Alma Mater Studiorum · Università di Bologna
Sede Di Cesena

SECONDA FACOLTÀ DI INGEGNERIA CON SEDE A CESENA
CORSO DI LAUREA IN INGEGNERIA ELETTRONICA
INFORMATICA E DELLE TELECOMUNICAZIONI

TITOLO DELL' ELABORATO

Amplificatore per elettrofisiologia integrato in una USB key: realizzazione dell'interfaccia a microcontrollore

Elaborato in

Elettronica Dei Sistemi Digitali

Relatore
Prof. Ing. Aldo Romani

Presentata da
Davide Fabbri

Correlatore
Prof. Ing. Federico Thei

Sessione III
A.A. 2011/2012

Indice

Introduzione

Nel lontano 1965, Gordon Moore definì mediante una legge empirica, che il numero dei transistor, ogni anno, raddoppia all'interno di un singolo chip; possiamo oggi non far altro che prendere atto di tale affermazione, osservando i decenni trascorsi da quella data. Di fatto l'elettronica si è spinta a massimizzare le prestazioni dei dispositivi, riducendo fortemente la scala di integrazione degli stessi. Ciò ha consentito di realizzare integrati con alta densità di componenti al loro interno, pur mantenendo contenute le dimensioni. Ad oggi con l'evoluzione dei processori abbiamo in mano un potenziale enorme per risolvere svariate applicazioni, in particolare, relative al mondo dell'Embedded System. Ciò rappresenta la traccia che ci ha permesso di comprendere la possibilità sostituire la ormai datata apparecchiatura in possesso. Si tratta di un amplificatore per elettrofisiologia (Fig. 1), il quale consente di misurare le correnti ioniche nei canali ionici delle cellule. I canali ionici sono proteine trans-membrana che permettono il passaggio di determinati ioni dall'esterno verso l'interno e viceversa. Essi possiedono un ruolo essenziale in molti processi biologici e i loro malfunzionamenti rappresentano la causa di numerose malattie o patologie gravi. La capacità del canale ionico di rispondere a diversi stimoli chimico-fisici ha ispirato la progettazione di sensori ibridi, che consentissero di di caratterizzare il comportamento elettrico del canale con elevata accuratezza e precisione. Il sistema che vogliamo progettare fa riferimento ad una recente board comprendente di 8 canali di acquisizione dati. Ciascun canale fa capo ad un chip definito dal nome Berillio, il quale è stato realizzato per consentire di rilevare variazioni di pico-Amper di corrente elettrica nei canali ionici e di convertirle in un segnale digitale di tipo delta-sigma. Tale uscita accompagnata da un segnale di sincronismo viene gestita da un FPGA, il cui compito principale consiste nell'eseguire un filtraggio digitale sinc3 del segnale stesso e procedere con l'invio dell'informazione ad un calcolatore, mediante modulo USB esterno. I dati acquisiti dal pc, sono visualizzabili su un apposita interfaccia software. Essa consente anche di programmare il chip Berillio, con trasferimento dati su comunicazione seriale di tipo SPI gestita da FPGA. L'oggetto della tesi propone un cambiamento di approccio progettuale nella realizzazione del sistema appena descritto. Sostituiamo il dispositivo FPGA con un microcontrollore, per ottenere un sistema miniaturizzato ad un singolo canale (Fig. 2). Mentre il primo è un' unità digitale programmabile, che contiene solo circuiti logigi e connessioni, il secondo è costituito da un sistema più complesso, che include numerose funzionalità che gli consentono di operare in assoluta autonomia. Esso possiede, in particolare, una memoria RAM, una memoria Programma, moduli SPI e USB integrati. D'altra parte però, le prestazioni di calcolo, che consentono la gestione del programma software caricato al suo interno, non permettono all'utente di gestire numerosi canali paralleli, come era stato

implementato precedentemente in FPGA, pertanto, prendiamo in considerazione un unico canale di acquisizione dati. Il software, contenuto nel micro, è prodotto all'interno dell'ambiente di sviluppo*Mplab* e scritto in linguaggio C. In definitiva, l'obiettivo della tesi è quello di sfruttare il progresso scentifico e tecnologico, che risiede in un dispositivo stand-alone, ai fini di realizzare un progetto con ottime garanzie prestazionali e ridotte dimensioni, che gli conferiscono importanti proprietà di portatilità e, vedremo, ci consentiranno di introdurre il sistema all'interno di una semplice USB-key pronta all'uso.

Figura 1: Amplificatore per elettrofisiologia *Axon*

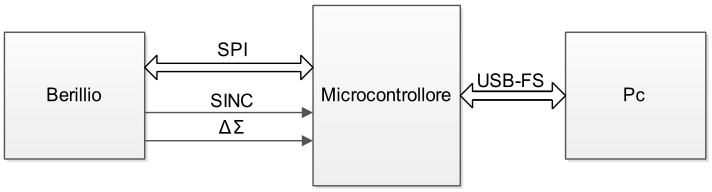


Figura 2: Schema a blocchi progetto

Capitolo 1

Il Microcontrollore

Un microcontrollore o microcalcolatore **MCU**(*Micro-Controller Unit*) è un circuito elettronico integrato di tipo programmabile, che include al suo interno un'unita a microprocessore, periferiche di memoria(RAM-ROM) e numerose periferiche di tipo analogico e digitale. Questa importante peculiarità, rende il micro appetible per svariate applicazioni, a discapito di un FPGA o un microprocessore. Infatti, anche i microprocessori, comunemente utilizzati in apparecchiature nelle quali si predilige la risorsa di calcolo, vengono messi in disparte, la dove è richiesta la realizzazione di dispositivi portatili con basso consumo energetico. Cercheremo di definire in questo capitolo quale microcontrollore faccia al caso nostro, in base alle specifiche richieste.

1.1 Scelta del Micro

Considerando la leadership di casa Microchip nella produzione di microcontrollori e della vasta gamma offerta agli utenti, la scelta ricade su uno dei suoi dispositivi. Vengono messi a disposizione micro per applicazioni generiche e per applicazioni mirate all'elaborazione digitale di segnali analogici (DSC, *Digital Signal Controller*), con architetture ad 8,16 e 32 bit. Dalle specifiche si richiede di realizzare un filtraggio digitale sinc3, che prende in ingresso un segnale delta-sigma, accompagnato da un sincronismo ad onda quadra, alla frequenza $f_{sinc} = 1.25MHz$. Se pensassimo in un primo momento di voler gestire i dati in *real time*, occorre che la cpu sia in grado di realizzare il filtraggio e possibilmente di inviare i pacchetti al pc mediante USB, nell'intervallo di tempo a disposizione (Eq. 1.1) prima che arrivi il nuovo dato.

$$T_s = \frac{1}{f_{sinc}} = 800ns \tag{1.1}$$

Pic 8 bit

Si è partiti con l'analizzare i pic a 8 bit, considerando la sola serie 18(l'unica che presenta il modulo USB)(Fig. 1.1). Essi possono operare alla frequenza massima di 20MHz, con una alimentazione di +5v.

$$T_{clock} = \frac{1}{f_{clock}} = 50ns \tag{1.2}$$

L'architettura del dispositivo, non consente di processare un'istruzione assembly in un solo periodo di clock, ma ne servono ben 4, nel caso peggiorativo di considerare il processo su un dato a 32 bit. Per cui ogni singola istruzione è completata in un tempo pari a $T_{ist} = 4 * Tc = 200ns$. Capiamo bene che, in riferimento alla (Eq. 1.1), abbiamo a disposizione solo 4 istruzioni assembly per eseguire il filtraggio e le operazioni di calcolo necessarie. Non sono sufficienti, considerando anche il fatto che il compilatore impiega in media 3/4 istruzioni assembly per produrre una istruzione C. Detto questo decliniamo la serie a 8 bit, per valutare nel passo successivo i 16 bit.

	Baseline Architecture	Mid-Range Architecture	Enhanced Mid-Range Architecture	PIC18 Architecture
Pin Count	6-40	8-64	8-64	18-100
Interrupts	No	Single interrupt capability	Single interrupt capability with hardware context save	Multiple interrupt capability with hardware context save
Performance	5 MIPS	5 MIPS	8 MIPS	Up to 16 MIPS
Instructions	33, 12-bit	35, 14-bit	49, 14-bit	83, 16-bit
Program Memory	Up to 3 KB	Up to 14 KB	Up to 28 KB	Up to 128 KB
Data Memory	Up to 138 Bytes	Up to 368 Bytes	Up to 1,5 KB	Up to 4 KB
Hardware Stack	2 level	8 level	16 level	32 level
Features	▪ Comparator ▪ 8-bit ADC ▪ Data Memory ▪ Internal Oscillator	In addition to Baseline: ▪ SPI/I²C™ ▪ UART ▪ PWMs ▪ LCD ▪ 10-bit ADC ▪ Op Amp	In addition to Mid-Range: ▪ Multiple Communication Peripherals ▪ Linear Programming Space ▪ PWMs with Independent Time Base	In addition to Enhanced Mid-Range: ▪ 8x8 Hardware Multiplier ▪ CAN ▪ CTMU ▪ USB ▪ Ethernet ▪ 12-bit ADC
Highlights	Lowest cost in the smallest form factor	Optimal cost to performance ratio	Cost effective with more performance and memory	High performance, optimized for C programming, advanced peripherals
Total Number of Devices	16	58	29	193
Families	PIC10, PIC12, PIC16	PIC12, PIC16	PIC12FXXX, PIC16F1XX	PIC18

Figura 1.1: Compare 8-bit PIC MCU Architectures

Pic 16 bit

La serie a 16 bit si basa sempre su un'architettura di tipo ARM come nei pic di fascia inferiore, con la particolarità di offrire dispositivi ad uso generico come PIC24 e di tipo **DSP**(*Digital Signal Processor*) designati dalla sigla dsPIC. Questi ultimi hanno una importante novità rispetto ai cugini della stessa famiglia, ovvero quella di possedere un core aggiuntivo di tipo DSP. Il vantaggio si concretizza in migliori performance, relative ad applicazioni audio e/o elaborazioni digitali. La frequenza massima di sistema, garantita da Microchip per i 16 bit, è pari a $f_{clock} = 40MHz$. Ponendoci nel caso più critico, di processare un dato a 32 bit, occorrono due fronti di clock per eseguire un istruzione. Deduciamo che $T_{ist} = 2 * Tc = 50ns$; pertanto in 800ns, possono essere eseguite un numero massimo di 16 istruzioni assembly. Questi dati sono puramente teorici e ideali, ovvero non tengono conto delle non idealità dei segnali applicati, ma forniscono un'importate supporto per scegliere il giusto dispositivo. Anche in questo caso

il numero di istruzioni a disposizione è abbastanza residuo, quindi, per non precluderci già da ora la possibilità di processare ciascun dato in tempo reale, facciamo riferimento anche ai pic a 32 bit.

Pic 32 bit

I microcontrollori a 32 bit hanno un'architettura differente rispetto a quella precedente e si basano su un singolo core M4K^{TM} di MIPS® Technologies che può lavorare alla frequenza massima di $f = 80MHz$. Esso è un processore di tipo RISC(reduced instruction set computer), per cui mantiene forti analogie con i suoi predecessori in termini di set di istruzioni, il quale si presenta sempre molto contenuto ed efficiente. L'architettura di tipo RISC consente di eseguire un'istruzione elementare in un singolo ciclo. Se volessimo completare la comparativa con i dispositivi precedenti, definiamo il tempo di clock $T_{clock} = 12.5ns$. Pertanto, in tal caso si possono eseguire circa 64 istruzioni assembly. Questa è un pò la nota dolente, di fatto anche con la miglior tecnologia messa a disposizione da Microchip, abbiamo sempre un numero limitato di istruzioni a disposizione. Ciò comporta la necessità di gestire il programma nella maniera più efficiente possibile, ovvero far largo uso di interrupt annidati(interruzioni programmate del software) oltre ad un corretta impostazione della fase di filtraggio e dell'invio dati mediante USB. Deciso di intraprendere la strada dei micro a 32 bit, ai fini di mantenere un certo marigine di risorse rispetto ai suoi predecessori, si è lavorato inizialmente sulla scheda di sviluppo *Pic32 USB Starter Kit II*(Fig. 1.2), utilizzabile per un qualsiasi tipo di applicazione, in particolare, per sviluppare l'interfaccia USB richiesta da specifica.

Figura 1.2: Pic32 USB Sterter Kit II

Caratteristiche Tecniche:

- microcontrollore a 32bit PIC32MX795F512L
- microcontrollore a 32bit PIC32MX440F512H dedicato al debug
- Tre led di differente colore
- Oscillatore al quarzo di 8 MHz
- Tre pulsanti

- Connettore miniUSB per programmazione/debug
- Connettore microUSB per applicazioni OTG/device
- Connettore USB di tipo A per applicazioni Host
- Connettore per espansioni esterne

Dalle specifiche tecniche, si nota che la scheda ci consente di programmare il dispositivo integrato senza circuiteria esterna, grazie alla sezione di programmazione/debug inclusa. Di essa sfrutteremo in particolare il connettore micro USB dedicato alle applicazioni di tipo *device*, per realizzare la trasmissione dati al PC. Analizzando il PIC32MX795F512L, vediamo di definire in dettaglio alcune peculiarità necessarie per un ottima gestione software e diverse periferiche hardware largamente utilizzate nel progetto.

1.2 Modulo USB

L'**USB**(*Universal Serial Bus*) è uno standard di comunicazione seriale utilizzato principalmente per la connessione di periferiche al pc. Il suo successo è legato al fatto di avere una sola interfaccia standardizzata, con un ristretto numero di connettori in commercio. Il modulo in questione, contiene componenti digitali e analogici che consentono di implementare lo standard USB 2.0 in modalità full-speed(12 Mbps) e low-speed(1.5 Mbps) per applicazioni Host, solo full-speed per device e OTG(*On the go*). Quest'ultima consente di definire la comunicazione di due dispositivi, senza l'ausilio del calcolatore. Infatti viene realizzata una connessione di tipo client-server, che il protocollo USB-OTG crea automaticamente tra due dispositivi USB, i quali possono assumere l'etichetta sia di host sia di device. Noi non useremo questo tipo di approccio, ma ci preoccuperemo di definire il nostro sistema come device, in modo tale che possa essere letto dal pc come se fosse una comune periferica estera(*mouse, tastiera...*). Di fatto è l'host(Pc) che assume il controllo della comunicazione, interrogando coclicamente le periferiche con l'invio di un pacchetto dati(*token*) contenente i parametri:

- *tipo di transizione*
- *direzione della transizione*
- *indirizzo della periferica USB interessata*
- *endpoint number*

Ciascun dispositivo di tipo *device* presente, riceve il token, occupandosi della fase di decodifica, per riconoscere se risulta lui stesso il destinatario del pacchetto. Nel caso di esito positivo, procede con la conversione dello stream di dati.

In riferimento alla (Fig. 1.3), il contenuto della periferica hardware viene descritto dalle seguenti parti:

- **Sorgente di clock**

  Il clock utilizzato dalla periferica USB, del valore standard di 48MHz in modalità full-speed, può essere generato a partire dall'oscillatore esterno,

in modo diretto, nel caso in cui esso rappresenti già il valore di 48 MHz voluto, ponendo a 1 il bit **UPLLEN**. Invece, utilizzando la demoboard sopra menzionata, abbiamo a disposizione un oscillatore esterno di 8 MHz, per cui attraverso il bit **UPLLIDIV**, si seleziona una divisione per due, in modo tale di avere $UF_{IN} = 4MHz$, richiesto dal modulo. A questo punto il moltiplicatore interno **PLL** esegue una moltiplicazione $x24$ per avere 96 MHz, i quali nel passo successivo vengono dimezzati. Il clock della periferica USB può essere generato anche a partire dall'oscillatore interno FRC, il quale a causa della sua limitata stabilità in frequenza, non viene spesso utilizzato.

- **Serial Interface Engine (SIE)**

  Si tratta di un modulo fisico, che si preoccupa di definire la serializzazione o de-serializzazione del pacchietto inviato (conversione stream di dati da seriale a parallelo e viceversa). Per cui realizza una codifica NRZI per i dati in uscita, mentre decodifica i dati in ingresso. Inoltre esegue anche un controllo sulla ridondanza, con una verifica del **CRC**(*Cyclic Redundancy Check*) sullo stream in ingresso e genera il CRC per lo stream di uscita. Ai fini di individuare la tipologia del pacchetto, viene verificato il **PID**(*Packet's ID*), al quale sono associate anche alcune segnalazioni relative al bus, come ad esempio lo stato di *reset* della linea. La SIE può ricevere le informazioni da trasmettere, mediante l'accesso ad un interfaccia di controllo, oppure per mezzo del bus principale, accedere direttamente alla memoria RAM. Quest'ultima applicazione risulta possibile nel caso in cui venga programmato il modulo DMA(*Direct Memory Acces*)interno.

- **Transceiver**

  L'ultimo blocco del *phisical layer* è a stretto contatto con il mondo esterno, attraverso i conduttori **D+** e **D-**. I dati vengo trasmessi e ricevuti secondo codifica **NRZI**(*Not Return to Zero Inverted*), per cui lo 0 porta il segnale D+ a livello basso ed il segnale D- a livello alto definendo una permanenza di stato logico, viceversa per 1.

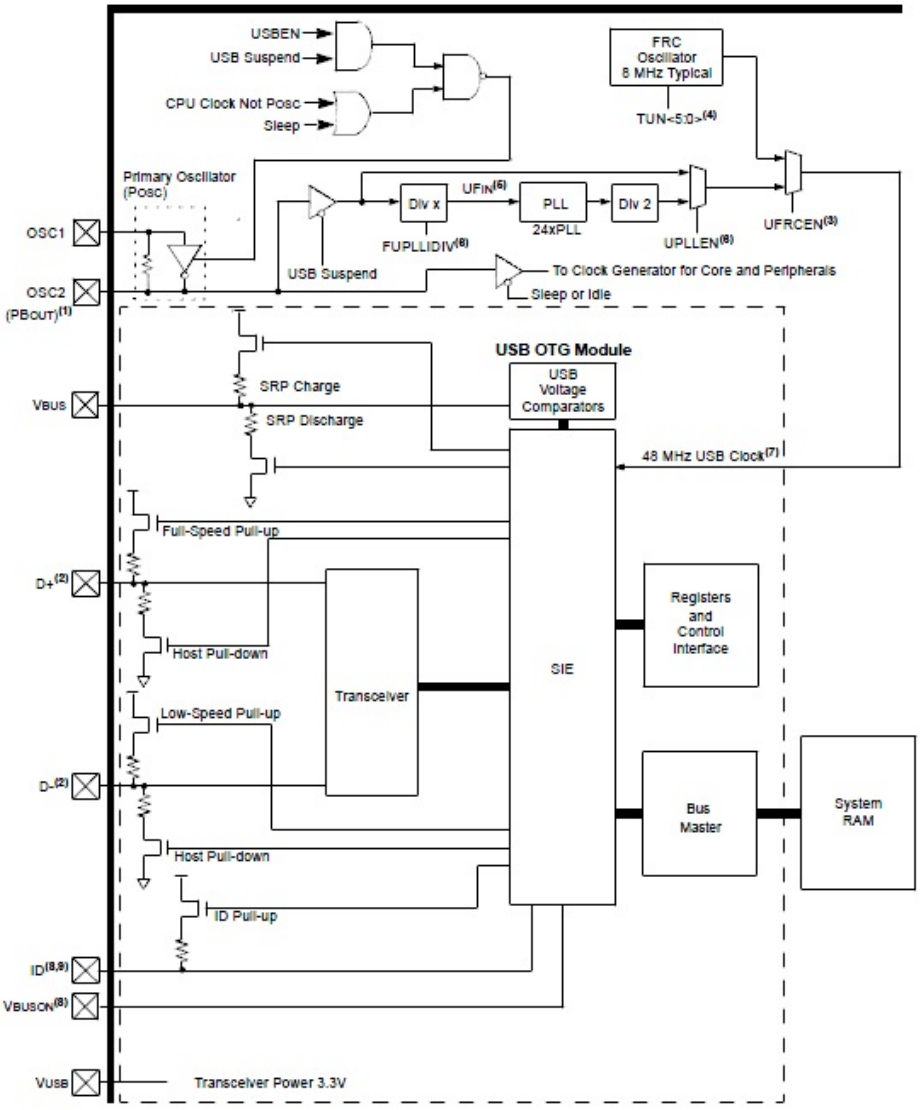


Figura 1.3: Diagramma interfaccia USB

1.3 SPI

L'interfaccia (**SPI***Serial Perifierical Interface*) implementa una comunicazione seriale, sincrona *full-duplex* tra un **Master** ed uno **Slave**. Essa è stata introdotta dalla *Motorola* per applicazioni dedicate ai suoi microprocessori. Successivamente viene adottata dalla stragrande maggiornanza dei produttori, anche se inizialmente non era standardizzata. Per questo, con il tempo, ha acquisito molteplici caratteristiche e opzioni che vanno opportunamente selezionate per garantire la corretta trasmissione. L'interfaccia è spesso preferita rispetto al sistema di comunicazione seriale bifilare di tipo I^2C, a causa delle maggiori frequenze di trasmissione ottenibili.

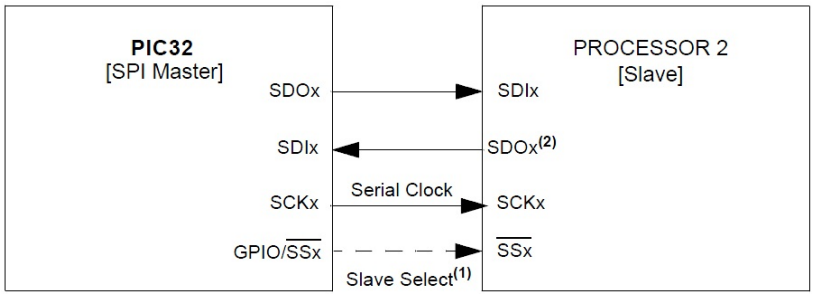


Figura 1.4: SPI Master-to-Slave Device Connection Diagram

La (Fig. 1.4), indica una trasmissione SPI tra un master (il pic) ed uno slave generico. Si possono definire 4 differenti segnali, che consentono di garantire una corretta gestione dell'interfaccia:

- **SDI** (*Serial data In*)
- **SDO** (*Serial data Out*)
- **SCK** (*Serial Clock*)
- **SS** (*Slave Select*)

Principio di funzionamento

Supponendo di voler mantenere la modalità master-slave, sarà il microcontrollore, in qualità di *primario*, a destribuire il clock di sincronizzazione alle periferiche, a indicare quale di esse deve ricevere il dato trasmesso mediante lo Slave select e a definire l'inizio di una trasmissione. L'Spi si presenta come un registro a scorrimento, la cui dimensione può essere programmata per contenere 8,16 o 32 bit (Fig. 1.5).

Prima di avviare la comunicazione, il Master attiva la linea SS relativa allo Slave con la quale vuole instaurare la tramissione dati e successivamente viene fornito il clock che gestisce la comunicazione. Una volta che lo Slave risulta attivo, i dati passano bit a bit dal master (SDO) allo slave (SDI) a partire dal quello più significativo. Ciascun bit, gradualmente entra nel registro dello slave, che inizia a sua volta lo svuotamento a partire sempre dal bit più significativo, in direzione del master(SDI). Una volta che sono stati trasmessi tutti i bit, la

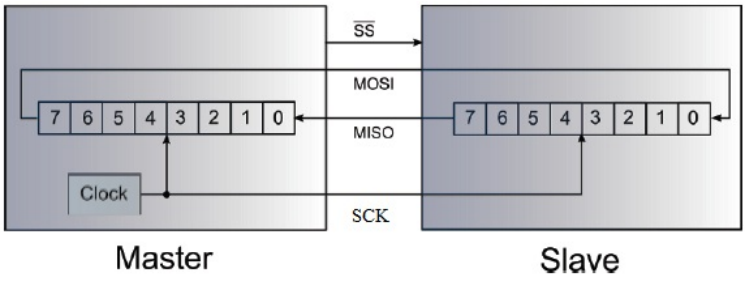


Figura 1.5: Registro a scorrimento Master-Slave

comunicazione termina in quanto viene rimosso il clock di sincronismo e lo slave select torna a livello logico basso. In genere se si possiede un solo slave, la linea SS può essere ingorata, inoltre se volessimo leggere un dato utile dallo slave, siamo costretti in questo modo a inviare un dato fittizio, per consentire il meccanismo dello scorrimento.

1.3.1 Interfaccia SPI Pic32

Il microcontrollore utilizzato presenta ben 4 moduli SPI, ognuno dei quali è analizzabile secondo lo schema di (Fig. 1.6).

I registri in evidenza sono i seguenti:

- **SPIxBRG**

  Il registro contiene 9 bit, i quali consentono di definire la *baud rate* del modulo, impiegata per generare il segnale di clock interno, di cui si serve l'SPI. Attraverso il bit di selezione **MCLKSEL** si può definire se, prelevare la sorgente di clock direttamente dalle periferiche **PBCLK**, oppure mediante sorgente esterna **MCLK**. A causa del fatto che quest'ultima opzione non è implementata nei microcontrollori di fascia alta, tra i quali fa parte anche quello da noi impiegato, occorre utilizzare il clock delle periferiche. Sussiste un'importante relazione tra la $PBCLK$ e l'SCK generato, rappresentata dalla seguente espressione:

  $$F_{SCK} = \frac{F_{PB}}{2 * (SPIxBRG + 1)} \tag{1.3}$$

  Per un motivo che tratteremo in seguito, definiamo il valore del clock esterno paria a $F_{SCK} = 10MHz$. Facendo lavorare il pic32 alla massima frequenza garantita di $F_{PB} = 80MHz$, è facile ricavare che:

  $$SPIxBUF = \frac{F_{PB}}{2 * F_{SCK}} - 1 = 3 \tag{1.4}$$

  Questo è il valore con la quale andremo a programmare tale registro, il quale consente di definire valori di baud-rate compresi tra $\frac{F_{PB}}{2}$ e $\frac{F_{PB}}{1024}$.

- **SPIxBUF**

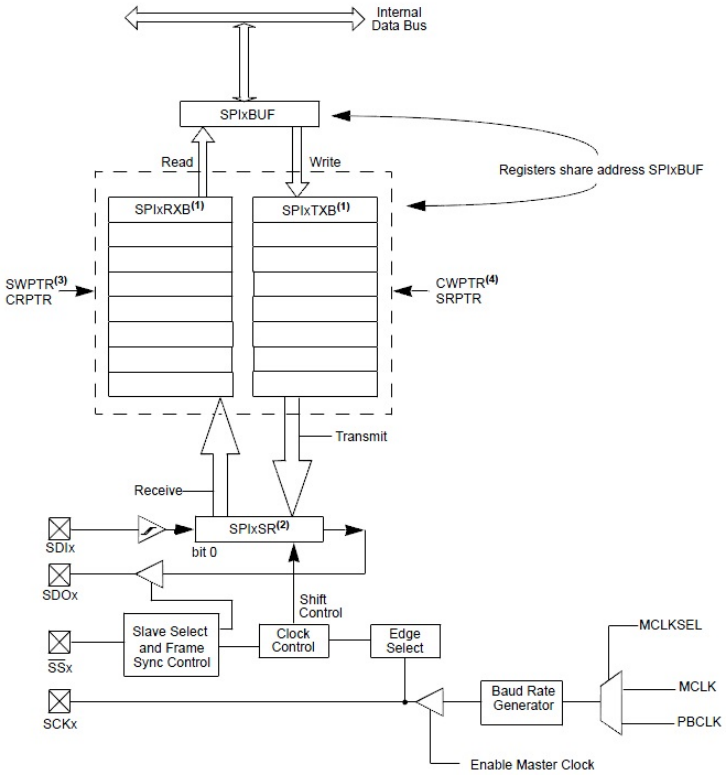


Figura 1.6: Schema a blocchi Interfaccia SPI

Ogni qual volta si vuole trasferire un dato, occorre inserire il suo valore all'interno del registro in questione. Quando è attiva la modalità di trasferimento dati a 32 bit, selezionabile mediante due bit $MODE< 32, 16 >$ presenti nel primo registro di controllo dell'SPI, il registro SPIxBUF prende in considerazione tutta la sua estensione. Nel caso in cui si voglia implementare una comunicazione dati, con pacchetti più piccoli di 8 e 16 bit, in tal caso, vengono considerati solo i bit corrispondenti meno significativi del registro stesso. E' possibile programmare il buffer, secondo la *Standard mode* o *Enhanced Buffer mode*. Nel primo caso si utilizza un'unica memoria di transito per le operazioni di trasmissione/ricezione, mentre nel secondo, si possiedono due stack dedicati singolarmente alla ricezione e alla trasmissione del pacchetto desiderato secondo la modalità **FIFO**(*First In First Out*).

- **SPIxSR**

  Nello *shift register* transitano tutti i dati da trammettere e ricevere, secondo la modalità con la quale è stato impostato il buffer.

- **SPIxSTAT**

  Rappresenta lo *Status register*, il quale contiene numerosi *flag*, per indicare alcuni avvenimenti di rilevata importanza, durante la comunicazione seriale.

Tra questi, ad esempio, può verificarsi il *Receive Overflow Status* segnalato dal bit **SPIROV**.

- **SPIxCON**

  Esso contiene i bit di controllo dell'SPI, i quali consentono di definire tutte le varie modalità di funzionamento.

- **SPIxCON2**

  Il secondo registro di controllo incorpora principalmente flag di segnalazione di errore sugli eventi di interrupt e sulla gestione del buffer. La Microchip non ha implentato tali verifiche di controllo nel microcontrollore da noi utilizzato, per cui non verrà più trattato in seguito.

1.3.2 Modalità operative

In base ai ruoli che possiedono le periferiche in uso, si possono definire alcune modalità operative, ai fini di realizzare un tipo di comunicazione, adatta alle proprie esigenze progettuali:

- **Master e Slave mode**
- **Framed mode**
- **Audio Protocolo Interface mode**

Nel (Cap. 4) analizzeremo in particolare solo le prime due modalità sopra indicate, perchè saranno oggetto della tesi qui sviluppata.

1.4 Interrupt

La gestione delle interruzioni ha un ruolo determinante nell'ottimizzazione del software. Esse consentono, a fronte di un evento, di interrompere la normale esecuzione del programma per soddisfare una routine di interrupt, nella quale risiedono le istruzioni da eseguire, secondo determinate priorità (Fig. 1.7). Questo tipo di approccio si contrappone al *polling*, usato per interrogare ciclicamente il verificarsi di un determinato evento.

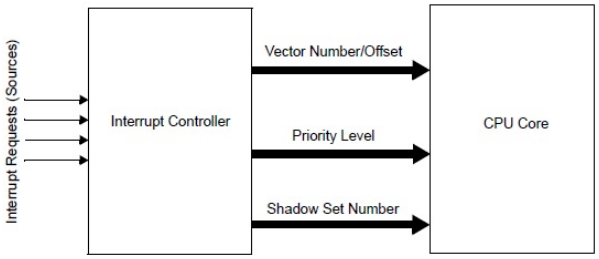


Figura 1.7: Schema a blocchi della Gestione di un Interruzione

Nei pic32 vi sono ben 256 **IRQ**(*Interrupt Request*), le quali vengono gestite dall'*interrupt controller*. Quest'ultimo ha il compito di classificare ogni richiesta

con un vettore, contenuto nella tabella **IVT**(*Interrupt Vector Table*), con il quale la CPU è in grado di identificare la routine corretta da servire(**ISR**). Inoltre alla richiesta viene associato anche un livello di priorità (da 1 a 7), gestibile all'interno del registro IPC_x(*Interrupt Priority Control*). Nella situazione in cui due eventi si verifichino contemporaneamente, viene servito prima quello a priorità più alta. Supponendo che, invece, si scateni un evento ad alta priorità mentre è in esecuzione un altro a priorità più bassa, quello che accade prende il nome di **Nesting**, ovvero si crea un annidamento degli interrupt, interrompendo la ISR che si stava eseguendo, per gestire quella con maggiore importanza. Nel nostro caso, saremo portati a rendere più prioritario il filtraggio digitale sinc3, in quanto l'esigenza è quella di evitare perdita di campioni in ingresso, che potrebbe falsare la forma d'onda in uscita. Tutte le interrupt request sono campionate sul fronte di salita del clock di sistema. Una volta che è stato riconosciuto un determinato evento, la CPU pone a 1 il flag di segnalazione corrispondente, contenuto nel registro **IFS**(*Interrupt Flag Status*). Tale bit risulta bloccante, per cui se il flag non viene azzerato prima di uscire dalla routine di interrupt, non viene servito più nessun tipo di evento associato ad esso. Mediante le modalità *single vector mode* e *multiple vector mode*,modificabile nel registro **INTCON**, è possibile rispettivamente realizzare una routine di interrupt unica, nella quale fanno capo tutti gli eventi gestiti, oppure di creare una routine per ciascun evento da soddisfare, con associato ad esso la corrispondente sorgente. L'implementazione di quest'ultimo modello, rende sicuramente più fluida l'esecuzione del programma, in quanto non occorre discriminare via software l'evento scatenante.

In base alle nostre esigenze, tra le numerose applicazioni per cui si possono interpellare gli interrupt, noi ci limiteremo a controllare le seguenti sorgenti:

- **Overflow del Timer**
- **Evento esterno**

1.5 Timer

Per dare una definizione chiara e contenuta, si potrebbe dire che il Timer rappresenta un registro contatore a incremento. In realtà, la struttura implementata all'interno di un pic32 è molto più complessa di questa semplice funzione. Di seguito è riportato il contenuto di ciascun Timer interno:

- **Contatore 16 bit**
- **Selezione clock interno o esterno**
- **Programmazione degli interrupt**
- **Conteggio impulsi esterni**

Vengono messi a disposizione dell'utente, due tipologie di Timer a seconda delle esigenze applicative, le cui caratteristiche sono rappresentare in (Fig. **??**).

Come si può notare, tutti i timer presenti hanno un contatore a 16 bit, mentre per generarne uno da 32, occorre affiancarne due di tipo B. Il microcontrollore da noi usato, possiede 5 timer, tra i quali solo il **TMR1** è di tipo A. Come si è visto, a parte qualche funzione, tutti i timer lavorano allo stesso modo, ma noi ci limiteremo ad analizzare la tipologia A di (Fig. 1.9).

Available Timer Types	Secondary Oscillator	Asynchronous External Clock	Synchronous External Clock	16-Bit Synchronous Timer/Counter	32-Bit(1) Synchronous Timer/Counter	Gated Timer	Special Event Trigger
Type A	Yes	Yes	Yes	Yes	No	Yes	No
Type B	No	No	Yes	Yes	Yes	Yes	Yes

Figura 1.8: Tipologia di Timer

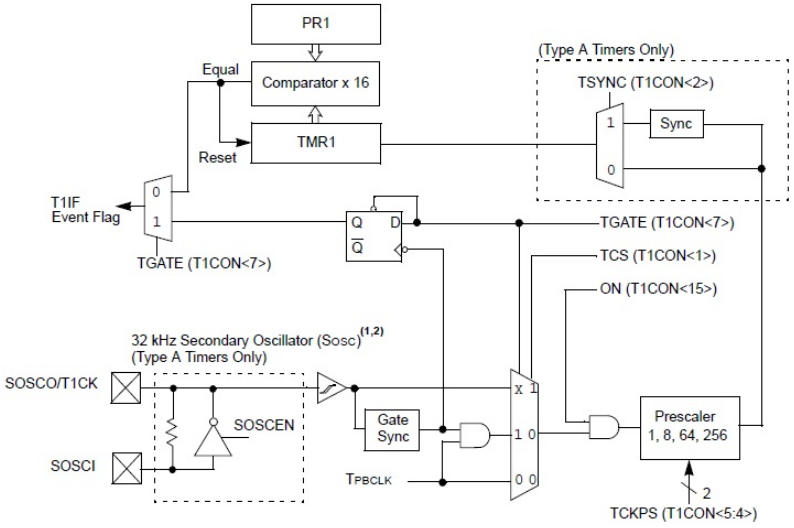


Figura 1.9: Schema a blocchi del Timer

Il Timer viene attivato con $TON = 1$, in quanto questo bit è posto in ingresso ad una porta *and*. Il secondo ingresso della porta definisce il tipo di clock utilizzato come base del conteggio. Se volessimo utilizzare il clock interno(PBCLK), in tal caso occorre impostare $TCS = 0$, $TGATE = 0$. Il valore di TGATE risulta indifferente per $TCS = 1$, in caso contrario con $TGATE = 1$ il conteggio si arresta, in mancanza della sorgente di clock. Spesso il clock interno possiede una frequenza molto elevata, per cui in cascata è presente un prescaler con valori da 1 a 256 per ridurre la base dei tempi, consentendo di ottenere δ T più elevati. Per i timer di tipo A è presente un bit **TSYNC** il quale, posto a 1, permette di selezionare il sincronismo con un clock esterno, se presente. Tutte le configurazioni precedenti sono settabili mediante il registro di programmazione **T1CON**. Arrivati nel cuore del dispositivo, vi sono due registri molto importanti, che gestiscono le operazioni di conteggio:

- **TMR1**

  E' un registro a 16 bit, il quale contiene il valore del conteggio istante per istante.

- **PR1**

  Esso definisce il valore finale del conteggio, dopo il quale il TMR1 riparte da 0.

Pertanto, quando si usa un timer ci si chiede quanto deve contare, per ottenere un determinato intervallo di tempo. Per definire ciò si parte dal clock utilizzato dal contatore:

$$T_{OSC} = T_{TIMER} = \frac{1}{F_{OSC}} \tag{1.5}$$

Considerando la possibilità di introdurre un fattore di prescaler, per incrementare il tempo di clock, qualora non si riesca a realizzare la tempistica richiesta:

$$T_{PRESCALER} = T_{OSC} * f_{PS} \tag{1.6}$$

A questo punto se volessimo realizzare un tempo ΔT generico, possiamo ricavare il numero da inserire nel TMR1:

$$CONTEGGI = \frac{\Delta T}{T_{PRESCALER}} \tag{1.7}$$

$$TMR1 = (2^{16} + 1) - CONTEGGI \tag{1.8}$$

Dal momento che in tale registro si possono caricare solo numeri interi, sarà necessario troncare il numero nel caso possieda valori decimali. In alternativa è possibile utilizzare il registro PR1, caricando in esso il valore dei conteggi da eseguire.

Spesso nell'utilizzo del Timer si è interessati al suo stato di overflow. Di fatto è stato predisposto il bit **T1IF** che identifica tale evento, consentendo all'utente di gestire un eventuale interrupt associato.

Capitolo 2

Comunicazione Pic-PC

Nel capitolo ci prepoccuperemo di fornire i dettagli della comunicazione tra il microcontrollore e il Pc, mediante la periferica USB, definendo i protocolli di comunicazione utilizzati e le varie fasi implementative affrontate.

2.1 Classe CDC

L'Universal Serial Bus (USB) ha semplificato notevolmente, per gli utenti finali, il collegamento di una periferica ad un Personal Computer, in particolare per la dimensione compatta e universale dei sui cavi. Di fatto con gli ultimi anni, la porta fisica seriale COM è diventata sempre più rara, tant'è che nei pc moderni non viene più implementata. Questo potrebbe presentare un problema per un progettista, che necessita di una comunicazione seriale da una periferica ad un calcolatore. Fortunatamente, vi è la possibilità di utilizzare la USB *Communication Class Device*(**CDC**), che consente di mantenere la semplicità strutturale dell' USB fornendo, però, le funzionalità di una porta COM. L'insieme delle funzioni descritte dalla classe, consentono di emulare una porta seriale, creando una UART(*Universal Asynchronous Receiver-Transmitter*) virtuale. L'UART è un dispositivo hardware, che converte i flussi di bit di dati da un formato parallelo ad un formato seriale asincrono o viceversa, il quale veniva implementato nelle interfacce RS-232. La notevole flessibilità e potenza dell'USB richiede, però, numerosi protocolli di gestione per l'identificazione dei dispositivi, la loro configurazione, il controllo e il trasferimento dati. La Microchip, ha raccolto in un unica libreria chiamata *USB Framework*, l'insieme di questi protocolli e delle funzionalità implementabili con l'USB, agevolando l'operato degli sviluppatori.

In (Fig. 2.1) sono riportati tutti i file di tipo sorgente e header, necessari per implementare la classe CDC. Questi vengono inseriti nel progetto software all'interno dell'ambiente di sviluppo MPLAB utilizzato. In particolare per instaurare una corretta trasmissione faremo uso di alcune funzioni:

- **USBInitialize()**

  La funzione consente di inizializzare lo stack USB, dallo strato fisico a quello applicativo. Essa deve essere chiamata come prima funzione, ai fini di rendere operativo il modulo USB.

File	Description
usb_device.c	USB device layer (device abstraction and *"Universal Serial Bus Specification, Revision 2.0, Chapter 9"* protocol handling)
usb_hal.c	USB Hardware Abstraction Layer (HAL) interface support
usb_hal_core.c	USB controller functions, used by HAL interface support
usb_device_local.h	Private definitions for USB device layer
usb_hal_core.h	Private definitions for HAL controller core
usb_hal_local.h	Private definitions for HAL
usb.h	Overall USB header (includes all other USB headers)
usb_ch9.h	USB device framework (*"Universal Serial Bus Specification, Revision 2.0, Chapter 9"*) definitions
usb_common.h	Common USB stack definitions
usb_device.h	USB device layer interface definition
usb_hal.h	USB HAL interface definition
usb_device_cdc_serial.h	CDC serial function driver API header
usb_func_serial.c	CDC serial function driver implementation
usb_func_serial_local.h	Private definitions for CDC serial function driver
HardwareProfile.h	Hardware configuration parameters
io_cfg.h	Macros supporting use of GPIO bits connected to switches
main.c	Primary application source file
usb_config.h	Application-specific USB configuration options (see**"USB Firmware Stack Configuration"**)
usb_app.c	Application-specific USB support

Figura 2.1: File sorgente utilizzati per implementare una classe CDC

- **USBDeviceAttach()**

  Consente di definire se la periferica USB è stata collegata o meno all'Host Pc, mediante un controllo sul VBUS. Questa funzione, deve essere chiamata solo se si sta utilizzando la periferica USB con modalità Interrupt.

- **USBDeviceTask()**

  E' responsabile del controllo e della gestione della comunicazione USB. Essa consente la corretta trasmissione e ricezione dei pacchetti inviati e di fornire i dettagli dell'enumerazione USB. Infatti quando una periferica viene collegata per la prima volta ad un calcolatore, questo pone delle domande al device relative alla sua originalità(costruttore), i numeri VID(*Vendor ID*) e PID(*Product ID*),contenuti nel file usb-descriptors.c, che identificano il prodotto. Inoltre la funzione rileva alcuni eventi relativi alla connessione, come lo stato di *sospensione*. Se si gestisce il modulo ad interrupt, questa macro viene chiamata in seguito allo scatenarsi di ogni evento di interruzione, nel caso, invece, di utilizzo a polling, risulta necessario richiamarla con una frequenza tale per cui il registro USTAT non raggiunga il riempimento. Il registro è una finestra di lettura di 4 byte FIFO di stato, contenuto nel SIE. Quando la Serial Interface Engine sta completando un trasferimento, mentre allo stesso tempo riceve un'ulteriore dato da inviare, segnala il processo all'interno dell'USTAT. Per questo motivo la funzione USBDeviceTasks(), che detiene il controllo, deve essere richiamata in maniera ciclica, in particolare ogni circa 10ms.

- **USBUSARTIsTxTrfReady()**

  Questa funzione, esegue un controllo sulla linea, per verificare se il modulo USB è pronto ad inviare un dato. Essa ritorna 1 se la periferica è libera per trasmettere dati, altrimenti 0 se è occupata. Ciò significa che deve essere chiamata, sempre, prima dell'invio di un pacchetto.

- **putUSBUSART(char *data, BYTE length)**

  La trasmissione avviene in maniera diretta tramite questa macro. Essa permette di scrivere dati dalla RAM al modulo USB. L'espressione *char *data* rappresenta il puntatore alla variabile o array presente in memoria, in cui è contenuta l'informazione da trasmettere. Infatti, il secondo parametro da fornire, rappresenta proprio la lunghezza dell'Array. Questa procedura, consente il trasferimento di un massimo di 64 byte per ogni chiamata di funzione.

- **getsUSBUSART(char *buffer, BYTE len)**

  Viene utilizzata per leggere dati che arrivano in ingresso al modulo USB. La funzione accetta come parametri: il puntatore alla struttura dati della variabile o Array, in cui scrivere i dati e il numero di byte che ci si aspetta siano presenti in lettura. Nel caso in cui i byte ricevuti siano in quantità maggiore, verranno letti solo quelli richiesti. La funzione ritorna il valore 0 se non è presente nessun dato da leggere.

  Oltretutto può essere utile il controllo anche sulla variabile di sistema *USBDeviceState*. Tale variabile viene aggiornata in base allo stato di configurazione del sistema:

1. **Powered-State**
   Il cavo USB è collegato, ma il sistema non è stato ancora configurato dall'Host.
2. **Default-State**
   Il sistema è in stato di Default, per permettere l'inizio della comunicazione da parte dell'Host.
3. **Address-State**
   L'Host ha iniziato la comunicazione con il sistema, per poter configurare il device e il sistema opertativo. In questa fase avviene il caricamento dei driver del dispositivo.
4. **Configurated-State**
   L'hardware è stato configurato con successo e pronto per l'uso.

2.2 Trasferimento dati al Pc

Le informazioni vengono trasmesse a byte nell'ordine MSB e LSB, i quali compongono un dato di 16 bit. Essi sono rappresentati con combinazioni da 0001h a FFFEh ($2^{16} - 2$), mentre le configurazioni 0000h e FFFFh sono dedicate esclusivamente alla segnalazione del sincronismo (Fig. 2.2).

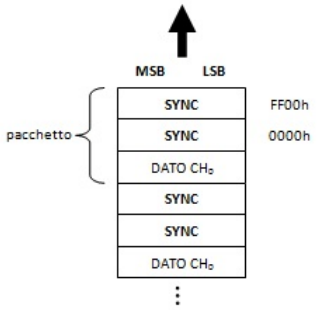


Figura 2.2: Protocollo di comunicazione Pic-Pc

Per implementare la comunicazione, è stato predisposto in memoria un array di 6 elementi, contenente i 4 byte del sincronismo e i due byte del dato in uscita dal filtraggio digitale. Da specifica, inoltre, il sensore ibrido Berillio, lavora con 4 bande di frequenza differenti (625, 1250, 5000, 10000) Hz, per cui in (Fig.2.3) sono rappresentati i tempi di trasmissione per ciascun pacchetto inviato, con associata la byte-rate raggiunta.

banda	Byte-rate
625Hz	6 byte (un pacchetto) ogni 819,2 µs → 7324,2 Byte/sec
1250Hz	6 byte (un pacchetto) ogni 409,6 µs → 14648,4 Byte/sec
5000Hz	6 byte (un pacchetto) ogni 102,4 µs → 58593,7 Byte/sec
10000Hz	6 byte (un pacchetto) ogni 51,2 µs → 117187,5 Byte/sec

Figura 2.3: Byte-rate da Pic a Pc

Per ottenere le tempistiche richieste, si è partiti con l'idea di scatenare un interrupt sull'overflow del TMR1. In particolare, faremo in modo che il conteggio eseguito dal timer, generi degli intervalli di tempo ΔT, come da specifica, al termine dei quali si proceda con l'invio dei dati al Pc. Prendiamo in considerazione i dati tecnici sui pic32, presenti nel (Cap. 1), per ottenere i seguenti risultati:

- **625 Hz (6 pacchetti ogni 819.2us)**

$$F_{CLK} = F_{TMR1} = 80MHz \Rightarrow TMR1 = 12.5ns \tag{2.1}$$

$$CONTEGGI = \frac{819.2 * 10^{-6}}{12.5 * 10^{-9}} = 65536 \tag{2.2}$$

  I calcoli sono stati condotti con un prescaler unitario. Da notare il valore ottenuto, è a tutti gli effetti l'intero contenuto di un registro a 16 bit, per cui il TMR1 viene fatto partire da 0 ed incrementare fino al valore finale del conteggio.

- **1250 Hz (6 pacchetti ogni 409.6us)**

$$CONTEGGIO = \frac{409.6 * 10^{-6}}{12.5 * 10^{-9}} = 32768 \tag{2.3}$$

$$TMR1 = (2^{16} + 1) - CONTEGGIO = 32769 \tag{2.4}$$

- **5000 Hz (6 pacchetti ogni 102.4us)**

$$CONTEGGIO = \frac{102.4 * 10^{-6}}{12.5 * 10^{-9}} = 8192 \tag{2.5}$$

$$TMR1 = (2^{16} + 1) - CONTEGGIO = 57345 \tag{2.6}$$

- **10000 Hz (6 pacchetti ogni 51.2us)**

$$CONTEGGIO = \frac{51.2 * 10^{-6}}{12.5 * 10^{-9}} = 4096 \tag{2.7}$$

$$TMR1 = (2^{16} + 1) - CONTEGGIO = 61441 \tag{2.8}$$

L'approccio fin'ora utilizzato, è servito principalmente per consentirci di instaurare una prima comunicazione con il Pc e verificare il funzionamento del trasferimento dati sulla porta seriale. Verificato ciò, purtroppo questo percorso deve essere abbandonato. Infatti, il protocollo utilizzato per la trasmissione presenta un elevato *Overhead*, il quale richiede maggiori risorse di calcolo al Pc, per l'elaborazione dei dati, più di quanto in realtà sarebbero necessarie. Per far fronte a questo particolare inconveniente, si è definito un nuovo protocollo di comunicazione USB che andasse a limitare il problema riscontrato (Fig. 2.4).

Per testare la correttezza del protocollo si può far riferimento alla (Fig. 2.5). Esso rappresenta il flusso di informazioni sulla porta COM del Pc, riconosciuto da un un generico software che monitorizza la seriale. Per rendere il protocollo più *user friendly* possibile, a che deve gestire l'interfaccia di acquisizione dati lato PC, il flusso delle informazioni, tra un sincronismo e l'altro, è realizzato mediante un contatore con valori che vanno da 0 a 16384 (come da nuovo protocollo).

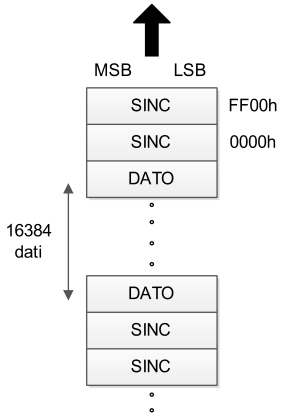


Figura 2.4: Protocollo di comunicazione Pic-Pc con ridotto Overhead

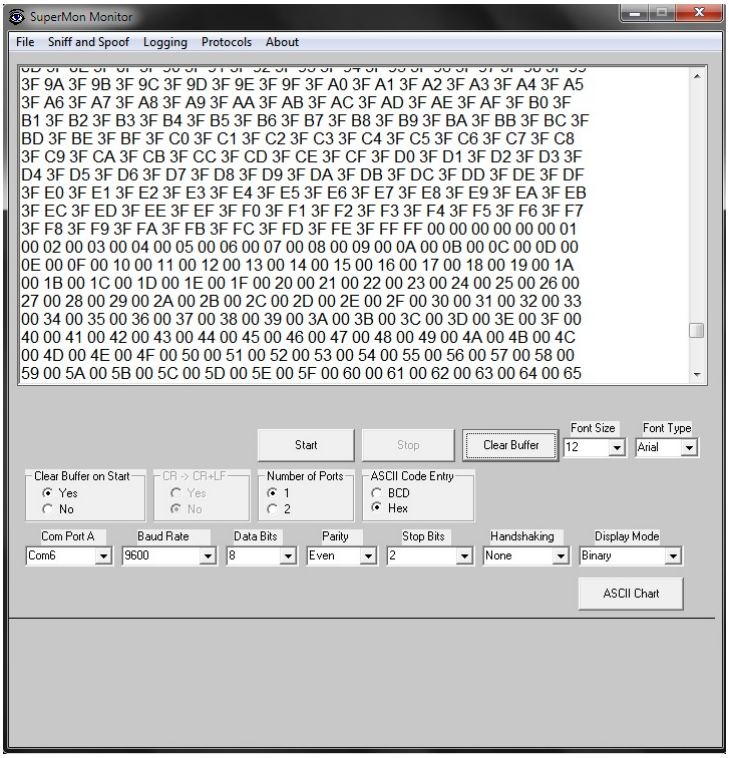


Figura 2.5: Flusso dati su porta seriale COM

2.3 Ricezione dati dal Pc

I dati vengono trasmessi dal Pc con pacchetti di un byte alla volta. Le informazioni sono rappresentate da combinazioni da 00h a 7Fh, mentre la configurazione 80h è proibita in quanto è riservata per il byte di start (Fig. 2.6).

Si noti che lo stream di dati è composto da 12 byte, compreso lo start, tutti quanti adibiti alla programmazione del chip Berillio. In particolare, vogliamo

	MSb							LSb
START	80h							
CONFIG0	0	En_trial	TrialType2	TrialType1	TrialType0	Amp9	Amp8	Amp7
CONFIG1	0	Amp6	Amp5	Amp4	Amp3	Amp2	Amp1	Amp0
CONFIG2	0	T0-2	T0-1	T0-0	T1-3	T1-2	T1-1	T1-0
CONFIG3	0	Delta6	Delta5	Delta4	Delta3	Delta2	Delta1	Delta0
CONFIG4	0	Tag6	Tag5	Tag4	Tag3	Tag2	Tag1	Tag0
CONFIG5	0	free	free	free	free	free	free	free
ChipSelect	0	free	free	free	free	free	free	free
D0	0	DAC6	DAC5	DAC4	DAC3	DAC2	DAC1	DAC0
D1	0	0	0	0	0	DAC9	DAC8	DAC7
D2	0	EN_triang	ShuntCDS	0	ShortVc	Vc_sel	EN_Sum	EN_dComp
D3	0	EN_ExtRg	ShuntLPF	Dis_CDS_DS	100k	Range0	Band1	Band0

Figura 2.6: Flusso dati in ricezione

soffermarci su due bit **Band0** e **Band1**, i quali consentono la selezione della banda di frequenze voluta, secondo la configurazione di (Fig. 2.7).

BAND1	BAND0	FREQ.
0	0	10 KHz
0	1	5 KHz
1	0	1.25 KHz
1	1	625 Hz

Figura 2.7: Selezione delle bande di frequenze

Fino ad ora, abbiamo gestito la comunicazione USB, con il Pc, mediante la programmazione di un timer, il cui stato di overflow scatena un interrupt. Capiamo bene che questa tecnica, seppur utilizzata in precedenza per creare un interfaccia USB funzionante, ora, avendo a nostra disposizione i bit *Band0* e *Band1*, faremo in modo che siano loro a definire con quale banda si debba trasmettere. In particolare essi consentiranno, come vedremo nel prossimo capitolo, di individuare un determinato **OSR**(*Over Sampling Ratio*) del filtraggio digitale, sul quale viene eseguita la trasmissione dati in cascata. Inoltre, una volta che avremo a disposizione i dati del filtraggio, che rappresentano il nostro segnale di interesse, essi verranno inseriti al posto del contatore, utilizzato solo ed esclusivamente in questa fase di debug.

Capitolo 3

Filtraggio digitale Sinc3

Prima di definire gli elementi teorici e implementativi del filtraggio digitale di tipo sinc^3, cerchiamo di capire quali sono le motivazioni che ci portano ad adottare questo tipo di filtro, conoscendo meglio le caratteristiche del segnale $\Delta\Sigma$ presente in uscita al chip Berillio.

3.1 Modulatore $\Delta\Sigma$

Il modulatore $\Delta\Sigma$ converte un segnale analogico in un segnale digitale composto da uni e zeri. Di fatto, il duty-cycle del segnale modulato in uscita, è una funzione del segnale analogico in ingresso (Fig. 3.1).

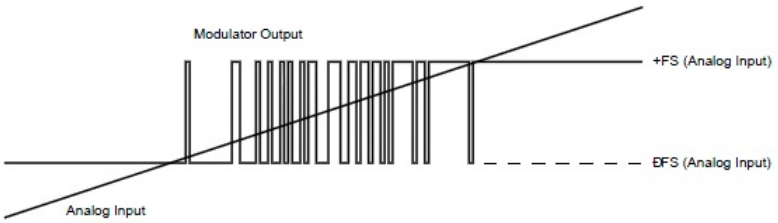


Figura 3.1: Tipico segnale di uscita di un modulatore $\Delta\Sigma$ in funzione di un generico segnale analogico in ingresso

Mediante le tecniche di *Oversampling* e *Noise Shaping* si riduce il rumore di quantizzazione, introdotto nel processo di conversione, all'interno della banda di interesse. Ciò si spiega dal fatto che la densità spettrale di potenza del rumore di quantizzazione è costante e dipende solo dalla risoluzione del quantizzatore, per cui si può fare in modo che il rumore venga distribuito in frequenza. Applicando un filtro digitale in uscita, siamo in grado di filtrare il rumore al di fuori della banda richiesta, con l'ulteriore effetto di riuscire ad aumentare la risoluzione, a discapito però della frequenza(*decimazione*).

Topologia

Lo schema di (Fig. 3.2), descrive una soluzione tipica per implementare un modulatore $\Delta\Sigma$ del secondo ordine. Esso è costituito da due stadi integratori, un

comparatore e un convertitore DAC(*Digital to Analog Converter*), con risoluzione di un bit. La sua uscita viene sottratta in due nodi, corrispondenti agli ingressi dei due integratori, con il segnale analogico in ingresso, che ci fornisce X_2 e con l'uscita del primo integratore, il cui risultato produce X_3. In funzione dell'ingresso e al tipo di integratori utilizzati(invertenti o non invertenti), questi generano delle rampe di tensione (positive o negative). Quando il valore del segnale X_4, eguaglia il valore della tensione di riferimento V_{REF}, l'uscita del comparatore si inverte da positiva a negativa o viceversa, in funzione del valore assunto in precedenza. Allo stesso modo il convertitore DAC, cambia la propria uscita X_6, causando un processo inverso di integrazione. Il feedback del modulatore, forza il valore dell'uscita degli integratori a monitorare il valore medio dell'ingresso.

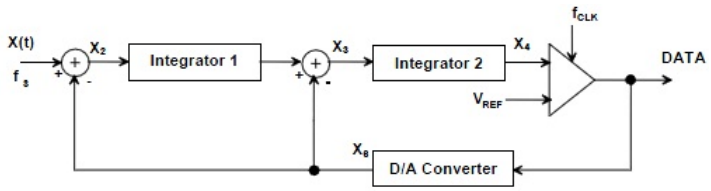


Figura 3.2: Diagramma a blocchi di un modulatore $\Delta\Sigma$ del secondo ordine

3.2 Processo di decimazione

Abbiamo già espresso in precedenza la necessità di porre in cascata al modulatore $\Delta\Sigma$, un processo di decimazione per ridurre il rumore di quantizzazione in banda e aumentare la risoluzione in termini di bit per campione. Il *fattore di decimazione* è rappresentato da un intero **M**(definito anche con il nome di *OSR*), il quale riduce la frequenza di campionamento dello stesso valore che esso assume.

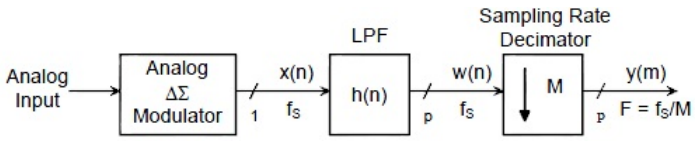


Figura 3.3: Schema a blocchi di un decimatore per Convertitore $\Delta\Sigma$

Con riferimento alla (Fig. 3.3), il bit stream di dati in uscita al modulatore, si presenta all'ingresso del filtro passa basso con una frequenza di campionamento pari a f_S. La frequenza di taglio del filtro è di $\frac{\pi}{M}$, dove π è la frequenza normalizzata (radianti) corrispondente alla frequenza di Nyquist o alla metà della frequenza di campionamento. Il filtro rimuove tutta l'energia associata alle componenti di rumore, al di sopra della frequenza di taglio e evita la formazione di *aliasing* nel processo di decimazione, quando il segnale w(n) è ricampionato dal *Sampling Rate Decimator*. Quello che per adesso abbiamo chiamato con il nome di processo di decimazione, verrà implementato all'interno del pic32 con un filtro sinc^K.

3.3 Filtro digitale Sinc^K

Questo tipo di filtro è molto utilizzato, grazie alla sua semplicità implementativa sia hardware sia software. Di fatto, non richiede l'ausilio di moltiplicatori digitali, ma è costituito da K stadi accumulatori in cascata, operanti alla frequenza di campionamento f_S, seguiti da altrettanti stadi differenziali, i quali lavorano, però, ad una frequenza pari a $\frac{f_s}{M}$ (Fig. 3.4).

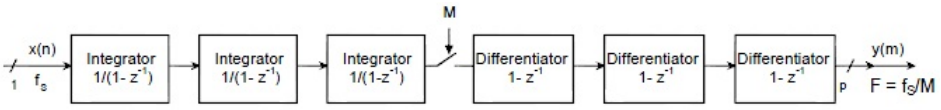


Figura 3.4: Topologia filtro digitale Sinc^3

La relazione che sussiste tra la frequenza di campionamento f_S e la *data rate* in uscita è definita dalla seguente relazione:

$$DataRate = \frac{f_S}{M} \tag{3.1}$$

E' possibile, agendo sul fattore di decimazione, modificare la selettività del filtro. Tuttavia, nella scelta dell'ordine del filtro, è necessario conoscere l'ordine del modulatore $\Delta\Sigma$ dal quale provengono i dati. L'ordine K del filtro sinc_K, deve essere almeno di un ordine maggiore di quello del modulatore $\Delta\Sigma$, ai fini di garantire un ridotto *aliasing* del rumore fuori banda sovrapposto in banda base.

$$K \geq 1 + (\text{order}\Delta\Sigma) \tag{3.2}$$

L'*output word size* del filtro digitale Sinc^K, è definibile mediante la seguente espressione:

$$p = K * log_2(M) \tag{3.3}$$

Il punto di risposta del filtro a -3 dB è fortemente dipendente dall'ordine del filtro stesso e meno dal suo rapporto di decimazione M. Sulla base delle considerazioni fatte, sapendo che il Berillio offre un modulatore $\Delta\Sigma$ del secondo ordine, si ricava facilmente che il filtro da utilizzare per il processo di decimazione, deve essere di tipo **Sinc^3**. Si possono ottenere buoni risultati in termini di selettività del filtraggio, infatti, aumentando il fattore di decimazione si raggiungono frequenze di risposta del filtro a -3 dB nell'intorno di pochi KHz. Un'altra peculiarità risiede nella forte attenuazione dei lobi secondari, come si nota dalla (Fig. 3.5), dopo appena 7 lobi l'attenuazione raggiunge circa i - 80 dB.

3.4 Implementazione filtro Sinc^3

Rispetto a ciò che abbiamo visto in precedenza, parliamo ora delle fasi che ci consentiranno di realizzare un corretto filtraggio all'interno del microcontrollore. E' stato appena definito l'output word size in funzione del fattore di decimazione e dell'ordine del filtro. In realtà se volessimo far in modo che non vi siano

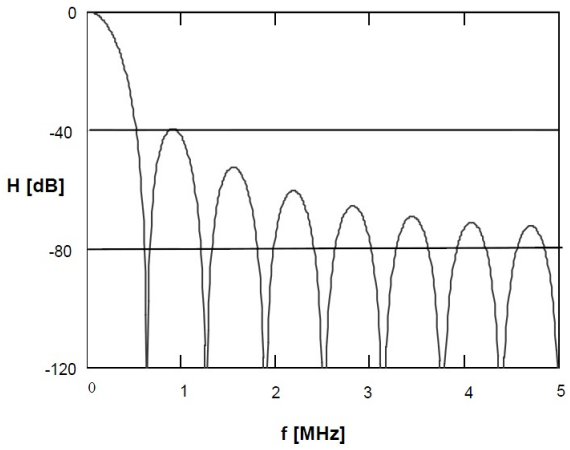


Figura 3.5: Risposta in frequenza filtro Sinc^3 con $M = 16$

perdite di campioni, è necessario che il *bus* interno del pic, dedicato all'uscita del filtraggio, abbia una dimensione in termini di bit pari a:

$$BusWidth = 1 + K * log_2(M) \tag{3.4}$$

Si noti che ogni valore dell'output word size è incrementato di uno. E' sempre meglio disporre il dato in un registro le cui dimensioni siano maggiori del dato stesso, onde evitare che eventuali bit di riporto vengano persi durante le somme e le sottrazioni eseguite. Prendendo spunto dalla (Fig. 3.4), per realizzare il filtraggio, si possono definire due fasi:

1. **Accumulator**

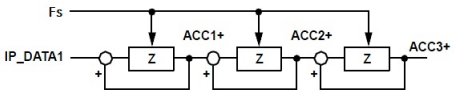


Figura 3.6: Stadio Accumulatore

Si predispongono 3 variabili da 32 bit ciascuna(il cui valore è scelto in funzione del massimo OSR utilizzato), per eseguire le somme alla frequenza di campionamento:

```
acc1 = acc1 + ip_data1;
acc2 = acc2 + acc1;
acc3 = acc3 + acc2;
```

2. **Differentiator**

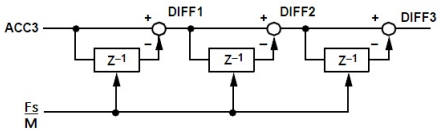


Figura 3.7: Stadio Differenziatore

```
diff1 = acc3 - acc3_d2;
diff2 = diff1 - diff1_d;
diff3 = diff2 - diff2_d;
acc3_d2 = acc3;
diff1_d = diff1;
diff2_d = diff2;
```

Anche in questo caso tutte le variabili in gioco hanno un'estensione di 32 bit, quelle contrassegnate da una lettera *d* finale rappresentano il dato precedente

Si è scelto di definire un bus a 32 bit, per ovvie ragioni dovute all'*Over Sampling Ratio*, infatti, noi ci ricordiamo che dobbiamo implementare 4 differenti bande di frequenze, per ognuna delle quali è associabile un determinato valore di OSR (Fig. 3.8).

BANDA (Hz)	OSR	BUS WIDTH (bits)
625	1024	31
1250	512	28
5000	128	22
10000	64	19

Figura 3.8: Larghezza del Bus interno in funzione del rapporto di decimazione

Tra i bit facenti parte del *bus width*, vengono prelevati solo i 16 più significativi, suddivisi in due byte(MSB e LSB), pronti per essere inviati secondo il protocollo di trasmissione USB, definito nel precedente capitolo.

3.5 Test del filtraggio

Per poter essere in grado di condurre un test sul filtraggio poco fa realizzato, occorre disporre di un segnale $\Delta\Sigma$ da applicare in ingresso. Sebbene questo passaggio sia la soluzione finale voluta, non è affatto utile per definire eventuali errori presenti nel processo. Per cui si è pensato, di generare dei segnali di test di tipo onda quadra a duty-cycle variabile. Fornendo tali forme d'onda in ingresso al filtro, quest'ultimo deve rispondere con il corrispondente valor medio del segnale applicato. Le forme d'onda di test, sono state generate esternamente, mediante l'ausilio di un modulo FPGA, onde evitare di portar via risorse di

calcolo utili al microcontrollore, nella gestione del filtro e delle periferiche in uso. Un particolare importante è che i segnali generati siano tra loro correlati, in funzione del periodo T del segnale di sincronismo di $f = 1.25MHz$. Questo approccio è indispensabile, per eseguire le somme e le differenze in maniera corretta, in ogni istante di campionamento(salita o discesa del sincronismo). Potevamo realizzare, delle altre forme d'onda sempre con i duty-cycle richiesti, ma che mantenessero la stessa frequenza del segnale di sincronismo. Ciò però non avrebbe consentito di riconoscere il loro corretto valor medio. Infatti, come riportato in (Fig. 3.9), se campioniamo sul fronte di discesa del sincronismo e prendiamo in considerazione l'ultimo segnale, quello non correlato, notiamo che in realtà invece che avere in uscita al filtro, un valore medio pari al 75%, si ottiene lo stesso identico risultato di un segnale costantemente a livello alto, il che risulta una valutazione errata.

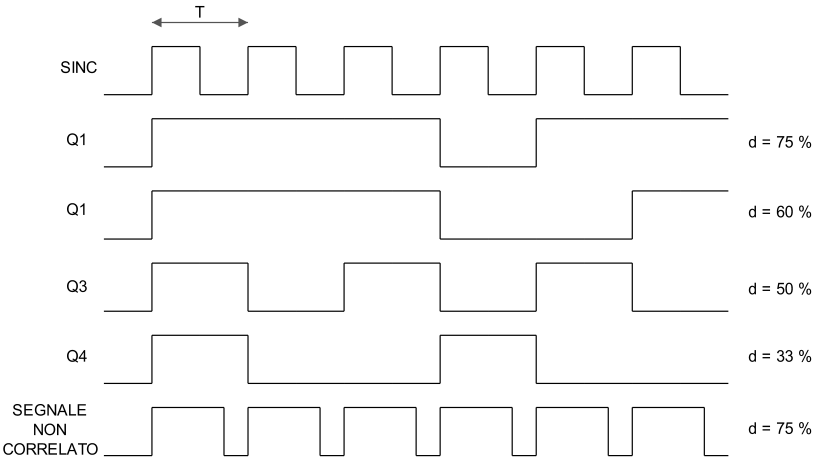


Figura 3.9: Forme d'onda dei segnali di Test per il filtraggio digitale $Sinc^3$

La nostra preoccupazione, a questo punto, è quella di gestire in maniera opportuna i segnali in ingresso al microcontrollore. Si è pensato, almeno inizialmente, di portare il segnale di sincronismo in un ingresso adibito agli interrupt esterni. Questo ci consente di programmare l'evento di interrupt sul fronte di salita o discesa del *Sinc*. Le forme d'onda da filtrare, sono poste, invece, in un comune ingresso di I/O del pic. Una volta dentro alla ISR, si campiona il dato utile e si eseguono le sole somme. In base ai valori già discussi di *band0* e *band1* viene definito l'OSR, per cui, nel caso ad esempio sia stato selezionato $M = 1024$, un contatore farà in modo che ad ogni 1024 accumulazioni eseguite, vi sia un processo di differenziazione. Esso, inoltre, è seguito dal trasferimento del dato filtrato via USB al Pc. Tuttavia, la tecnica implementata, presenta diverse problematiche. Verificando la trasmissione USB mediante il monitor della seriale, il flusso dati risulta molto lento e presenta delle incongruenze nei valori, i quali dovrebbero essere byte tutti uguali, alla luce del fatto che si vuole rappresentare un valor medio. Nella ricerca dell'errore, è stato tolto il filtro, per vedere se le forme d'onda in ingresso fossero campionate in maniera adeguata e i valori inviati correttamente al Pc. Mediante l'ausilio dell'interfaccia software di gestione del chip Berillio, sono state visualizzate le forme d'onda, sulle quali si presenta un vistoso rumore. A quel punto, consapevoli che il problema non

risiede nel filtro, è stata condotta un'ultima prova di verifica, portando su un pin esterno del pic i dati campionati di un segnale costantemente a livello alto. Con oscilloscopio si è affiancato il segnale di sincronismo, che definisce la frequenza di campionamento e i campioni effettivamente rilevati dal microcontrollore. Il risultato ci ha messo in evidenza ciò che già pensavamo, ovvero una netta perdita di campioni. In realtà, è più che comprensibile il risultato ottenuto, di fatto per entrare e uscire da una interrupt service routine il pic impiega circa 60 istruzioni assemply. Delle 64 disponibile (vedi Cap. 1) ne rimangono 4 per gestire il filtro, le quali non sono sufficienti. Constatato, quindi, che il dispositivo non riesce a campionare un dato alla frequenza di $f_S = 1.25MHz$, per cui, non potendo rinunciare ad un eventuale perdita di dati, si è studiato un metodo alternativo che consentisse di risolvere il problema. L'idea può essere schematizzabile in due passaggi:

1. Apertura di un modulo SPI in modalità slave a 32 bit, il cui ingresso SCK è rappresentato dal segnale di sincronismo f_S, mentre nella porta SDI, inseriamo il segnale $\Delta\Sigma$.

2. Apertura di un modulo timer, il quale opera sulla frequenza PBCLK interna. Si vuole calcolare il tempo che L'SPI impiega per riempire il registro a scorrimento con 32 dati, per fare in modo che al termine di tale processo venga innescato un interrupt. Dentro la rountine si prelevano i dati memorizzati nell' SPIxBUF utilizzato, realizzando con essi un ciclo for, ai fini di eseguire le somme bit a bit per il filtraggio. Sapendo che tra un campionamento e l'altro vi sono 800 ns ($\frac{1}{f_S}$), l'interrupt in questo caso viene chiamato ogni:

$$\Delta T = 800 * 10^{-9} * 32 = 25.6 * 10^{-6} \tag{3.5}$$

   Questo intervallo di tempo, ben più ampio rispetto al precedente, ci permette di avere un quantitativo di istruzioni tale, da gestire la totalità del filtraggio.

Si è dato per scontato che la base dei tempi del timer sia la stessa del clock dell'SPI(f_S), definiamola per ora come un'esigenza progettuale, alla quale si daranno delle risposte nel (Cap. 4). Tuttavia, sono state abbandonate le funzioni di test utilizzate in precedenza, per condurre un'analisi che si avvicini sempre più alla fase finale del progetto, adoperando direttamente il segnale $\Delta\Sigma$. Quest'ultimo però, viene fornito esclusivamente dal chip Berillio, per cui i test associati a questo tipo di implementazione, sono stati condotti direttamente sul circuito dell' **Amplificatore Integrato**, le cui caratteristiche sono discusse nel (Cap. 5.1).

Capitolo 4

Comunicazione Pic-Berillio

Il chip Berillio implementa al suo interno un interfaccia SPI, per ricevere i dati di configurazione provenienti dal Pc. Per cui faremo in modo che, il microcontrollore, gestisca anch'esso un modulo SPI che consenta la corretta comunicazione. Sono state menzionate diverse modalità operative nel corso del (Cap 1), vedremo ora di definire in dettaglio la Framed mode, la quale risponde alle nostre esigenze progettuali.

4.1 Framed mode

Essa rappresenta un caso particolare della funzione master mode. In definitiva l'unico vero e proprio cambiamento risiede nella gestione del clock per l'invio dell'informazione. Infatti nella Framed mode, terminata la trasmissione dati, il clock continua ad essere operativo. Questo implica che per definire l'inizio della trasmissione occorre lo slave select, il cui fronte di salita/discesa diventa determinante (Fig. 4.1). Questa esigenza nasce da una problematica hardware, durante la fase realizzativa dell'Amplificatore integrato. Infatti, il microcontrollore richiede un oscillatore esterno che abbia un valore che sia in funzione della potenza del 2 (4,8,16..)MHz, a causa del fatto che il modulo USB lavora ad una frequenza di 48MHz, diversamente non ottenibile. Invece il sensore ibrido Berillio opera con una frequenza di 10 MHz, per cui, ai fini di evitare l'istallazione di due oscillatori, i quali causerebbero notevoli problemi di sincronismo tra i dispositivi, si è scelto di predisporre un oscillatore di 8 MHz, che servisse solo il microcontrollore. Sulla base dei tempi dell'oscillatore esterno, viene generato internamente il clock dedicato alla periferica SPI, volutamente fissato ad un valore di $SCK = 10$ MHz, il quale deve servire come segnale di sincronismo dell'SPI per la comunicazione seriale, oltre alla richiesta di fornire il clock a Berillio. Per programmare il modulo SPI, ci si serve del primo registro di controllo SPIxCON. Si descrivono qui di seguito le configurazioni più importati selezionate:

- bit 31 **FRMEN**: abilita la Framed mode (1)
- bit 30 **FRMSYNC**: si definisce lo SS come output (0)
- bit 29 **FRMPOL**: impostiamo lo SS come attivo alto (1)

- bit 27 **FRMSYPW**: permette di scegliere se l'impulso dello SS rimanga attivo per un solo periodo di clock oppure per tutta la lunghezza della parola da trasmettere (selezionabile mediante i bit MODE$< 32, 16 >$). Nel nostro caso abbiamo bisogno che lo slave select rimanga attivo per tutta la durata della comunicazione, in quanto Berillio ignora tutti i dati che transitano in ingresso quando lo SS è a livello logico basso. (1)
- bit 23 **MCLKSEL**: selezione del PBCLK per la Baud Rate Generator(0)
- bit 15 **ON**: accensione del modulo SPI (1)
- bit 11-10 **MODE**$< 32, 16 >$: configuriamo la modalità a 32 bit (10)
- bit 8 **CKP**: stato *idle* per il clock è a livello basso, mentre stato attivo è a livello alto (0)
- bit 5 **MSTEN**: master mode (1)

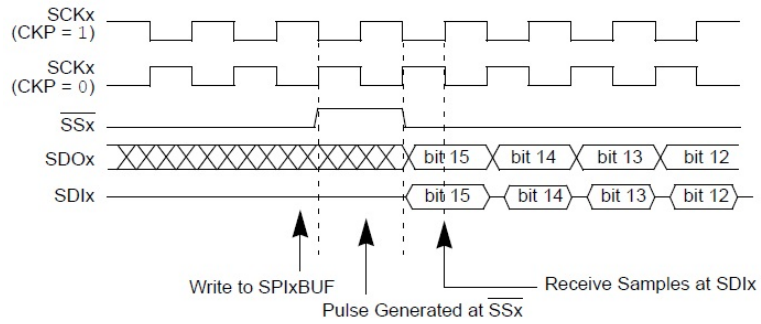


Figura 4.1: SPI Framed mode

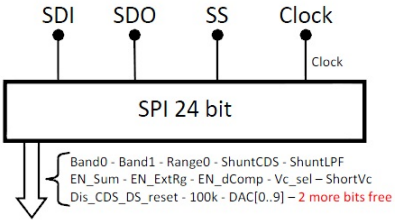
4.2 Dati di programmazione Berillio

La programmazione del sensore ibrido avviene per mezzo di un interfaccia grafica, la quale visualizza i dati acquisiti dal modulo USB del microcontrollore. Con riferimento alla (Fig. 4.3), cliccando sul tasto **connect**, si inizializza la comunicazione tra l'interfaccia e il device(USB-Key), scegliendo l'opportuna porta COM associata. Nella sezione bassa centrale risiede il pannello di controllo, nel quale è possibile definire:

- **range di corrente:** $\pm$ 200nA, $\pm$ 20 nA, external
- **bande di frequenza:** 625 Hz, 1250 Hz, 5000 Hz, 10000 Hz
- **Data storage option:** permette di registrare parte del flusso dati visualizzato sull'interfaccia grafica
- **Stimulus Vc:** E' la tensione di controllo del DAC interno a Berillio, la quale può assumere valori di un onda triangolare, costante o definita da esterno (Fig. 4.4)

- **Trial 1 (Vc):** Setta l'ampiezza dell'impulso del segnale di prova Vc , la durata e il ritardo t0
- **Trial 2 (Vc):** Definisce il numero di step in ampiezza sul segnale di prova, mediante la modiafica del paramentro *D*

Entrando in merito ai dettagli della comunicazione SPI instaurata tra il micro e Berillio, si faccia un confronto con la (Fig. 2.6) e la (Fig. 4.2), per notare subito una differenza sostanziale. La seriale del sensore è di 24 bit, la cui estensione nei pic32 non è implementata. Si utilizza, pertanto, un SPI a 32 bit, di modo che i primi 8 bit che entrano nel chip Berillio non rappresentino alcun dato informativo e siano posti a zero. Detto ciò, verranno memorizzati soltanto gli ultimi 24 bit a guadagnare l'ingresso. Inoltre, ciascun bit in ricezione dal pc, si trova fuori posto rispetto al flusso dati richiesto dal sensore. Per cui, occorre un riordino delle informazioni all'interno del microcontrollore prima di ritrasmetterle. Per testare il corretto posizionamento dei bit, si è fatto uso del monitor seriale, dal quale sono state inviati i 12 byte di configurazione. Successivamente, mediante oscilloscopio si è verificato il segnale SDO e SS che rispetti le specifiche richieste.



bit 23	bit 22	bit 21	bit 20	bit 19	bit 18	bit 17	bit 16
DAC9	DAC8	DAC7	DAC6	DAC5	DAC4	DAC3	DAC2
P0-3	P1-3	P2-3	P3-3	P4-3	P5-3	P6-3	P7-3

bit 15	bit 14	bit 13	bit 12	bit 11	bit 10	bit 9	bit 8
DAC1	DAC0	ShuntCDS	Free1	ShortVc	Vc_sel	EN_Sum	EN_dComp
P0-2	P1-2	P2-2	P3-2	P4-2	P5-2	P6-2	P7-2

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
EN_ExtRg	ShuntLPF	Free2	Dis_CDS_DS_reset	100k	Range0	Band1	Band0
P0-1	P1-1	P2-1	P3-1	P4-1	P5-1	P6-1	P7-1

Figura 4.2: Dati di programmazione Berillio

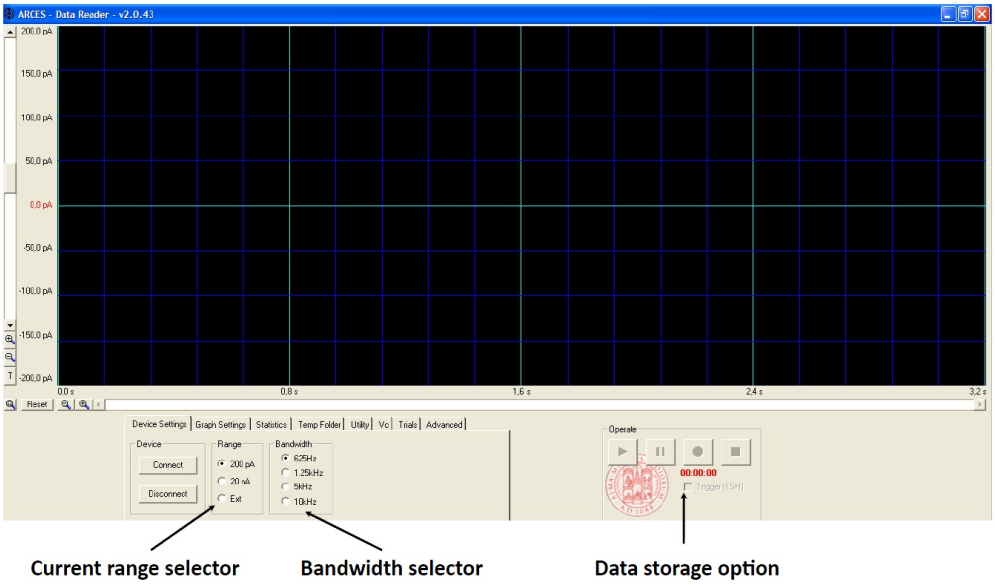


Figura 4.3: Interfaccia Software

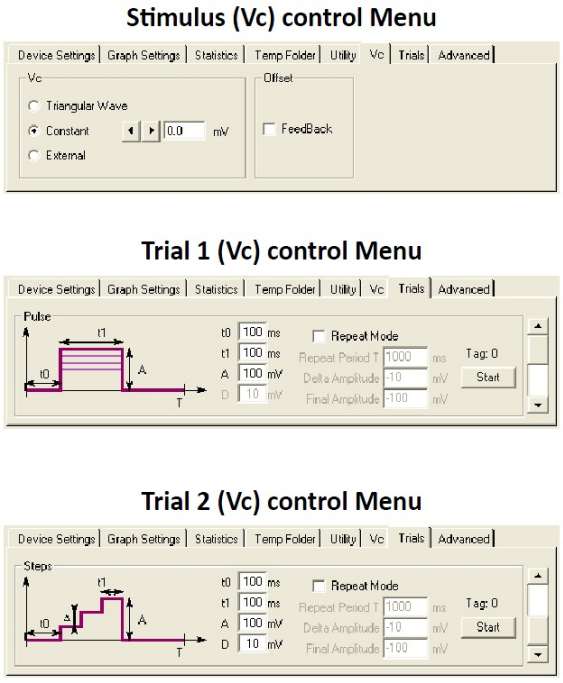


Figura 4.4: Menù di controllo

DAC[0..9]	Internal Vc generator. To apply 0V, set DAC[0..9] to 200hex
ShuntCDS	"1" → The input ant output pins of CDS are shorted ad the output pin is unplugged from the CDS. "0" → The CDS works properly.
ShortVc	"1" → Shorts the two outputs Vc1 and Vc2.
Vc_sel	"1" → The internally generated Vc signal is applied to the front-end. "0" → The external Vc is applied.
EN_Sum	"1" → Enables the analogic adder block (Sum).
EN_dComp	"1" → Enables the automatic compensation feedback.
EN_ExtRg	"1" → Enables the input pin ext_range.
ShuntLPF	"1" → The input ant output pins of LPF are shorted. "0" → The LPF works properly.
Dis_CDS_DS_reset	
100k	"1" → Enables the 100kΩ resistor inside the LPF block.
Range0	"1" → ±200 pA; "0" → ±20 nA;
Band1;Band0	00→625Hz; 01→1.5KHz; 10→5KHz; 11→10KHz bandwidth

Figura 4.5: Bit di programmazione Berillio

Capitolo 5

USB-Key

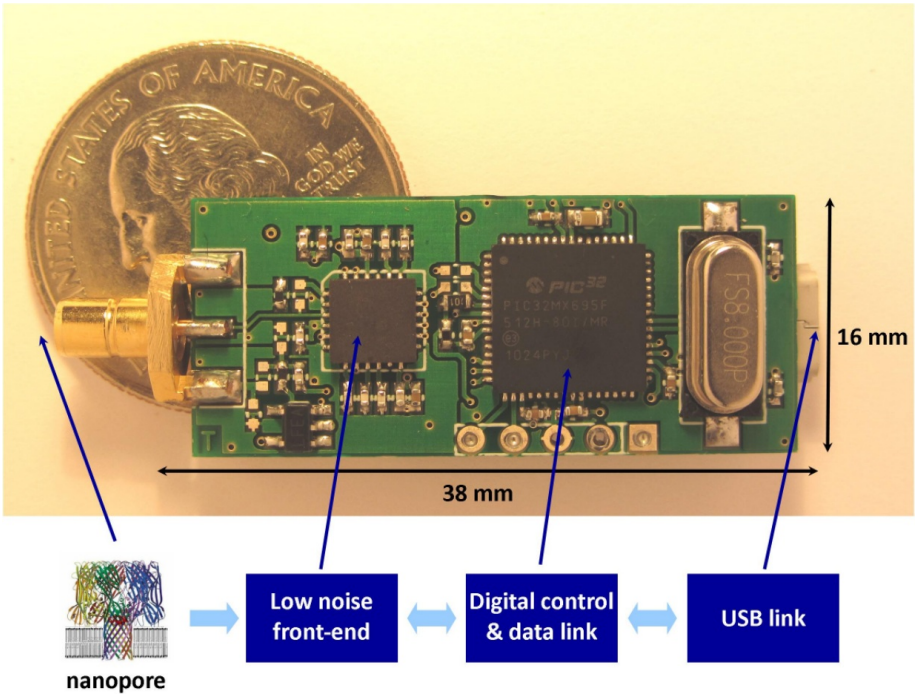


Figura 5.1: Amplificatore integrato Usb-Key

Mettendo insieme i vari concetti espressi fin ora, che hanno descritto l'idea progettuale dalla quale si è partiti, si è definito un sistema hardware realizzato appositamente per i nostri scopi, rappresentato dall' amplificatore per elettrofisiologia integrato di (Fig. 5.1). Procedendo da sinistra verso destra l'ASIC (*Application Specific Integrated Circuit*) è composto da una prima sezione adibita all'acquisizione della corrente ionica dei canali ionici, mediante il sensore ibrido Berillio, supportato da uno stadio di alimentazione a basso rumore. La seconda sezione dispone del PIC32MX695F512H, utilizzato per la realizzazione del filtraggio digitale sul segnale $\Delta\Sigma$ fornito dal sensore, con trasferimento dati al

calcolatore, mediante USB. In (Fig. 5.2) è riportato il risultato di un'acquisizione, gestita dall'interfaccia software, la quale è realizzata appositamente per la funzione grafica dei dati ricevuti dall'Usb-Key e programmazione Berillio. Inoltre, il microcontrollore, si occupa anche della ricezione e l'inoltro dei bit di programmazione del chip Berillio. Il chip è realizzato con tecnologia CMOS a 0.35μm e presenta della dimensioni veramente ridotte (38x16 mm) ed un peso di appena 6 grammi. Essa viene alimentata mediante la connessione USB, con un residuo consumo di corrente elettrica. Nonostante il prodotto ottenuto, abbia dimensioni molto ridotte, è possibile condurre in modo semplice la programmazione del microcontrollore. Essa viene eseguita mediante un programmatore esterno messo a disposizione della Microchip che prende il nome di **PICkit 3**. Il dispositivo, di dimensioni portatili, consente di portare a termine la programmazione con un connettore a 5 vie. Come si può constatare dalla (Fig. 5.3), i collegamenti vengono eseguiti in modo diretto con il dispositivo interessato senza la necessità di componentistica esterna, eccetto una resistenza di pull-up da 10 KOhm sul conduttore **MCLR**. In realtà può essere omessa tale resistenza, sfruttando la possibilità di impiegare quella interna al microcontrollore. Detto questo, è stato disposto sulla USB-Key, il solo connettore. La comunicazione che gestisce la programmazione, è di tipo seriale sincrono, bidirezionale i cui dati vengono trasmessi mediante il conduttore **PGD**, invece, il **PGC**, rappresenta la linea di clock che sincronizza lo scambio dati. Tuttavia il PICkit 3 ci offre anche la possibilità di condurre una fase di debug sul dispositivo, mediante gli stessi collegamenti sopra indicati. Infatti, in tal caso, si lascia connesso il programmatore al microcontrollore, dove il programma al suo interno è in esecuzione. Per cui, si possono andare a vedere, in real time, tutti i registri della cpu e le variabili dichiarate nel sistema software creato. Questa modalità è stata utile, durante le ultime fasi di test del filtraggio, perchè ha permesso di discernere se gli errori riscontrati, fossero relativi al filtro stesso, o a eventuali problemi legati al campionamento del segnale $\Delta\Sigma$ in ingresso, come discusso nel (Cap 4).

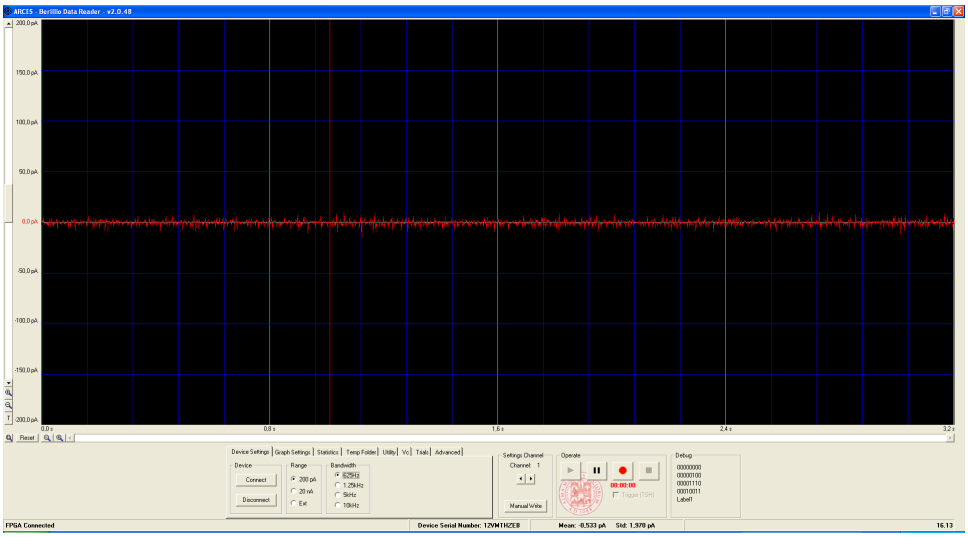


Figura 5.2: Visualizzazione di una generica Acquisizione dati

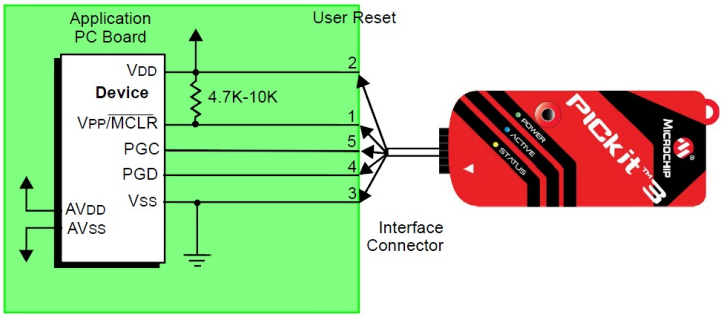


Figura 5.3: Connessione PICkit 3 per la programmazione dell'Usb-Key

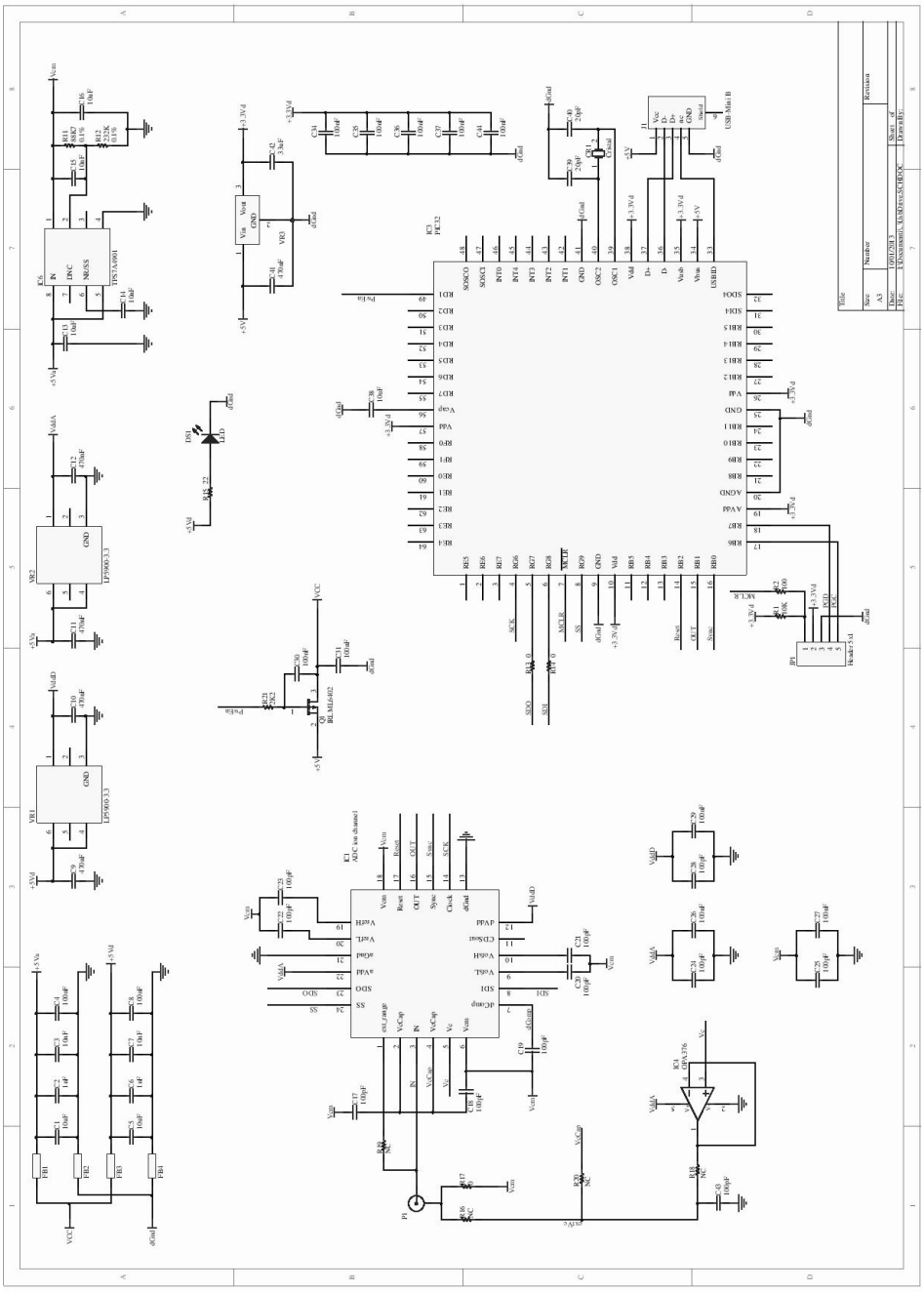


Figura 5.4: Schema elettrico USB-Key

Capitolo 6

Conclusioni

L'analisi progettuale condotta in questo elaborato, ci ha portato allo straordinario risultato, della realizzazione di un ASIC capace di unire qualità prestazionali e ridotte dimensioni fisiche a discapito dell' ingombrante e ormai datato Amplificatore Axon (Fig. 6.1). Non solo guadagniamo in termini di dimensioni, potendo inserire il sistema in una comune penna USB, ma anche di flessibilità, grazie alla programmazione gestita da un interfaccia grafica su Pc, mentre prima veniva definita in loco sullo strumento. I pochi componenti in uso, grazie anche alla proprietà *stand alone* del pic32, consente di abbattere i costi su produzione di larga scala. Inoltre,si possono mettere in conto alcuni cambiamenti e sviluppi futuri legati all'interfaccia a microcontrollore. Uno fra tutti, risiede nella possibilità di aumentare il numero di canali da elaborare cercando di ottimizzare il software. Se pensassimo, ad esempio, di gestire la comunicazione tra il microcontrollore e il PC, con l'ausilio del DMA(*Dinamic Memory Acces*), il quale si preoccupa lui stesso, in modo indipendente e parallelo, di trasferire le informazioni dalla memoria RAM al modulo USB, saremo in grado di sostituire le pesanti istruzioni della classe CDC, offerte dalla libreria Microchip. Infatti, esse richiedono un tempo di esecuzione, quando vengono richiamate, di ben oltre 250ns, circa 1/3 del tempo a disposizione tra un campionamento e l'altro del dato $\Delta\Sigma$ in ingresso al micro. Successivamente, sarà possibile eliminare dal dispositivo, il connettore dedicato alla programmazione mediante *PICkit 3*. La rimozione ci consente di guadagnare spazio ed è giustificata dalla possibilità di condurre una programmazione per via del connettore USB. Infatti, l'operazione è eseguibile mediante un *Bootloader* rilasciato dalla Microchip per quasi tutti i suoi dispositivi. Si tratta di un firmware da precaricare sul microcontrollore, il quale si prendere la responsabilità di capire se l'operatore vuole, mediante la connessione USB, interfacciarsi con il software sul Pc, messo a disposizione da Microchip, per il caricamento di un nuovo programma, oppure lanciare il software corrente già presente sul pic32. In conlusione, le specifiche progettuali richieste sono state soddisfatte, con la consapevolezza di poter mettere a disposizione un prodotto pronto all'uso ed efficiente.

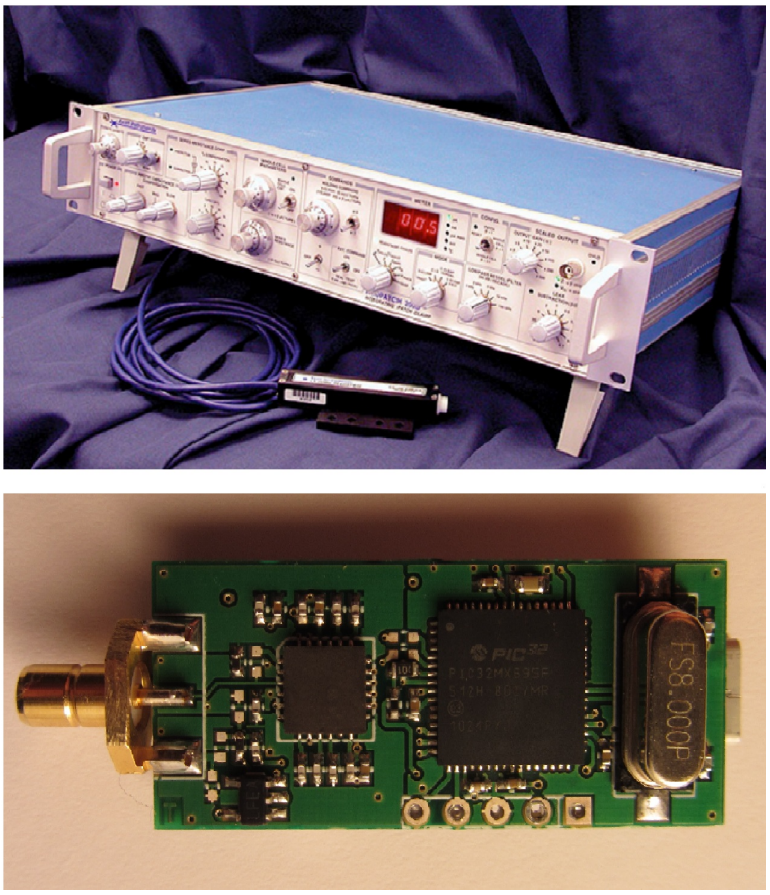


Figura 6.1: Confronto tra l'Amplificatore Axon (9x48x32 cm, peso: 5 Kg) e l'Usb-Key (38x16 mm peso: 6 grammi)

Elenco delle figure

Bibliografia

[1] ARCES, *Datasheet Berillio-SingleIonChannel-DS-ADC-V2*

[2] Microchip, *Pic32 Usb Starter Kit II User's Guide*, www.microchip.com

[3] Laurtec, *Tutorial - L'interfaccia SPI*, www.laurtec.com

[4] Lucio Di Jasio, *Programming 32-bit Microcontrollers in C: Exploring the PIC32*, 2008

[5] Microchip, *Usb CDC Class on an Embedded Device*, www.microchip.com

[6] Texas Instruments, Miroslav Oljaca e Tom Hendrick, *Combining the ADS1202 with an FPGA Digital Filter for Current Measurement in Motor Control Applications* , 2003

[7] Analog Device, *Datasheet of AD7401A : Isolated Sigma-Delta Modulator*

[8] Microchip, *PIC32 Peripheral Libraries for MPLAB C32 Compiler*, 2007, www.microchip.com

[9] Microchip, *PIC32MX5XX/6XX/7XX Family Datasheet*, 2009. www.microchip.com

[10] Microchip, *PIC32 Family Reference Manual, Section 6 Oscillators*, 2007. www.microchip.com

[11] Microchip, *PIC32 Family Reference Manual, Section 8 Interrupts*, 2007. www.microchip.com

[12] Microchip, *PIC32 Family Reference Manual, Section 14 Timers*, 2010. www.microchip.com

[13] Microchip, *PIC32 Family Reference Manual, Section 23 Serial Peripheral Interface*, 2007. www.microchip.com

[14] Microchip, *PIC32 Family Reference Manual, Section 27 USB OTG*, 2011, www.microchip.com