati più volte i test cercando di coprire situazioni più eterogenee possibili al fine di simulare delle condizioni reali di utilizzo.

Referenze/Link utili

[1] http://www.st.com/jp/evalboard/product/253215.jsp
[2] http://www.nordicsemi.com/eng/Products/2.4GHz-RF/nRF24L01
[3] http://www.iar.com/en/Products/IAR-Embedded-Workbench/
[4] https://sites.google.com/site/terminalbpp/
[5] http://www.nordicsemi.com/eng/Products/2.4GHz-RF/nRF24L01 pagina 13
[6] http://www.nordicsemi.com/eng/Products/2.4GHz-RF/nRF24L01 da pagina 53 a pagina 57
[7] http://www.nordicsemi.com/eng/Products/2.4GHz-RF/nRF24L01 pagina 46
[8] http://www.nordicsemi.com/eng/Products/2.4GHz-RF/nRF24L01 da pagina 35 a 38