ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA

SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA IN INGEGNERIA ELETTRONICA PER L'ENERGIA E L'INFORMAZIONE

TITOLO TESI:

PROGETTO SOFTWARE/FIRMWARE DI UN'INTERFACCIA PER ACQUISIZIONE DATI DA UN NODO SENSORE BASATO SU MICROCONTROLLORE

ELABORATO IN:

ELETTRONICA DEI SISTEMI DIGITALI

Relatore:

Prof. **Aldo Romani**

Presentata da:

Pastore Cesare

Correlatore

Dott. **Matteo Pizzotti**

ANNO ACCADEMICO 2015/2016

SESSIONE III

Indice

Introduzione

Nei circuiti elettronici, il costante aumento di integrazione fra la parti analogiche e parti di elaborazione/comunicazione digitale, specialmente nell'ambito della sensoristica, vara nuove sfide nella progettazione, sia del circuito stesso che dell'ambiente di test dedicato. In particolare, questo elaborato si concentra su quest'ultimo aspetto, relativo alla progettazione di interfacce di comunicazioni con sensori prototipali.

L'oggetto di studio è un sensore sperimentale realizzato all'interno del gruppo di ricerca del dipartimento DEI presso la sede di Cesena che, accoppiato ad un opportuno oscillatore elettromeccanico, ad esempio un elemento MEMS, è in grado di calcolarne il periodo di oscillazione con una precisione elevata. Molteplici sono le applicazioni che si possono adattare ad un sensore di questo tipo, come ad esempio un estensimetro ad alta precisione.

L'obiettivo di questo elaborato è lo studio delle modalità di comunicazione tra nodo sensore e un calcolatore, tramite un microcontrollore.

Il microcontrollore scelto è il PIC16F1823 della Microchip, fornitore leader di semiconduttori analogici e microcontrollori, che possiede anche un ambiente di sviluppo ben collaudato: MPLAB-X IDE, ultima versione di MPLAB IDE rilasciato dalla Microchip Tecnology. Basato su piattaforma NetBeans open-source, supporta l'editing, il debugging e la programmazione di microcontrollori PIC a 8bit, 16bit e 32bit. Il linguaggio di programmazione che supporta è il C.

L'obiettivo finale sarà quello di leggere e interpretare il messaggio di un utente, tramite interfaccia grafica, per poter trasmettere o ricevere dati al sensore tramite microcontrollore.

1. Descrizione del sistema

Come abbiamo poc'anzi introdotto, il progetto del sistema prevede un microcontrollore che possa interfacciare tra loro un sensore e un calcolatore e che supporti i protocolli UART e SPI perché, come possiamo notare dalla figura sottostante, da un lato il microprocessore è interfacciato a un calcolatore mediante un cavo seriale UART, dall'altra acquisisce e/o invia dati a un sensore, a seconda della configurazione letta su appositi registri, tramite l'utilizzo di un protocollo SPI.

Il PIC16F1823 è stato scelto per alcuni vantaggi. Prima di tutto perché supporta entrambi i protocolli di comunicazione sopra citati. È un dispositivo a 14 pin. Scelta non indifferente, dato che il protocollo UART ne utilizzerà due, mentre il protocollo SPI ne utilizzerà quattro, senza considerare i pin di alimentazione, massa e di programmazione (ossia altri cinque). Ha un range di alimentazione che varia dai 1.8V ai 5.5V.

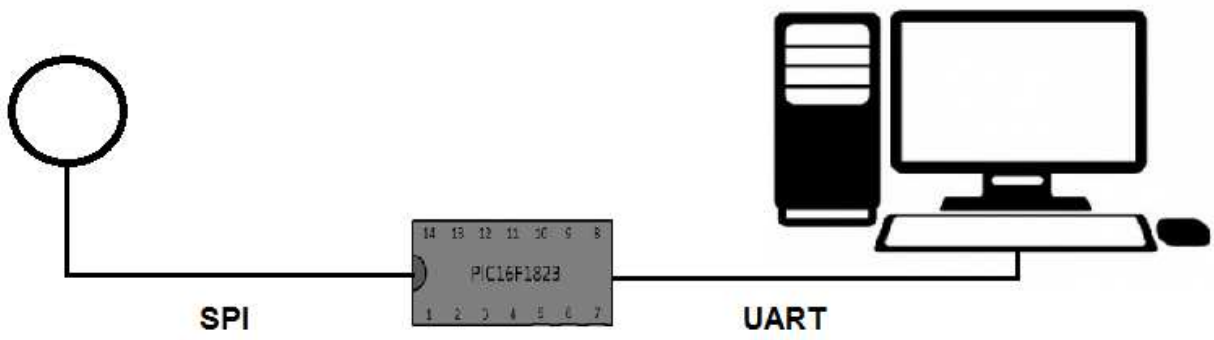


FIGURA 1 – Schema iniziale del sistema

Dal momento che questo sistema era destinato ad applicazioni Low Power, si è scelto di alimentare il microcontrollore ad una tensione di 1.8V.

Dall'altro versante, però, il calcolatore necessitava di una tensione di 5V per poter comunicare tramite UART. È stato così necessario utilizzare un traslatore di livello per permettere la comunicazione tra i due dispositivi.

Per quest'ultimo, sono state individuate due possibili soluzioni:

- Tramite un semplice traslatore di livello seriale.
- Tramite un adattatore USB-RS232 capace di emulare la porta seriale.

La scelta è ricaduta sulla seconda soluzione, sia per un minore sforzo a livello circuitale, sia perché lo standard USB ha ormai sostituito la comunicazione seriale nei Personal Computer.

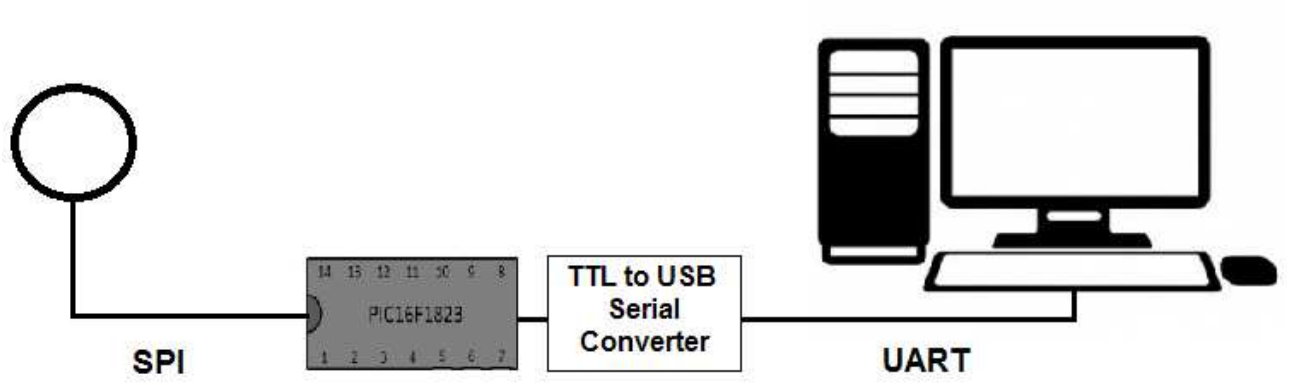
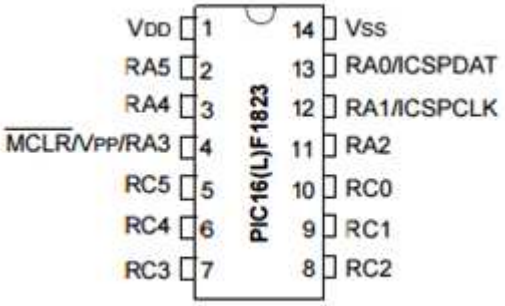


FIGURA 2 – Schema finale del sistema

1.1 Il Microcontrollore PIC16F1823

Il dispositivo oggetto della tesi è il PIC16F1823, facente parte della famiglia PIC16 di microcontrollori a 8bit prodotti dalla Microchip. Questi dispositivi hanno un'architettura di tipo RISC (Reduced Instruction Set Computer), ovvero sono in grado di svolgere tutte le loro funzioni facendo affidamento a un set ridotto di istruzioni.



I/O	14-Pin PDIP/SOIC/TSSOP	16-Pin QFN/UQFN	A/D	Reference	Cap Sense	Comparator	SR Latch	Timers	ECCP	EUSART	MSSP	Interrupt	Modulator	Pull-up	Basic
RA0	13	12	AN0	DACOUT	CPS0	C1IN+	—	—	—	TX(1) CK(1)	—	IOC	—	Y	ICSPDAT ICDDAT
RA1	12	11	AN1	VREF+	CPS1	C12IN0-	SRI	—	—	RX(1) DT(1)	—	IOC	—	Y	ICSPCLK ICDCLK
RA2	11	10	AN2	—	CPS2	C1OUT	SRQ	T0CKI	FLT0	—	—	INT/ IOC	—	Y	—
RA3	4	3	—	—	—	—	—	T1G(1)	—	—	$\overline{SS}$(1)	IOC	—	Y	$\overline{MCLR}$ VPP
RA4	3	2	AN3	—	CPS3	—	—	T1G(1) T1OSO		—	SDO(1)	IOC	—	Y	OSC2 CLKOUT CLKR
RA5	2	1	—	—	—	—	—	T1CKI T1OSI	—	—	—	IOC	—	Y	OSC1 CLKIN
RC0	10	9	AN4	—	CPS4	C2IN+	—	—	—	—	SCL SCK	—	—	Y	—
RC1	9	8	AN5	—	CPS5	C12IN1-	—	—	—	—	SDA SDI	—	—	Y	—
RC2	8	7	AN6	—	CPS6	C12IN2-	—	—	P1D	—	SDO(1)	—	MDCIN1	Y	—
RC3	7	6	AN7	—	CPS7	C12IN3-	—	—	P1C	—	$\overline{SS}$(1)	—	MDMIN	Y	—
RC4	6	5	—	—	—	C2OUT	SRNQ	—	P1B	TX(1) CK(1)	—	—	MDOUT	Y	—
RC5	5	4	—	—	—	—	—	—	CCP1 P1A	RX(1) DT(1)	—	—	MDCIN2	Y	—
VDD	1	16	—	—	—	—	—	—	—	—	—	—	—	—	VDD
VSS	14	13	—	—	—	—	—	—	—	—	—	—	—	—	VSS

Note 1: Pin function is selectable via the APFCON register.

FIGURA 3 – pin diagram and allocation table PIC16F1823

Configurando opportunamente i bit del registro APFCON (Alternate Pin Function Control Register), è possibile impostare una diversa funzionalità ad ogni pin. Per una comunicazione seriale attraverso protocollo SPI e UART, sarà necessario programmare il PIC attraverso un firmware dedicato che abiliti le funzioni del Microcontrollore.

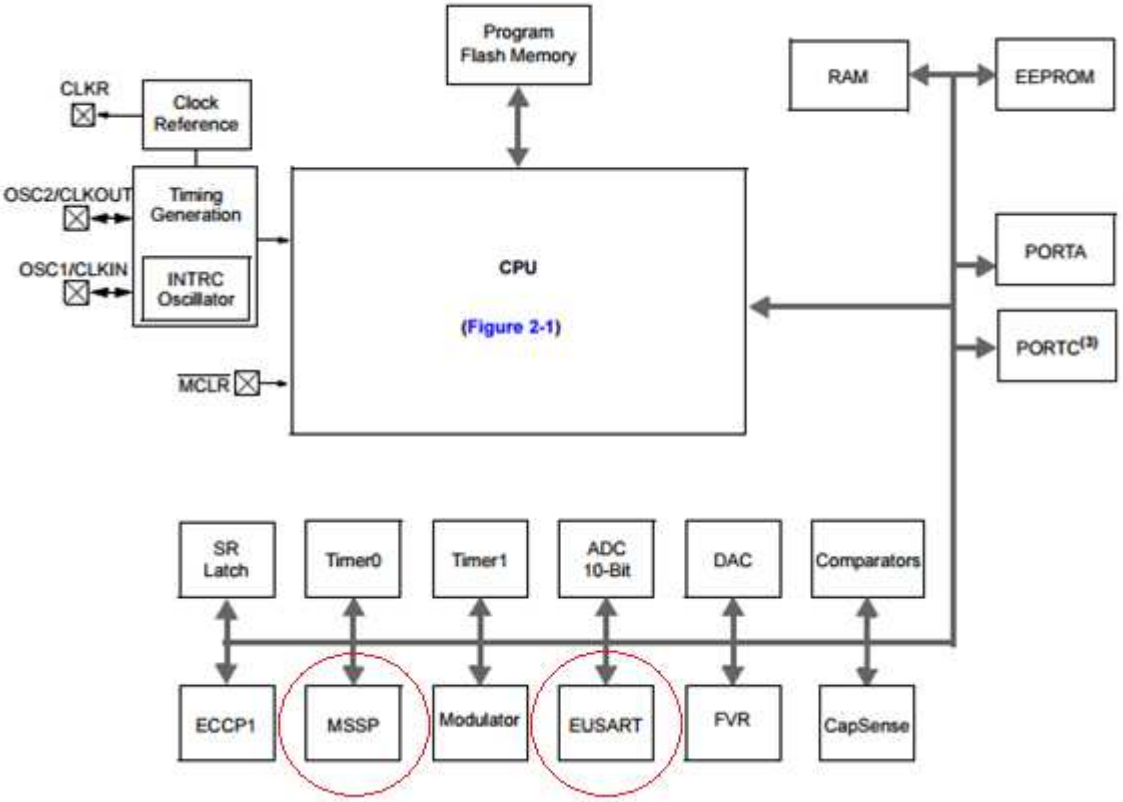


FIGURA 4 – Diagramma a blocchi sulla struttura interna del PIC

Nello schema a blocchi della figura sovrastante si evidenzia ancora una volta i protocolli seriali SPI e UART utilizzati in questa Tesi.

1.2 Il Protocollo SPI

Il protocollo SPI (Serial Peripheral Interface) è un sistema di comunicazione tra un microcontrollore e altri dispositivi esterni integrati. Ideato dalla Motorola, la sua trasmissione avviene tra un dispositivo detto "Master" e uno o più "Slave". Il Master controlla il bus, emette il segnale di clock e decide quando iniziare/terminare la comunicazione, mentre lo Slave esegue le operazioni che gli vengono imposte.

Il bus SPI è quindi di tipo:

- Sincrono, per la presenza di un clock che coordina la trasmissione e ricezione dei bit e determina la velocità di trasmissione.
- Full-duplex, in quanto la trasmissione e la ricezione possono avvenire contemporaneamente.

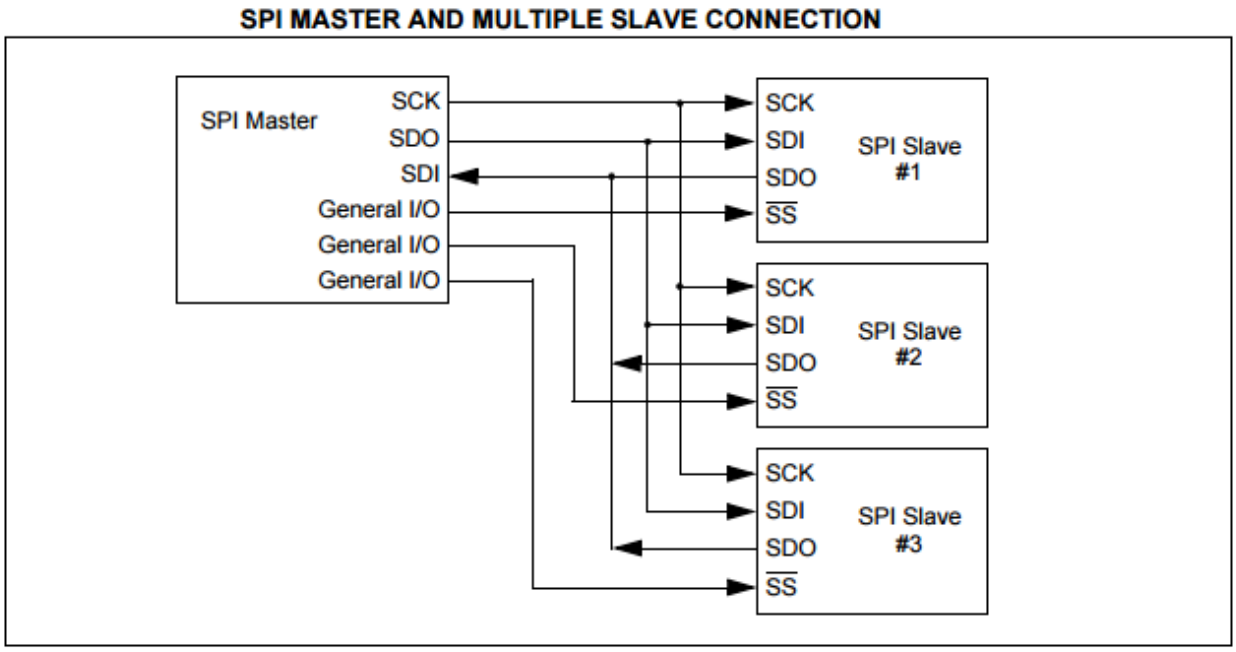


FIGURA 5 – Connessione tra un Master e più Slave

Il sistema è definito da quattro linee di comunicazione, più il collegamento di massa. I segnali in figura sono descritti in questo modo:

- SCLK – SCK: Serial Clock, emesso dal master.
- SDI – MISO: Serial Data Input, Master Input Slave Output
- SDO – MOSI: Serial Data Output, Master Output Slave Input
- CS – SS: Chip Select, Slave Select, emesso dal master per decidere con quale dispositivo vuole comunicare.

1.2.1 La Comunicazione

La trasmissione dei dati sul bus SPI si basa sul funzionamento a "Shift Register". Ogni dispositivo, sia Master che Slave, è dotato di un registro a scorrimento interno, i cui bit vengono emessi e contemporaneamente immessi tramite l'uscita SDO – MOSI e l'ingresso SDI – MISO. Il registro in questione ha dimensione di 8 bit. Esso è un'interfaccia mediante la quale vengono trasmessi comandi e dati in modo seriale, ma che internamente sono riconvertiti in modo parallelo.

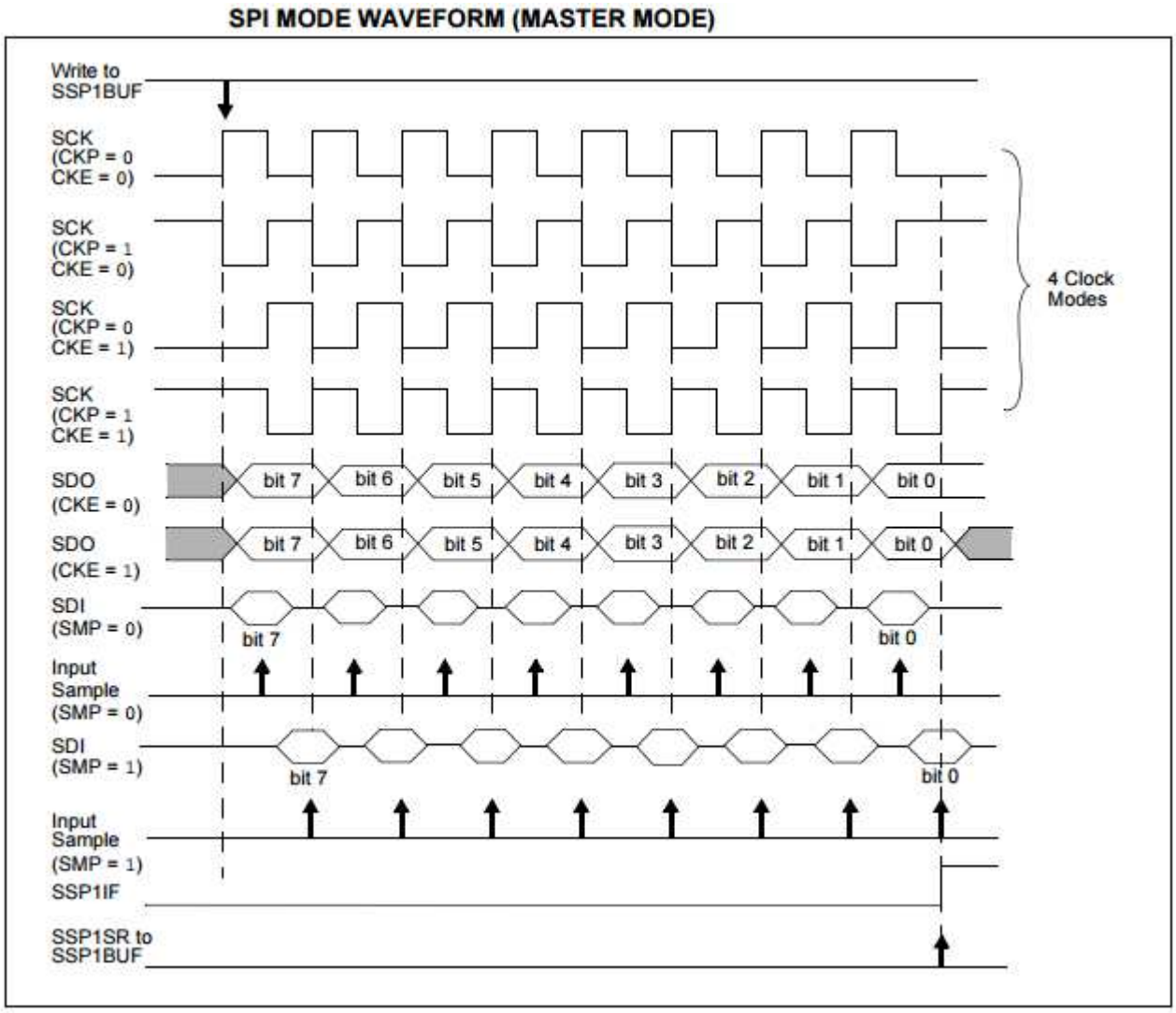


FIGURA 6 – Forme d'onda in modalità Master

Ad ogni impulso di clock i dispositivi che stanno comunicando, trasmettono un bit dal loro registro interno, rimpiazzandolo con un bit emesso dall'altro interlocutore. Come da figura, la sincronizzazione è fatta sui fronti di salita o discesa del clock, regolata dai parametri CKE, CKP, SMP. In base a come imposto questi bit, si sceglie una diversa polarità del clock, si regola la fase e si imposta se campionare alla fine o al centro della trama di dati.

1.3 Il protocollo UART

Lo UART (Universal Asynchronous Receiver-Trasmitter) è un protocollo che converte flussi di bit di dati da un formato parallelo a un formato seriale asincrono o viceversa. Ogni famiglia di microprocessori ha la sua UART dedicata.

Come è stato appena detto, la trasmissione è asincrona, per cui non è presente alcuna trasmissione del clock. Il trasmettitore e il ricevitore dovranno accordarsi sulla velocità di trasmissione (Baud Rate) e sulla struttura della trama prima di procedere.

1.3.1 La trama

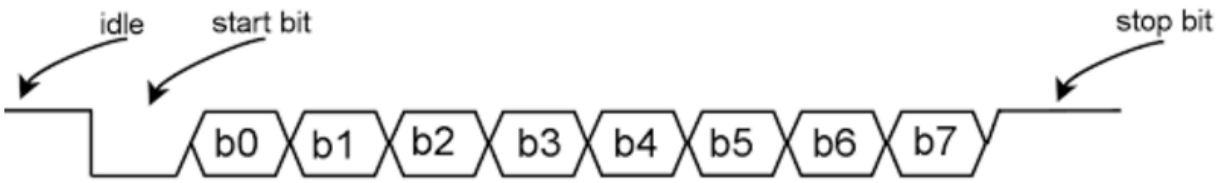


FIGURA 7 – Trama protocollo UART

Come possiamo vedere dalla figura, la trama di bit trasmessi è composta da:

- IDLE: rappresenta lo stato in cui non c'è nessun trasferimento. La linea rimane alta.
- START BIT: indica al ricevitore l'inizio di trasmissione della trama. La linea viene posta a livello logico basso per un tempo parti al tempo di bit.
- TRAMA: contiene gli 8 bit da trasmettere, che vengono inviati sequenzialmente dal meno al più significativo.
- PARITY BIT: bit opzionale che verifica la correttezza nella trasmissione della trama. Può essere pari o dispari.
- STOP BIT: indica al ricevitore la fine della trama. Possono essere uno o due gli stop bit e la linea viene posta a livello logico alto per un tempo pari ad uno o due tempi di bit.

Per questa tesi è stata scelta la seguente configurazione di bit:

- Baud Rate = 9600bps
- Trama = 8 bit
- Parità assente
- Stop bit = 1

1.4 Collegamenti Hardware

Descriviamo ora tutti i collegamenti fisici fatti per inizializzare il sistema:

- Per quanto riguarda l'UART, il cavo TTL-232R.3V3 avrà tre dei sei pin collegati al PIC, ossia TXD, RXD e GND. Il primo è stato collegato al pin 6 (RC4) del PIC. Il secondo è stato collegato al pin 5 (RC5) del PIC, e ovviamente l'ultimo è stato collegato alla massa.
- I pin 1 e 14 del PIC sono stati collegati rispettivamente a VDD (1.8V) e massa.
- I pin 10 (RC0), 9 (RC1) e 8 (RC2) sono interamente dedicati a gestire i segnali di CLK, MISO e MOSI dell'interfaccia SPI, verso il sensore.
- Il pin 11 (RA2), ausiliario, è stato usato come Chip–Enable, sempre dall'SPI.
- Il pin 3 (RA4), anch'esso ausiliario, è stato usato da un led, per una comunicazione di errore.
- Il pin 7 (RC3) è stato usato come Start Bit, ossia colui che comunicherà l'inizio della trasmissione dati.
- Il pin 4 (RA3) è stato usato come Data Ready, ossia colui che comunicherà l'avvenuta ricezione della trama.
- Il pin 2 (RA5) è stato usato come Reset, poiché il Micro necessità di esso di tanto in tanto.
- Per ultimo, i pin 13 (RA0), 12 (RA1) e 4 (RA3) sono stati usati dal PICKit3 per configurare e quindi programmare il Micro.

1.4.1 Il PICKit3

Come debugger/programmer è stato usato il Pickit3, prodotto dalla Microchip.

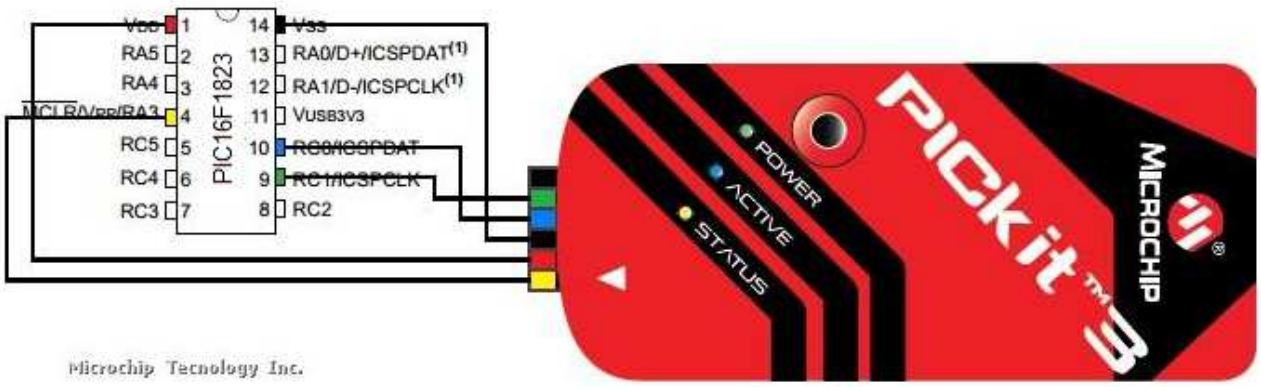


FIGURA 8 – Collegamenti tra il PIC e il PICKit3

Il principio di funzionamento è molto semplice. Come possiamo vedere nella figura soprastante, ogni singolo pin del dispositivo è stato collegato ad un determinato pin del PIC, come descritto in precedenza. Al tempo stesso esso viene collegato al PC tramite cavo USB sia per ricevere l'alimentazione necessaria al suo funzionamento, sia per poter trasmettere al PIC il firmware di programmazione.

1.4.2 Strumentazione

Per fornire alimentazione all'intero circuito, montato su breadboard, è stato usato un alimentatore Agilent E3630A. Tramite due cavi è stata fornita una tensione di 1.8V ai capi del PIC.

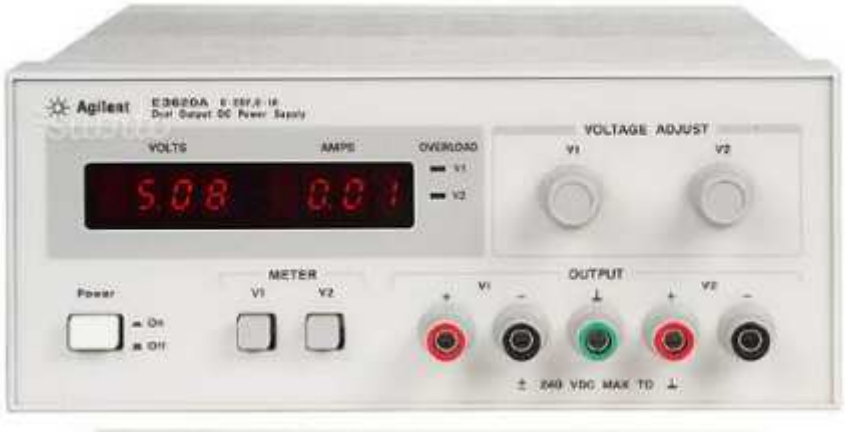


FIGURA 9 – Immagine dell' alimentatore usato in esame

Per poter leggere e quindi accertarsi di ogni passaggio corretto della nostra Tesi, è stato utilizzato un oscilloscopio Tektronix TDS 1012B. Utilizzando due sonde, in dotazione dello strumento, è stato possibile testare parametri quali il Clock, i segnali TXD e RXD dell'UART e dell'SPI, la Baudrate, ecc …

FIGURA 10 – Immagine dell'oscilloscopio usato in esame

2. Il Firmware

Si passa ora a descrivere il Firmware, implementato per collegare il Microcontrollore sia verso il sensore tramite il protocollo SPI, sia verso il calcolatore tramite il protocollo UART. Come abbiamo già introdotto, come ambiente di sviluppo è stato usato MPLab X-IDE.

Dato che il sensore può essere visto come un blocco costituito da una serie di registri ad 8 bit, per implementare il sistema Full-duplex è necessario scrivere su uno di questi registri un dato proveniente dal calcolatore, oppure leggerlo. In entrambi i casi è necessario comunicare al sensore a quale dei suoi registri si voglia accedere, tramite l'invio di un byte contenente l'indirizzo.

Per riassumere, la procedura di scrittura prevede di trasmettere in sequenza prima l'indirizzo (col bit più significativo posto a 1) e in successione il dato. Viceversa in lettura, in cui la procedura prevede di inviare l'indirizzo a cui vuole accedere (col bit più significativo posto a 0) per poi restare in attesa del dato da ricevere, tramite la porta seriale.

Il sistema è strutturato in questo modo:

all'interno di un ciclo infinito, attendo la ricezione dell'indirizzo del registro interno del sensore a cui si vuole accedere. Dopo averlo ricevuto, effettuo un primo controllo sul tempo, ovvero, se sono passati più di 3s dalla prossima istruzione abilito un flag per segnalare un'anomalia. Qual ora questo non dovesse avvenire si effettuano una serie di 3 controlli:

1) Verificare che l'indirizzo proveniente dall'UART sia compreso tra in valori [0x00 - 0x10] esadecimali. Se così è, controllo se l'indirizzo contiene lo 0 o l'1 come cifra più significativa. Se contiene l'1 eseguo le procedure di lettura e scrittura dall'UART verso l'SPI, se contiene lo 0 eseguo solo la procedura di lettura verso l'SPI.
2) Se l'indirizzo proveniente dall'UART contiene il valore 0x11, attivo la procedura di reset del Micro.
3) Se l'indirizzo proveniente dall'UART contiene il valore 0x12, per prima cosa invio un impulso allo START bit, dopodiché avvio la procedura di lettura degli indirizzi 0x03 fino a 0x08 compresi e tramite la procedura di scrittura li invio all' UART.

La figura sottostante mostra uno schema a blocchi sul funzionamento del firmware appena escritto:

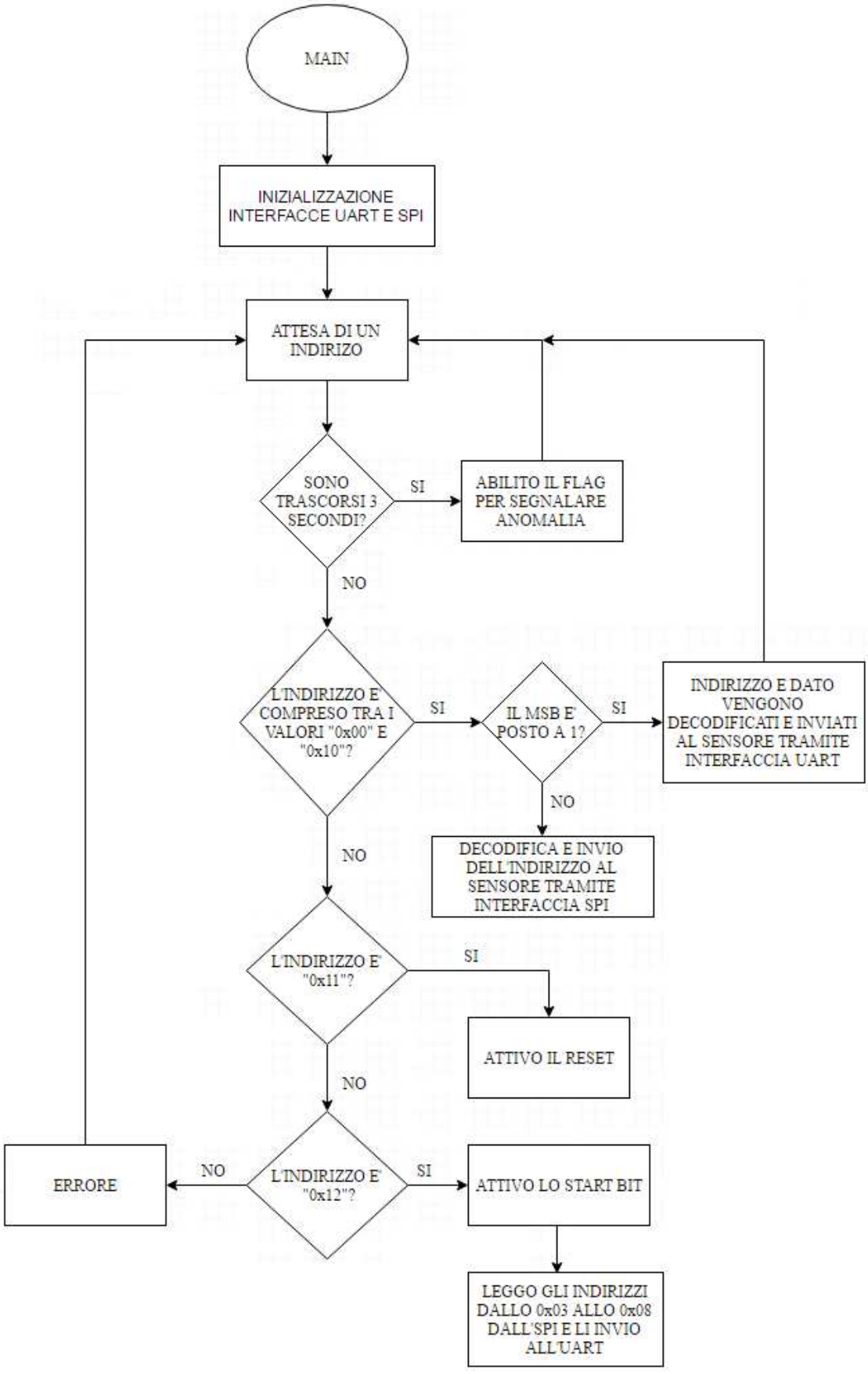


FIGURA 11 - schema a blocchi Firmware

Per quanto riguarda la frequenza di clock, il valore è di 8MHz ma grazie al modulo PLL è stato possibile portarla a 32MHz, impostato tramite il seguente indirizzo:

```
OSCCON = 0b01110000;
```

R/W-0/0	R/W-0/0	R/W-1/1	R/W-1/1	R/W-1/1	U-0	R/W-0/0	R/W-0/0
SPLLEN	IRCF<3:0>				—	SCS<1:0>	
bit 7							bit 0

Legend:		
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'
u = Bit is unchanged	x = Bit is unknown	-n/n = Value at POR and BOR/Value at all other Resets
'1' = Bit is set	'0' = Bit is cleared	

bit 7 **SPLLEN:** Software PLL Enable bit
If PLLEN in Configuration Word 1 = 1:
SPLLEN bit is ignored. 4x PLL is always enabled (subject to oscillator requirements)
If PLLEN in Configuration Word 1 = 0:
1 = 4x PLL Is enabled
0 = 4x PLL is disabled

bit 6-3 **IRCF<3:0>:** Internal Oscillator Frequency Select bits
000x = 31 kHz LF
0010 = 31.25 kHz MF
0011 = 31.25 kHz HF(1)
0100 = 62.5 kHz MF
0101 = 125 kHz MF
0110 = 250 kHz MF
0111 = 500 kHz MF (default upon Reset)
1000 = 125 kHz HF(1)
1001 = 250 kHz HF(1)
1010 = 500 kHz HF(1)
1011 = 1 MHz HF
1100 = 2 MHz HF
1101 = 4 MHz HF
1110 = 8 MHz or 32 MHz HF(see **Section 5.2.2.1 "HFINTOSC"**)
1111 = 16 MHz HF

bit 2 **Unimplemented:** Read as '0'

bit 1-0 **SCS<1:0>:** System Clock Select bits
1x = Internal oscillator block
01 = Timer1 oscillator
00 = Clock determined by FOSC<2:0> in Configuration Word 1.

FIGURA 12 – Datasheet del registro OSCCON

3. Sviluppo Interfaccia SPI

Dato che il protocollo SPI è sincrono, quindi dipende dal tempo, per poter funzionare al meglio necessita di alcune inizializzazioni che mostrerò immediatamente.

3.1 Inizializzazione

Per prima cosa, bisogna abilitare l'ENABLE, il CLOCK, il MISO e il MOSI come ingressi o come uscite sui pin del Microcontrollore. Il registro TRISC, ad 8 bit attivi alti, mi ha permesso di impostare la seguente configurazione:

```
//setto le porte
TRISC2 = 0; // MOSI
TRISC0 = 0; // SCK
TRISC1 = 1; // MISO
TRISA2 = 0; // EN
```

FIGURA 13 – Settaggio delle porte

Dato che nel protocollo SPI siamo in modalità Master, abbiamo deciso di usare il TIMER2 come base dei tempi, settando opportunamente il registro SSP1CON1:

```
// SPI Master mode, clock = TMR2 output/2
SSPM0 = 1;
SSPM1 = 1;
SSPM2 = 0;
SSPM3 = 0;
//abilito le porte
SSPEN = 1;
SSP1IF = 0;

//TMR2 is off
TMR2ON = 0;

//prescaler is 1:64
T2CKPS0 = 1;
T2CKPS1 = 1;

//postscaler 1:1
T2OUTPS0 = 0;
T2OUTPS1 = 0;
T2OUTPS2 = 0;
T2OUTPS3 = 0;

PR2 = PR2VAL/64;
//TMR2 is on
TMR2ON = 1;
```

FIGURA 14 – Settaggio TIMER2 come base dei tempi

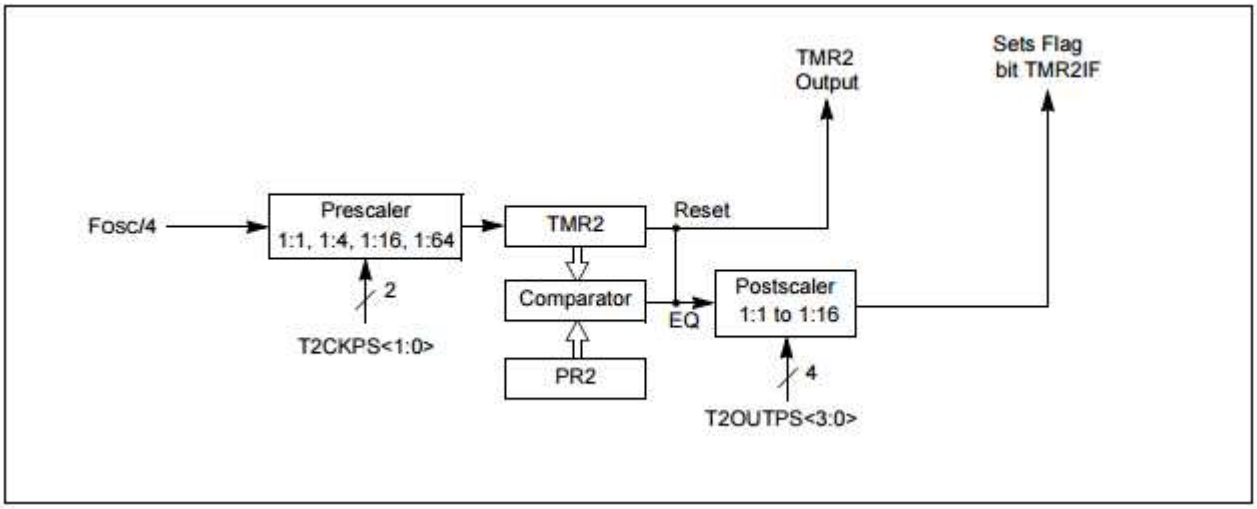


FIGURA 15 – Schema a blocchi per il settaggio del TIMER2

Come ben notiamo in questa figura, per avere il TMR2 in uscita abbiamo dovuto impostare un valore al PRESCALER. Quest'ultimo non è altro che un contatore, utilizzato per ridurre segnali ad alta frequenza in segnali a frequenza minore tramite una divisione del clock. L'operazione che ci ha permesso di impostare il valore del prescaler è la seguente:

```
#define  _CPU_FREQ 32000000
#define  _SPI_FREQ_HZ 10000
#define PR2VAL (long)(((long)_CPU_FREQ)/(2*4*(long)_SPI_FREQ_HZ))
```

FIGURA 16 – Calcolo del Prescaler

3.2 Lettura del Dato

Per distinguere il processo di lettura da quello di scrittura, è stato deciso di agire sul bit più significativo del registro. Nel caso della lettura varrà 0, viceversa nel caso della scrittura varrà 1. Dato che abbiamo a disposizione 8 bit, ci saranno $2^{n-1} = 128$ possibili registri indirizzabili, sufficienti a coprire l'intera area di memoria del sensore.

Per quanto riguarda il ciclo di lettura, come illustrato nell'introduzione, essendo una comunicazione seriale possiamo leggere gli 8 bit in uscita solo se altri 8 bit verranno inseriti in ingresso. È proprio su questo principio che si basa il codice scritto, ossia, all'interno di un while impostiamo un valore al buffer SSP1BUF, dopodiché aspettiamo che il registro TX sia vuoto, azzeriamo il flag SSP1IF e ripetendo con un secondo ciclo while ritorniamo in uscita sullo stesso buffer il valore da leggere.

La figura sottostante illustrerà ciò che ho appena detto:

```
unsigned char spiread(unsigned char addr)
{
    SSP1BUF=addr;
    while(!SSP1IF){} //wait for TX buffer to empty
    SSP1IF = 0;

    SSP1BUF = 0x00;
    while (!SSP1IF) {}
    SSP1IF = 0;

    return SSP1BUF;
}
```

FIGURA 17 – procedura di lettura SPI

3.3 Scrittura del Dato

Abbiamo detto che, per quanto riguarda la scrittura, verrà effettuata solo quando io leggo un 1 nel bit più significativo del registro. Analogamente alla lettura, per quanto riguarda la scrittura di un dato tramite il protocollo SPI, utilizzando due cicli while si fa in modo di ricevere prima l'indirizzo, e poi il dato da trasmettere.

Ancora una volta la figura sottostante mostrerà ciò di cui abbiamo parlato:

```
void spiwrite(unsigned char addr,unsigned char data)
{
    SSP1BUF=(addr | 0x80);
    while(!SSP1IF){} //wait for TX buffer to empty
    SSP1IF = 0;
    SSP1BUF=data;
    while(!SSP1IF){} //wait for TX buffer to empty
    SSP1IF = 0;
}
```

FIGURA 18 – procedura di scrittura SPI

Per finire, la procedura setCE mi permette di gestire il segnale ENABLE per il PIC, attraverso un semplice controllo di una variabile di ingresso della funzione:

```
void setCE(char boolean){
    if(boolean == 1){
        __delay_ms(WAIT_CS);
        CSEN = 1;
    }else {
        CSEN = 0;
        __delay_ms(WAIT_CS);
    }
}
```

FIGURA 19 – procedura per impostare il Chip Enable

4. Sviluppo Interfaccia UART

Se il protocollo SPI aveva bisogno di una base dei tempi per funzionare, poiché sincrono, l'interfaccia UART non ne necessita, poiché ha una comunicazione di tipo asincrona. Bisogna quindi studiare un modo che permetta al ricevitore e trasmettitore di sincronizzarsi. Definendo una BAUDRATE comune, io sono sicuro che i messaggi di lettura e scrittura siano gestiti in maniera efficace e senza errori.

4.1 Inizializzazione

Il valore di Bit Rate scelto è stato di 9600bps

```
#define _UART_BAUDRATE 9600
// with SYNC=0 , BRG16=0 , BGRH=1
// baudrate is: FOSC/[16 (n+1)] where n= SPBRGH:SPBRGL
#define SPBRGVAL (int)(((long)_CPU_FREQ-(long)_UART_BAUDRATE)/(16*(long)_UART_BAUDRATE))-1
SPBRGL = SPBRGVAL & 0xFF;
SPBRGH = SPBRGVAL << 8;
```

FIGURA 22 – definizione della BaudRate

La #define SPBRGVAL altro non è che il seguente calcolo per trovare il rapporto della coppia di registri SPBRGH e SPBRGL che determinano il periodo della velocità di trasmissione:

EXAMPLE 26-1: CALCULATING BAUD RATE ERROR

For a device with Fosc of 16 MHz, desired baud rate of 9600, Asynchronous mode, 8-bit BRG:

$$\text{Desired Baud Rate} = \frac{Fosc}{64([SPBRGH:SPBRGL] + 1)}$$

Solving for SPBRGH:SPBRGL:

$$X = \frac{\frac{Fosc}{\text{Desired Baud Rate}}}{64} - 1$$

$$= \frac{\frac{16000000}{9600}}{64} - 1$$

$$= [25.042] = 25$$

$$\text{Calculated Baud Rate} = \frac{16000000}{64(25 + 1)}$$

$$= 9615$$

$$Error = \frac{\text{Calc. Baud Rate} - \text{Desired Baud Rate}}{\text{Desired Baud Rate}}$$

$$= \frac{(9615 - 9600)}{9600} = 0.16\%$$

FIGURA 23 – calcolo del valore ai registri SPBRGH e SPBRGL

Dopodiché sono stati settati i vari pin del Microprocessore in modo corretto per la trasmissione e la ricezione dei dati, più una serie di impostazioni di base:

```
RXDTSEL = RXS; // RX alternate pin
TXCKSEL = TXS; // TX alternate pin
SYNC = 0; // asynchronous
SPEN = 1; // enable module, this will set TX as output, no RX
TXEN = 1; // enable transmitter
CREN = 1; // enable receiver
RX_PIN = 1; // RX as input
RX9 = 0; // 8 bit reception
TX9D = 0; // 8bit transmission
ADDEN = 0; // no addressing
ABDEN = 0; // autobaud disabled
WUE = 0; // normal receiver operation, no wake up
SCKP = 0; // transmit non-inverted data
BRG16 = 0; // 8 bit baudrate generator
BRGH = 1; // high speed
```

FIGURA 24 – settaggio dei pin TX e RX

Dato che il nostro Microcontrollore ha 14 pin, il protocollo UART è possibile settarlo su diverse scelte. Tramite le macro #define sottoscritte, abbiamo la possibilità di scegliere quale tra le configurazioni di pin sia la più adatta:

```
#ifndef USE_TX_ALT
#define TX_PIN TRISA0
#define TXS 0
#else
#define TX_PIN TRISC4
#define TXS 1
#endif

#ifndef USE_RX_ALT
#define RX_PIN TRISA1
#define RXS 0
#else
#define RX_PIN TRISC5
#define RXS 1
#endif
```

FIGURA 25 – scelta tra i vari pin TX e RX

4.2 Lettura del Dato

Il principio di funzionamento del ciclo di lettura del dato tramite UART è pressochè simile a quello usato per il protocollo SPI:

```
unsigned char getch()
    {
    /* retrieve one byte */
    while(!RCIF){}  /* set when register is not empty
                address = RCREG;
                while(!RCIF)    set when register is not empty
                      continue;
                      dato = RCREG;*/
    return RCREG;
    }
```

FIGURA 26 – ciclo di lettura del dato tramite UART

Nella funzione `getch()`, all'interno di un ciclo `while` controllo se il flag RCIF è settato o meno, questo discrimina l'avvenuta o mancata ricezione del dato da leggere, che verrà messo sul registro RCREG.

4.3 Scrittura del Dato

Anche per quanto riguarda il principio di funzionamento del ciclo di scrittura del dato tramite UART è analogo a quello usato per il protocollo SPI:

```
void putch(unsigned char byte)
    {
    /* output one byte */
    while(!TXIF)    /* set when register is empty */
        continue;
    TXREG = byte;
    }
```

FIGURA 27 – ciclo di scrittura del dato tramite UART

Con l'utilizzo della funzione `putch()`, sempre all'interno di un ciclo `while`, controllo se il flag TXIF sia settato o meno, questo implica se il registro è vuoto o pieno, e se così è viene scritto sul registro TXREG.

5. Risultati

Vengono ora mostrati i risultati ottenuti sperimentalmente:

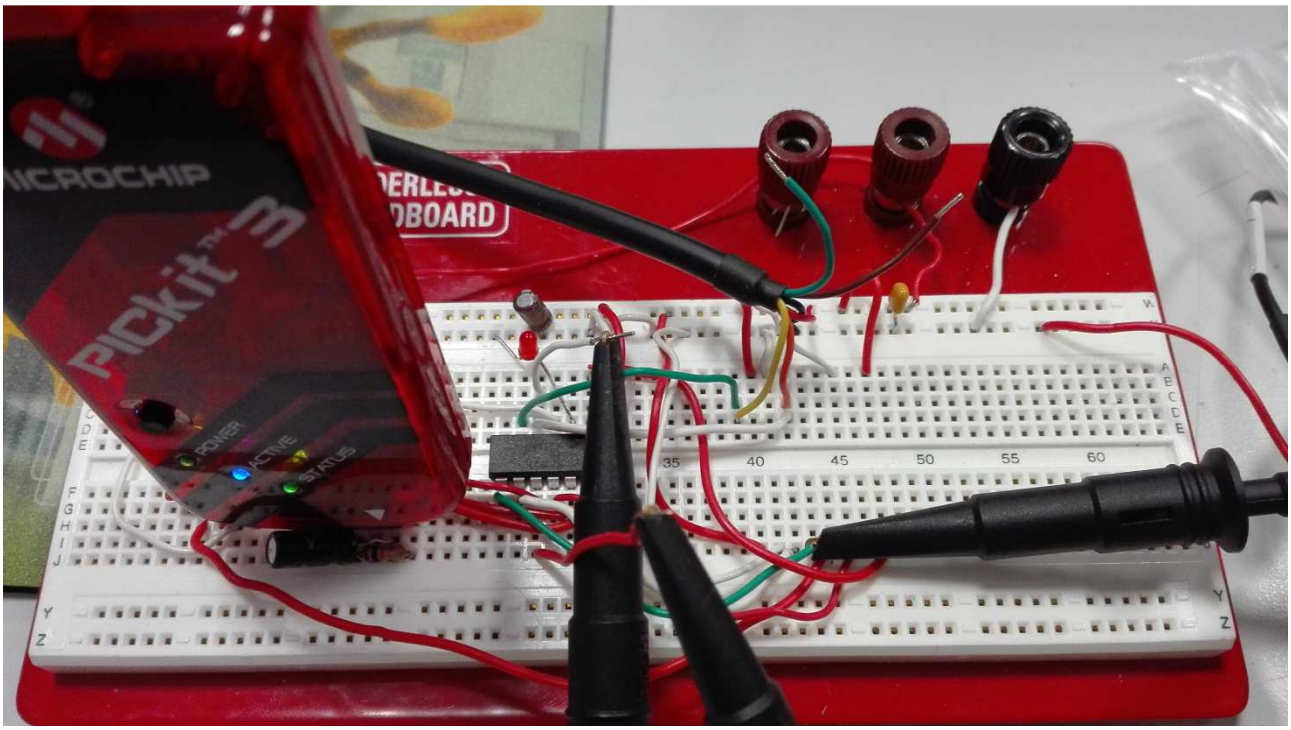


FIGURA 28 – Immagine del sistema montato su breadboard

Come si può notare nell'immagine, sulla sinistra abbiamo il programmatore Pickit3 di Microchip appositamente connesso al microcontrollore, collocato al centro, per potergli inviare il firmware da implementare. Il cavo nero, in alto, è il cavo seriale TTL-232R.3V3, che termina con 6 fili. Di questi ne sono stati utilizzati solo quattro. Il rosso è il cavo di alimentazione che fornirà la tensione di 1.8V al circuito, il nero è il cavo di massa, il giallo è il cavo di ricezione dati da parte del sensore per trasmetterli al PC, mentre l'arancione è il cavo di trasmissione con cui avverrà la comunicazione inversa, ossia tra PC e sensore.

Tramite opportune sonde per oscilloscopio è stato possibile verificare il funzionamento del sistema.

L’immagine seguente mostra l’interfaccia grafica, utilizzata per trasmettere al sensore sia l’indirizzo che il dato. Come possiamo notare è costituita da una serie di impostazioni di base quali il numero di bit di trasmissione, il numero di stop bit e la baud rate, più una serie di pulsanti di cui, quello in alto a destra mi permette di connettermi al cavo seriale del PC, mentre i tasti `write` e `read` mi permettono di trasmettere al sensore indirizzo e dato o di ricevere dal sensore una informazione.

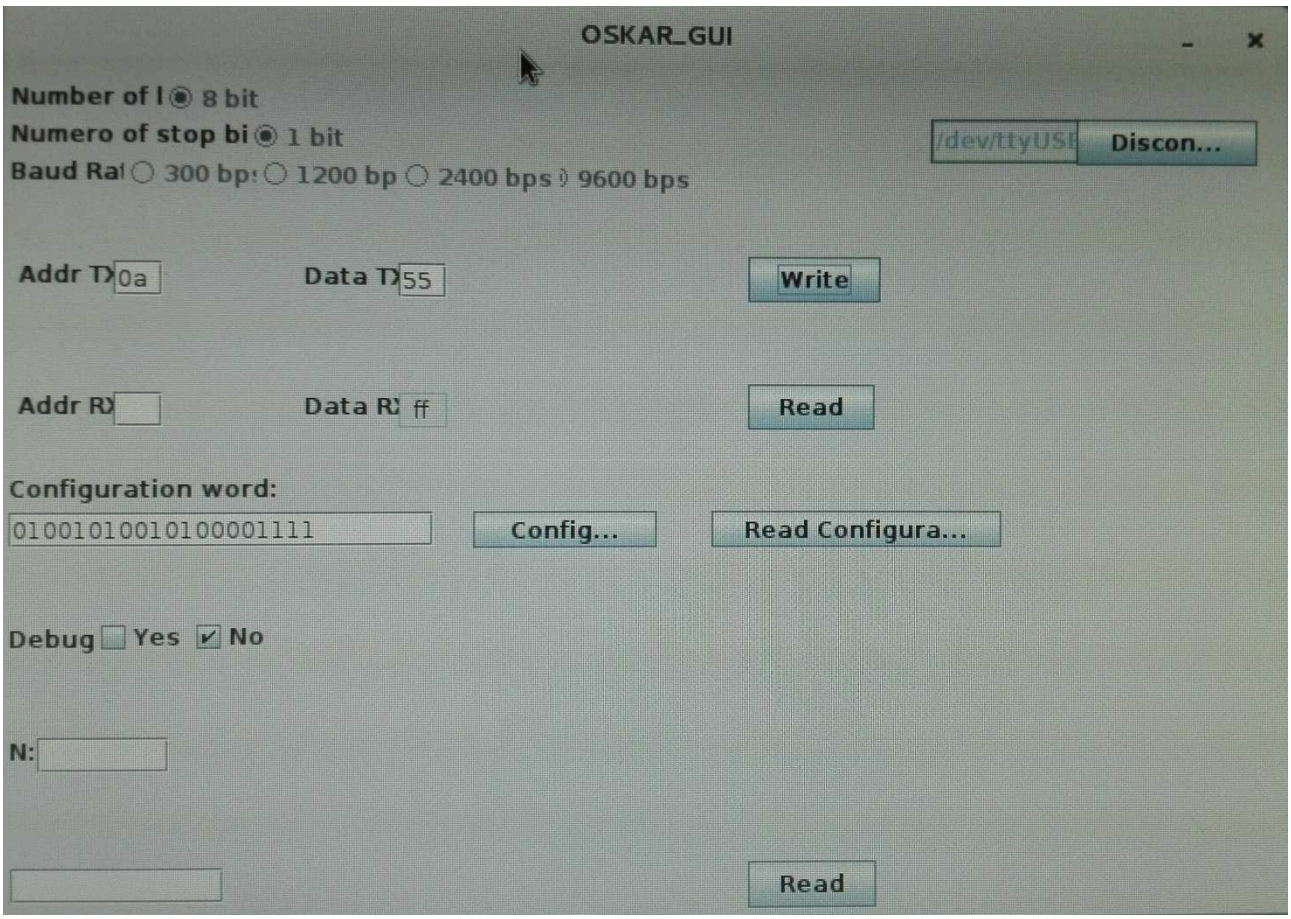


FIGURA 29 – Interfaccia grafica

Nella prossima immagine possiamo notare tre segnali mostrati a video nell'oscilloscopio. Il primo rappresenta un dato a 16bit contenente in serie l'indirizzo e il dato da voler trasmettere al sensore. Il secondo rappresenta il clock dell'interfaccia SPI, mentre il terzo segnale rappresenta il MOSI, sempre facente parte del protocollo SPI.

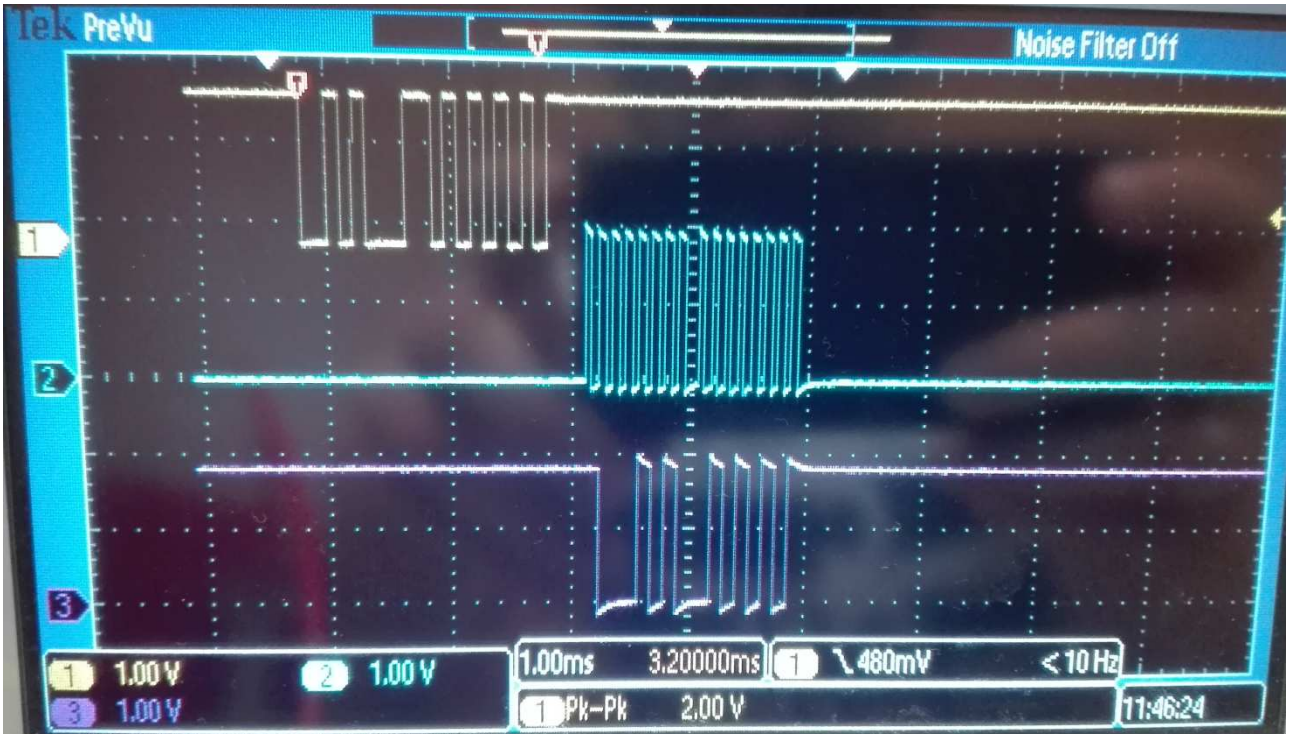


FIGURA 30 – Immagine dei segnali da trasmettere al sensore

In dettaglio possiamo notare che la trama in cui è compreso l'indirizzo e il dato da trasmettere, comprende come bit più significativo un 1, per cui in questo caso, volendo trasmettere l'indirizzo "0A" notiamo di aver trasmesso l'indirizzo "8A". In più a causa della politica FILO (First Input Last Output) utilizzata dal seguente registro, notiamo che la trama letta è invertita. Per cui una giusta implementazione del codice riconoscerà il bit più significativo come un 1, quindi di voler effettuare una scrittura e di mandare indirizzo e dato correttamente, senza l'1 come MSB.

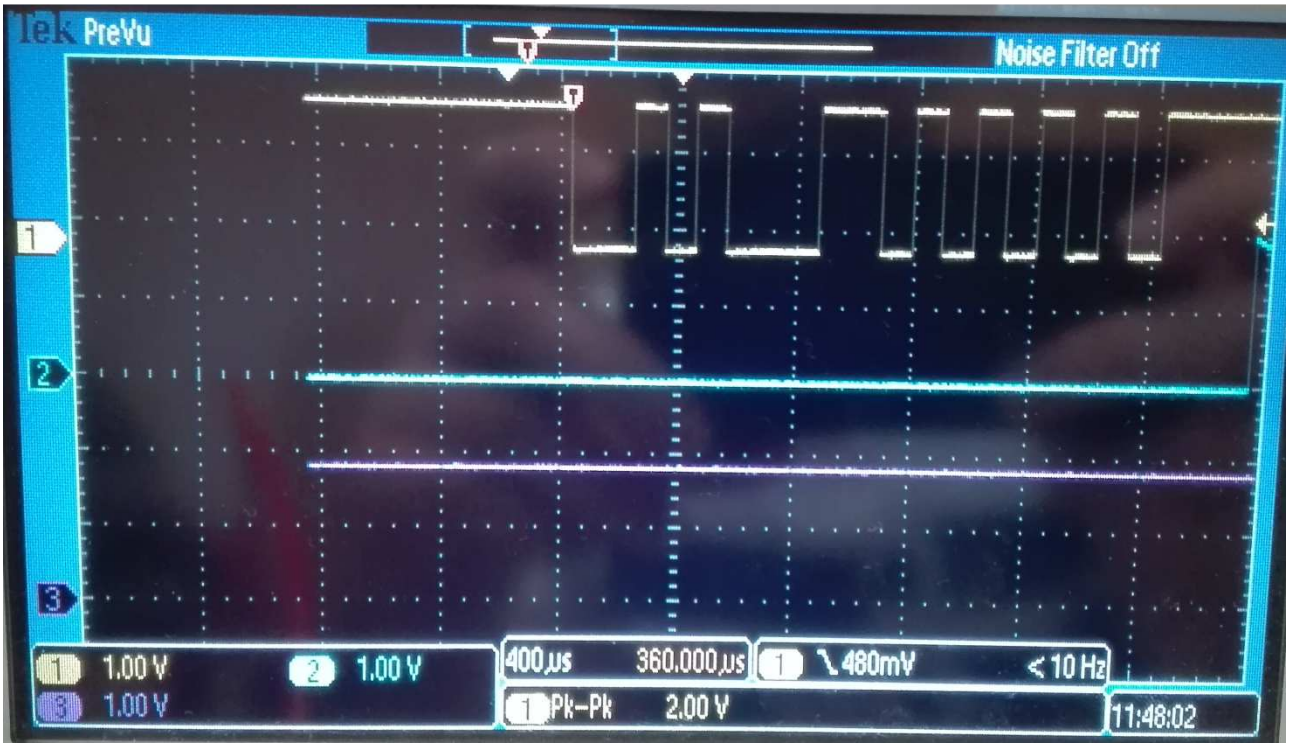


FIGURA 31 – Immagine rappresentante l'indirizzo e il dato da trasmettere al sensore

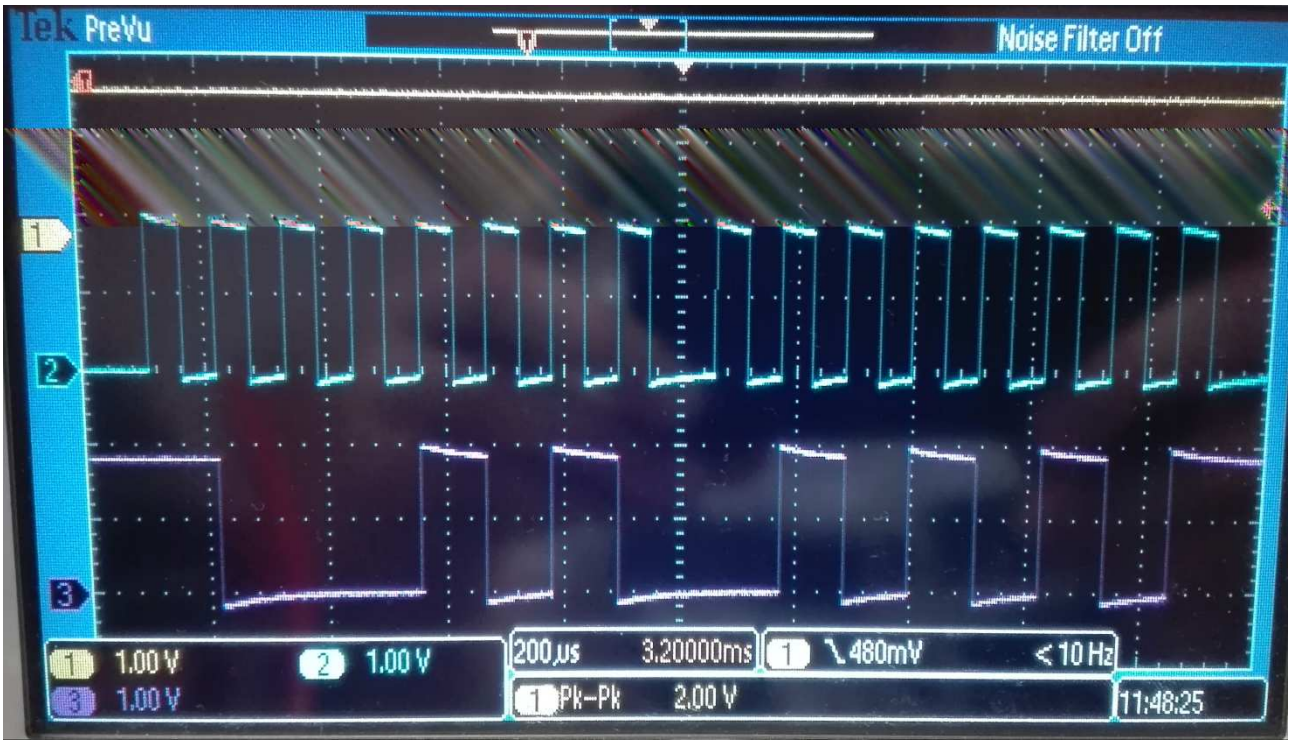


FIGURA 32 – Immagine dei segnale di CLK e MOSI

In quest'ultima immagine si notano in blu il segnale di CLOCK e in viola il MOSI dell'interfaccia SPI che corrispondono ai valori effettivi di trasmissione dati.

6. Conclusioni

L’obiettivo di questa tesi è stato la progettazione e la programmazione di un firmware in grado di pilotare un microcontrollore a scambiare dati tra un calcolatore e un sensore.

Sono state implementate due tipi di interfacce seriali, una SPI per permettere al Microcontrollore di comunicare col sensore e una UART, per permettere allo stesso di comunicare con un personal computer.

Al termine della fase di progettazione se ne è verificato il corretto funzionamento attraverso i segnali rilevati tramite oscilloscopio e si è convalidato il programma attraverso varie prove di lettura/scrittura reali.

Queste ultime hanno dato esito positivo, dimostrando il corretto funzionamento del sistema con le specifiche iniziali richieste dalla tesi.

Allegati

Codice C del MAIN

```
#include <xc.h>
#include <stdio.h>

#include "common.h"
#include "usart16f1823.h"
#include "spi16f1823.h"

// #pragma config statements should precede project file includes.
// Use project enums instead of #define for ON and OFF.

// CONFIG1
#pragma config FOSC = INTOSC
#pragma config WDTE = OFF
#pragma config PWRTE = OFF
#pragma config MCLRE = ON
#pragma config CP = OFF
#pragma config CPD = OFF
#pragma config BOREN = OFF
#pragma config CLKOUTEN = OFF
#pragma config IESO = OFF
#pragma config FCMEN = OFF

// CONFIG2
#pragma config WRT = OFF
#pragma config PLLEN =  ON
#pragma config STVREN = OFF
#pragma config BORV = LO
#pragma config LVP = OFF

//parametri configurazione


#define NREG 7
#define rb_NREG 10
static const unsigned short int address[NREG] = {0,1,2,16,13,14,15};
static const unsigned short int datawrite[NREG] = {150,0,0,0x11,0xC3,0x87,0x03};
unsigned short int dataread[NREG];

static const unsigned short int rb_address[rb_NREG] = {3,4,5,6,7,8,9,10,11,12};
unsigned short int rb_dataread[rb_NREG];
unsigned long int K;
unsigned long int Nt;
unsigned long int TREF;
```

```
void reset(void);

void main(void) {

    int count;
    int error = 0;

    OSCCON = 0b01110000;

    ANSELA = 0; // tutti io come digitali
    ANSELC = 0; // tutti io come digitali
    ADON = 0; // modulo AD spento
    WPUA = 0; // pullup disabilitate
    WPUC = 0;

    TRISA5 = 0; // RESET
    TRISA4 = 0; // LED

    LED = 0; //AZZERO BIT ERRORE
    // Attendo che il clock sia pronto (MFIOFR)
    while (!HFIOFR) {
    }
    while (!PLLR) {
    }

    // Inizializzo l'UART
    uart_init();
    // Inizializzo l'SPI
    setCE(1);
    spi_init();

while (1) {
    //uso del protocollo UART ed SPI completo
    vogliamo_scrivere=NO;

    getch_timeout(0,&uart_address);
    if((uart_address & 0x80) == 0x80){
        vogliamo_scrivere = SI;
        uart_address  &= (~0x80);
    }
```

```
if((uart_address>=0x00) && (uart_address<=0x10)){
    if(vogliamo_scrivere == SI){
        if(getch_timeout(100000,&uart_dato)){
            setCE(0);
            spiwrite(uart_address,uart_dato);
            setCE(1);
        } else {
            setCE(0);
            setCE(1);
        }
    }else{
        setCE(0);
        uart_dato = spiread(uart_address);
        setCE(1);
        putch(uart_dato);
    }

        } else if(uart_address == 0x11){
            reset();    //attivo il reset
        }  else if(uart_address == 0x12){
            RC3 = 1; //start bit
            RC3 = 0;
            //aspettiamo il data ready!!!!!!
            uart_dato = spiread(0x03);
            putch(uart_dato);
            uart_dato = spiread(0x04);
            putch(uart_dato);
            uart_dato = spiread(0x05);
            putch(uart_dato);
            uart_dato = spiread(0x06);
            putch(uart_dato);
            uart_dato = spiread(0x07);
            putch(uart_dato);
            uart_dato = spiread(0x08);
            putch(uart_dato);
        } else{
            //putch(ERRORE);
        } //tutti gli altri casi errore
    }
}

  //OSKaR necessita di un impulso di reset
  void reset(void) {
      RESET = 0;
      //__delay_ms(10);
      RESET = 1;
      //__delay_ms(10);
      RESET = 0;
      //__delay_ms(10);
  }
```

Codice C Interfaccia SPI

```
#include <xc.h>
#include "spi16f1823.h"
#include <stdint.h>

void spi_init()
    {

        //setto le porte
        TRISC2 = 0; // MOSI
        TRISC0 = 0; // SCK
        TRISC1 = 1; // MISO
        TRISA2 = 0; // EN


        //Master mode, disabilito le porte
        SSPEN = 0;

        SMP = 1;
        CKE = 0;
        CKP = 0;
        // SPI Master mode, clock = TMR2 output/2
        SSPM0 = 1;
        SSPM1 = 1;
        SSPM2 = 0;
        SSPM3 = 0;
        //abilito le porte
        SSPEN = 1;
        SSP1IF = 0;

        //TMR2 is off
        TMR2ON = 0;

        //prescaler is 1:64
        T2CKPS0 = 1;
        T2CKPS1 = 1;

        //postscaler 1:1
        T2OUTPS0 = 0;
        T2OUTPS1 = 0;
        T2OUTPS2 = 0;
        T2OUTPS3 = 0;

        PR2 = PR2VAL/64;
        //TMR2 is on
        TMR2ON = 1;

    }
```

```
void spiwrite(unsigned char addr,unsigned char data)
{
    SSP1BUF=(addr | 0x80);
    while(!SSP1IF){} //wait for TX buffer to empty
    SSP1IF = 0;
    SSP1BUF=data;
    while(!SSP1IF){} //wait for TX buffer to empty
    SSP1IF = 0;
}
```

```
unsigned char spiread(unsigned char addr)
{
    SSP1BUF=addr;
    while(!SSP1IF){} //wait for TX buffer to empty
    SSP1IF = 0;

    SSP1BUF = 0x00;
    while (!SSP1IF) {}
    SSP1IF = 0;

    return SSP1BUF;
}

//gestione firmware del chip select di oskar
void setCE(char boolean){
    if(boolean == 1){
        __delay_ms(WAIT_CS);
        CSEN = 1;
    }else {
        CSEN = 0;
        __delay_ms(WAIT_CS);
    }
}
```

Codice C Interfaccia UART

```c
#include <xc.h>
#include "usart16f1823.h"

void uart_init()
    {
    RXDTSEL = RXS; // RX alternate pin
    TXCKSEL = TXS; // TX alternate pin
    SYNC = 0; // asynchronous
    SPEN = 1; // enable module, this will set TX as output, no RX
    TXEN = 1; // enable transmitter
    CREN = 1; // enable receiver
    RX_PIN = 1; // RX as input
    RX9 = 0; // 8 bit reception
    TX9D = 0; // 8bit transmission
    ADDEN = 0; // no addressing
    ABDEN = 0; // autobaud disabled
    WUE = 0; // normal receiver operation, no wake up
    SCKP = 0; // transmit non-inverted data
    BRG16 = 0; // 8 bit baudrate generator
    BRGH = 1; // high speed

    SPBRGL = SPBRGVAL & 0xFF;
    SPBRGH = SPBRGVAL << 8;
```

```
    while(RCIF)
        {
        char Temp;
        Temp = RCREG;
        }
    TXIF = 0; // reset transmit interrupt flag
    __delay_ms(10);
    }

void putch(unsigned char byte)
    {
    /* output one byte */
    while(!TXIF)    /* set when register is empty */
        continue;
    TXREG = byte;
    }

unsigned char getch()
    {
    /* retrieve one byte */
    while(!RCIF){}  /* set when register is not empty
                address = RCREG;
                while(!RCIF)    set when register is not empty
                      continue;
                      dato = RCREG;*/
    return RCREG;
    }

unsigned char getche(void)
    {
    unsigned char c;
    putch(c = getch());
    return c;
    }
```

Bibliografia

[1] **Documenti riguardante il Microcontrollore**

http://www.settorezero.com/wordpress/corso-programmazione-pic-in-c-lezione-1-cose-un-microcontrollore-caratteristiche-note-introduttive-come-scegliere-programmatore-e-linguaggio-di-programmazione/

[2] **Datasheet del Microcontrollore PIC16F1822**

http://ww1.microchip.com/downloads/en/DeviceDoc/40001413E.pdf

[3] **Datasheet del PICKit3**

http://ww1.microchip.com/downloads/en/DeviceDoc/51795B.pdf

[3] **Pagina Wikipedia dedicata al protocollo SPI**

https://it.wikipedia.org/wiki/Serial_Peripheral_Interface

[4] **Pagina Wikipedia dedicata al protocollo UART**

https://it.wikipedia.org/wiki/UART

[5] **Informazioni e immagine alimentatore**

https://www.upc.edu/sct/ca/documents_equipament/d_336_e3630a.pdf

[6] **Informazioni e immagine oscilloscopio**

http://it.rs-online.com/web/p/oscilloscopi-digitali/0327728/