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 TELECOMUNICAZIONI

TITOLO DELLA TESI

REALIZZAZIONE DI UN SISTEMA DI MISURA IMPEDENZIOMETRICO MINIATURIZZATO AD ELEVATA RISOLUZIONE

Elaborato in
ELETTRONICA DEI SISTEMI DIGITALI

Relatore
Prof. Ing. ALDO ROMANI

Presentata da
LUCA MARTINI

Correlatore
Prof. Ing. FEDERICO THEI

II Appello III Sessione
Anno Accademico 2011-2012

“Far out in the uncharted backwaters of the unfashionable end of the western spiral arm of the Galaxy lies a small unregarded yellow sun.
Orbiting this at a distance of roughly ninety-two million miles is an utterly insignificant little blue-green planet whose ape-descended life forms are so amazingly primitive that they still think digital watches are a pretty neat idea.
This planet has - or rather had - a problem, which was this: most of the people on it were unhappy for pretty much of the time. Many solutions were suggested for this problem, but most of these were largely concerned with the movements of small green pieces of paper, which is odd because on the whole it wasn’t the small green pieces of paper that were unhappy.
And so the problem remained; lots of the people were mean, and most of them were miserable, even the ones with digital watches.”

Tratto da, “The Hitchhiker’s Guide to the Galaxy” (1979)
Douglas Adams(11 marzo 1952 -11 maggio 2001)

“*Tutte le tolleranze considerate in fase di progetto si allineeranno unidirezionalmente fino a raggiungere il massimo errore possibile”*

Legge di Klipstein applicata alla produzione di prototipi

Indice

Introduzione

L'attenzione alle tematiche ambientali oggigiorno risulta sempre più importante, una di queste senz'altro e quella riguardante il settore del monitoraggio delle acque in ambiente marino. L'importante progetto di ricerca sviluppato recentemente dal team di progettazione microelettronica dell'Università di Bologna presso i laboratori di Cesena in collaborazione con l'Università di Southampton (UK), ha riguardato la realizzazione di un interfaccia microelettronica per sensori destinati alla misurazione estremamente accurata di grandezze come temperatura, pressione, concentrazione di ossigeno disciolto e salinità delle acque in ambiente oceanico e finalizzata allo studio dei cambiamenti climatici.

Le caratteristiche fondamentali richieste all'interfaccia sono:

- consumo estremamente ridotto, dato che il suo utilizzo è previsto all'interno di boe da rilasciarsi nelle profondità del mare aperto, quindi con limitate risorse in termini energetici in quanto le boe saranno alimentate a batterie;
- elevata accuratezza, i parametri delle acque saranno misurati con estrema precisione, al fine di rilevare i minimi cambiamenti.

Lo scopo del presente lavoro di tesi è stato quindi quello di realizzare un sistema a microcontrollore che permetta ad un PC di interfacciarsi con il chip realizzato dai ricercatori dell'Università di Cesena, fisicamente questo integrato denominato “BORO” è in grado di leggere simultaneamente 4 canali che vengono caricati con altrettante impedenze, arrivando per ciascun canale ad una risoluzione di 16Bit *Fig.1*.

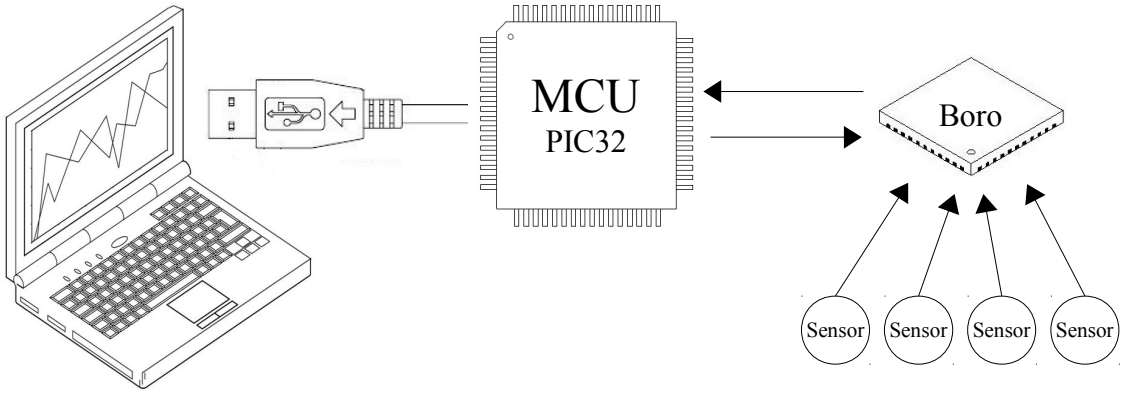


Fig.1 Sistema completo

L'obbiettivo finale, nonché scopo della tesi, è stato quindi quello di mettere in piedi un sistema che permetta la comunicazione in tempo reale tra l'interfaccia impedenziometrica ed un host interessato a compiere delle misurazioni attraverso essa. Si è fatto quindi uso di un microcontrollore opportunamente programmato in modo da permettere la comunicazione tra interfaccia ed host, in particolare per quanto riguarda la comunicazione da microcontrollore ad interfaccia il microcontrollore ha il compito di programmare l'interfaccia impedenziometrica secondo le specifiche richieste dal PC trasmettendole attraverso bus SPI, inoltre ha il compito di fornire un segnale PWM a modulazione sinusoidale in maniera continuativa.

Per quanto riguarda invece la comunicazione da interfaccia a microcontrollore, il compito del microcontrollore è quello di acquisire, filtrare e trasmettere al PC i dati provenienti dall'interfaccia.

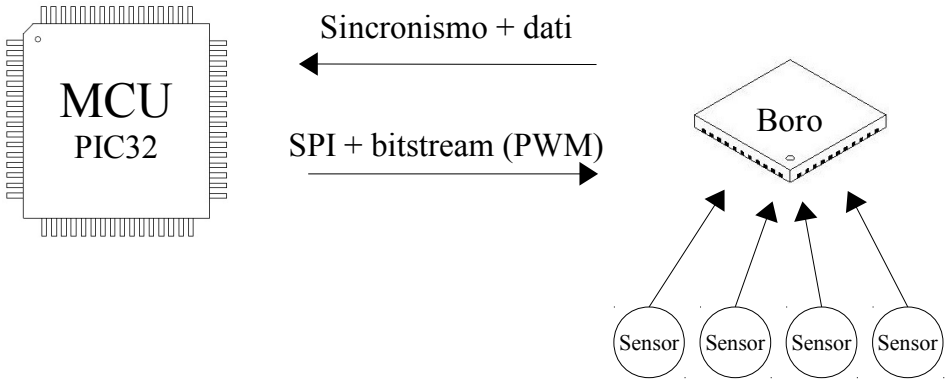


Fig.2 flussi di scambio dati

In particolare la comunicazione tra host e microcontrollore avviene attraverso un canale di tipo USB.

Nella seguente trattazione si parlerà distintamente di *interfaccia impedenziometrica* per quanto riguarda la parte analogica e relativa al dispositivo denominato BORO, mentre si parlerà di *sistema di lettura impedenziometrico* in riferimento alla totalità del sistema.

Schema a blocchi del sistema

Come è possibile notare osservando la *Fig.3* il sistema è composto principalmente dal microcontrollore e dal chip BORO opportunamente interfacciati per consentirne la comunicazione.

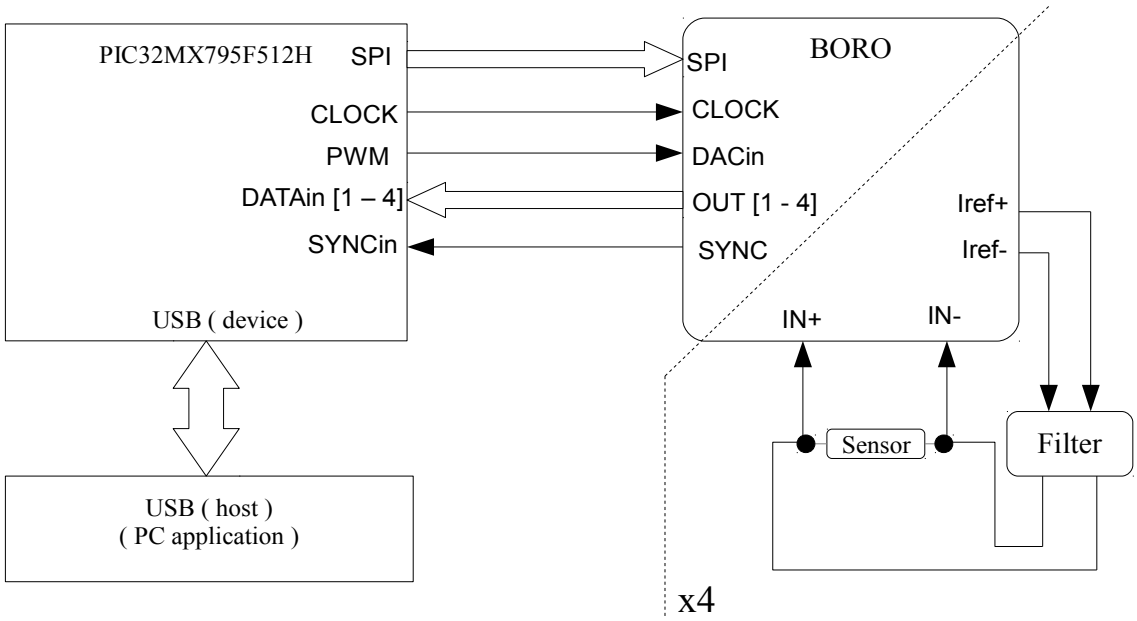


Fig. 3 Struttura hardware

L'interfaccia impedenziometrica, come verrà descritto più dettagliatamente nel corso dell'elaborato, grazie alla sua struttura interna composta da 4 core, ha la possibilità di eseguire parallelamente 4 letture distinte relative ad altrettanti ingressi distinti. Agli ingressi analogici del chip si presenteranno quindi le grandezze elettriche prodotte dai sensori, le quali in ultima analisi saranno gli oggetti della nostra misura.

Osservando la struttura hardware in *Fig.3* troviamo quindi 4 linee indipendenti *OUT[1 – 4]* in uscita dall'interfaccia e dirette verso il microcontrollore abbinate ad un segnale di sincronismo *SYNC* necessario per l'acquisizione dello stream di bit presente sulle 4 linee dati.

Anticipando quello che vedremo nel corso dell'elaborato quando studieremo nel dettaglio il segnale in uscita dall'interfaccia possiamo dire infatti che queste linee di uscita trasportano segnali appartenenti alla categoria delle modulazioni *sigma delta*, e necessitano quindi di un segnale di sincronismo per essere campionate.
Il controllo e l'inizializzazione dell'interfaccia impedenziometrica sono state previste dagli sviluppatori del chip attraverso la trasmissione di una parola di configurazione a 64bit su un bus di comunicazione di tipo SPI, dove il

microcontrollore riveste il ruolo di master.

Per quanto riguarda invece la misura impedenziometrica occorre ricordare che per definizione un impedenza è esprimibile come un numero complesso, composta cioè da una parte reale che rappresenta la pura *resistenza (R)*, e da una parte immaginaria che considera gli effetti di accumulo energetico chiamata *reattanza (X):*

$$\vec{Z} = R + jX \quad [ohm]$$

$$\vec{Z} = |Z| e^{j\theta}$$

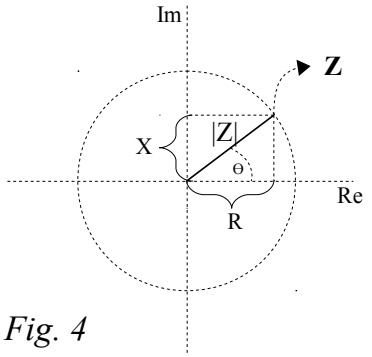


Fig. 4

Per garantire l'esatta lettura dell'impedenza, oltre alla corretta lettura del modulo è necessario porre particolare attenzione alla fase del vettore rappresentativo associato *Fig.4*, da qui nasce l'esigenza di fornire all'interfaccia un segnale periodico che garantisca con la propria fase un punto di riferimento stabile per la misura della parte immaginaria dell'impedenza. Questa esigenza viene servita da un segnale digitale di tipo PWM in uscita dal microcontrollore, che come vedremo nel dettaglio successivamente, produce attraverso l'evoluzione nel tempo del suo valore medio, l'approssimazione ad una funzione di tipo sinusoidale alla frequenza di 1KHz e avente fase costante rispetto al segnale di CLOCK fornito al chip BORO.

Un altra parte fondamentale della realizzazione riguarda la comunicazione verso PC, che è in ultima analisi il fruitore della misura effettuata dal sistema, questa avviene attraverso una connessione di tipo USB implementata facendo uso della libreria ad essa dedicata e fornita gratuitamente dal costruttore del microcontrollore, essa sarà configurata in modo da consentire l'emulazione di una porta di comunicazione seriale COM. L'host quindi una volta connesso il dispositivo e installati i relativi driver, vedrà tra le sue periferiche una nuova porta di tipo COM sulla quale, modellati dai rispettivi protocolli, verranno trasmesse da parte dell'host le parole di configurazione, mentre da parte del device le misure elaborate in tempo reale. A lato PC sarà quindi presente un software in ascolto sulla COM designata alla comunicazione, che si occuperà di produrre le "*config-word"* (parole di configurazione) necessarie al controllo e alla programmazione dell'interfaccia coerentemente con le caratteristiche della misura da effettuare. Successivamente il software lato PC dovrà essere pronto all'acquisizione e alla elaborazione dei dati prodotti dal sistema, nonché ad un

elaborazione grafica dei dati ricevuti.

Tuttavia la realizzazione del software lato PC esula dagli obbiettivi di questo elaborato e per i test è stato fornito un modello di prova scritto dai ricercatori che hanno progettato il dispositivo.

Funzionamento del software

La maggior parte del tempo impiegato nello sviluppo di questo lavoro di tesi è stata sicuramente dedicata alla progettazione del software necessario al microcontrollore per instaurare la comunicazione tra interfaccia impedenziometrica e PC. Le soluzioni adottate per le varie parti che compongono il codice verranno descritte nel dettaglio nel corso dell'elaborato, ma vediamo ora per punti quali sono i principali compiti affidati al microcontrollore:

- comunicare con un dispositivo *host* tramite interfaccia USB;
- ricevere e riconoscere le *config-word* ;
- programmare l'interfaccia coerentemente alle *config-word* ricevute attraverso il bus SPI;
- campionare i dati provenienti dai canali di uscita dell'interfaccia;
- elaborare i campioni in tempo reale secondo un algoritmo di filtraggio di tipo $Sinc^3$;
- formattare i dati secondo protocollo e trasmetterli all'*host*;
- produrre in maniera continuativa un particolare segnale di riferimento PWM;

L'esigenza principale nella progettazione del codice è stata quella di assolvere a tutti i compiti appena descritti nella maniera più veloce possibile non appena ne venga fatta richiesta. Si è fatto quindi uso di tecniche di interrupt, sfruttando diversi livelli di priorità e limitando la tecnica di polling alle sole funzioni il cui eventuale intervento tardivo non comprometta il funzionamento del sistema e non comporti in alcun modo perdita di informazione, ma nel peggiore dei casi solo un trascurabile ritardo nella disponibilità dei dati.

Il codice è stato scritto in linguaggio C utilizzando l'ambiente di lavoro fornito gratuitamente dal produttore denominato MPLAB IDE *(Integrated Development Environment)* e facendo uso del compilatore C32. La logica alla base della stesura del codice è rappresentata in *Fig.5*, in essa viene mostrata la struttura generale del codice e la gerarchia temporale che lega le funzioni sopra descritte.

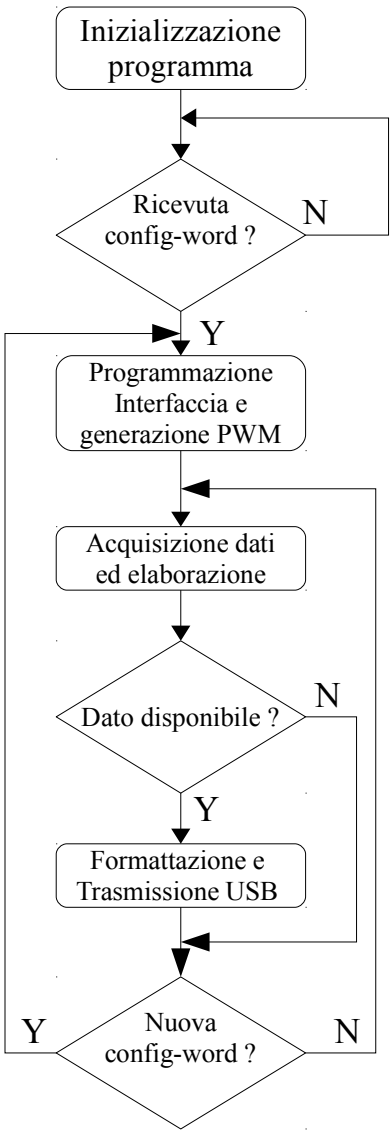


Fig.5 Diagramma a blocchi generale

Una volta alimentato il microcontrollore e stabilita la comunicazione con l'host il Software rimane in uno stato di sospensione (idle) nella quale attende dall'host una config-word valida, cioè coerente con il protocollo di programmazione che analizzeremo in seguito.

Quando viene rilevata una config-word valida il sistema memorizza ed elabora il suo contenuto, essa infatti contiene al proprio interno direttive di configurazione sia per il microcontrollore che per l'interfaccia impedenziometrica, il sistema si occupa quindi di abilitare il segnale di riferimento PWM e di trasmettere sul bus

SPI solo la parte dedicata all'interfaccia, rendendola in questo modo operativa e pronta a trasmettere i dati.

Sfruttando il segnale di sincronismo il microprocessore inizia a questo punto il processo di acquisizione dati e si occupa della loro elaborazione. Il processo di elaborazione prevede per i dati un filtraggio di tipo $Sinc^3$, questa tecnica, che verrà descritta in dettaglio nel relativo capitolo, fornisce un dato utile solo dopo aver collezionato un numero predefinito di campioni del segnale in ingresso, questo numero viene chiamato OSR (Over Sampling Ratio). Si può quindi capire la necessità di effettuare un controllo *(Fig.5)* sull'effettiva disponibilità del dato prodotto dal filtraggio, questo controllo regola quindi la trasmissione del dato all'host che avviene solo quando ve ne è la disponibilità. Il restante controllo che compare nel diagramma di flusso *(Fig.5)* permette infine di aggiornare in corso d'opera le impostazioni dell'interfaccia e del microcontrollore qualora l'host ne faccia richiesta inviando una nuova *config-word* , in tal caso infatti viene richiamata quella funzione adibita alla trattazione della *config-word* e si ripercorre il processo di programmazione dell'interfaccia impedenziometrica.

Il microcontrollore

Generalità

In elettronica digitale il microcontrollore o MCU *(Micro Controller Unit)* è un sistema programmabile miniaturizzato che include in un singolo chip un sistema digitale più o meno completo adatto ad applicazioni di tipo “stand alone”.

All'interno di un MCU possiamo trovare infatti:

- unità di elaborazione: CPU
- oscillatori per la generazione del clock
- memorie RAM, EEPROM per i dati
- memorie EEPROM o FLASH per la memorizzazione del programma
- periferiche di I/O digitali ed analogiche
- gestione Interrupt
- interfacce di comunicazione come SPI , I²C, USART, USB, CAN, ETHERNET..etc
- timer e contatori

ed altri componenti ancora che possono variare in base al modello di MCU.
Questi dispositivi grazie alla loro versatilità e completezza possono essere visti come dei computer miniaturizzati, e grazie alle interfacce di I/O di cui dispongono possono interagire direttamente con il mondo esterno tramite il programma residente nella propria memoria interna.
Sul mercato sono disponibili 3 fasce di MCU definite sulla base dell'ampiezza del bus dati: 8 bit, 16bit e 32 bit.
Esistono diverse aziende produttrici di microcontrollori, per questo sistema è stato scelto di affidarsi alla *Microchip* con la sua famiglia di MCU denominata *PICmicro*, scelta non vincolante ma dettata dalla grande disponibilità di modelli da essa prodotta, che permette quindi di adattarsi nel miglior modo alle esigenze del progettista.

Scelta del Microcontrollore

In questo progetto si è scelto di lavorare su un architettura a 32 bit per soddisfare le necessità di computazione legate al filtraggio di tipo Sinc³, in questo tipo di filtraggio infatti sono previsti registri di accumulo legati all'integrazione del segnale in ingresso che possono arrivare a contenere valori non rappresentabili con soli 16Bit. Inoltre si è scelto di lavorare su un dispositivo in cui una volta completate le funzioni base rimanessero disponibili risorse per prevedere una futura espansione o revisione del codice.

Il microcontrollore individuato per l'applicazione è il PIC32MX795F512H alloggiato su un PCB appositamente assemblato per i test *(Fig.6)*.

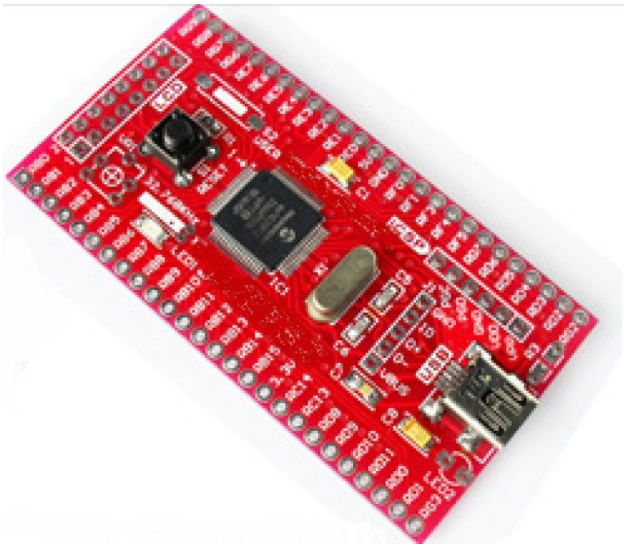


Fig.6 scheda di sviluppo usata per i test

La scheda oltre al microcontrollore monta una serie di componenti di contorno necessari al funzionamento ed alcune interfacce utili in fase di debug, troviamo infatti:

- quarzo da 8MHz
- regolatore di tensione a 3,3V
- condensatori di filtro per le alimentazioni
- pulsanti di reset e di test
- led indicatori di alimentazione e led di test
- connettore tipo mini USB
- connettore per la programmazione ICSP

Verrà presentato in seguito anche lo schema elettrico generale.

Interrupt

Generalità

Gli interrupt o *"interruzioni"* in generale sono eventi di natura asincrona che manifestandosi indicano il bisogno di attenzione da parte di una particolare periferica (interrupt hardware) oppure di una funzione di programma (interrupt software), l'avvento di un interrupt costringe il programma a memorizzare il suo stato di esecuzione e ad iniziare una subroutine o *"sottoprogramma"* che esegue le funzioni richieste dallo specifico tipo di interrupt manifestatosi. Terminata l'esecuzione della subroutine dedicata il processore riprende dalla parte di programma che aveva precedentemente salvato.
La tecnica di utilizzare questa risorsa per la gestione di particolari funzioni è detta ad interrupt, e consente di ottimizzare la gestione delle risorse del microprocessore da parte del codice, a differenza infatti della tecnica di polling che prevede il controllo insistente e ciclico del pin o del flag di attenzione, questa politica consente di dedicare le risorse computative all'evento solo quando ve ne sia effettiva richiesta, lasciando per il resto del tempo la CPU libera di assolvere altre mansioni.

Interrupt Vector Table (IVT)

Nei dispositivi della famiglia 32bit l'architettura prevede un ricco sistema di gestione degli interrupt, sono infatti disponibili 64 distinti tipi di interrupt che possono essere invocati. Ogni interrupt dispone di una propria routine di servizio chiamata Interrupt Service Routine (ISR).

Rispondere nel minore tempo possibile ad una richiesta di interrupt è una condizione assolutamente necessaria per il corretto funzionamento del sistema. L'obbiettivo essenziale infatti è quello di assegnare ad ogni ISR la più veloce routine possibile, per minimizzare il tempo di esecuzione ad esso associato, durante lo svolgimento di una ISR infatti il sistema risulta insensibile ad altre richieste di intervento dello stesso tipo di quella che l'ha generato, portando come intuibile ad una mancanza di intervento e quindi nel peggiore dei casi ad un malfunzionamento del sistema.

Questo massimo tempo di esecuzione viene definito come la stima del tempo che possa intercorre tra due richieste di interrupt dello stesso tipo.
Il compilatore C32 di cui si è fatto uso in questo lavoro ci viene in aiuto con una serie di funzioni adatte alla complessa gestione delle funzioni di interrupt.
Come abbiamo detto ad ogni tipo di interrupt corrisponde una subroutine da eseguire, per poter gestire questa collezione di subroutine l'architettura a 32bit

prevede l'indicizzazione di queste procedure attraverso la memorizzazione dell'indirizzo iniziale di ogni una di esse, questi indirizzi di memoria sono inseriti in una tabella chiamata Interrupt Vector Table (IVT) che è posizionata ad una specifica porzione di memoria.

In funzione della posizione occupata nella IVT ad ogni vettore è associato un numero progressivo che lo identifica *(Fig.7)*, questo numero viene chiamato Interrupt Request (IRQ), questo numero oltre a fornire una indicazione precisa del vettore a cui l'interrupt fa riferimento, instaura anche una gerarchia naturale delle richieste di interrupt, a parità infatti di priorità se due interrupt vengono generati simultaneamente quello ad essere servito prima è quello a cui corrisponde un IRQ inferiore.

PIC32MX5XX/6XX/7XX

TABLE 7-1: INTERRUPT IRQ, VECTOR AND BIT LOCATION

Interrupt Source(1)	IRQ	Vector Number	Interrupt Bit Location			
			Flag	Enable	Priority	Sub-Priority
Highest Natural Order Priority						
CT – Core Timer Interrupt	0	0	IFS0<0>	IEC0<0>	IPC0<4:2>	IPC0<1:0>
CS0 – Core Software Interrupt 0	1	1	IFS0<1>	IEC0<1>	IPC0<12:10>	IPC0<9:8>
CS1 – Core Software Interrupt 1	2	2	IFS0<2>	IEC0<2>	IPC0<20:18>	IPC0<17:16>
INT0 – External Interrupt 0	3	3	IFS0<3>	IEC0<3>	IPC0<28:26>	IPC0<25:24>
T1 – Timer1	4	4	IFS0<4>	IEC0<4>	IPC1<4:2>	IPC1<1:0>
IC1 – Input Capture 1	5	5	IFS0<5>	IEC0<5>	IPC1<12:10>	IPC1<9:8>
OC1 – Output Compare 1	6	6	IFS0<6>	IEC0<6>	IPC1<20:18>	IPC1<17:16>
INT1 – External Interrupt 1	7	7	IFS0<7>	IEC0<7>	IPC1<28:26>	IPC1<25:24>
T2 – Timer2	8	8	IFS0<8>	IEC0<8>	IPC2<4:2>	IPC2<1:0>
IC2 – Input Capture 2	9	9	IFS0<9>	IEC0<9>	IPC2<12:10>	IPC2<9:8>
OC2 – Output Compare 2	10	10	IFS0<10>	IEC0<10>	IPC2<20:18>	IPC2<17:16>
INT2 – External Interrupt 2	11	11	IFS0<11>	IEC0<11>	IPC2<28:26>	IPC2<25:24>
T3 – Timer3	12	12	IFS0<12>	IEC0<12>	IPC3<4:2>	IPC3<1:0>
IC3 – Input Capture 3	13	13	IFS0<13>	IEC0<13>	IPC3<12:10>	IPC3<9:8>
OC3 – Output Compare 3	14	14	IFS0<14>	IEC0<14>	IPC3<20:18>	IPC3<17:16>

Fig.7 assegnazione degli IRQ

Priorità

Per quanto riguarda la gestione di questi interrupt vi è anche la possibilità da parte del programmatore di poter assegnare agli interrupt utilizzati 7 livelli di priorità.
Questa assegnazione, coerentemente con le nostre esigenze di programmazione, dà al processore un ulteriore indicazione di quale sia l'ordine di priorità da seguire qualora venissero invocate più richieste di interrupt simultaneamente. L'assegnazione delle priorità alle routine di interrupt ci consente inoltre di sfruttare il sistema di annidamento, *(nesting mode)* che consiste nella possibilità

di servire richieste di interrupt anche durante lo svolgimento di una ISR relativa ad un interrupt a priorità inferiore.

La priorità viene definita in fase di programmazione attraverso l'assegnazione di un numero compreso tra 1 e 7 alla routine di interrupt, questa assegnazione verrà considerata in fase di compilazione andando ad associare ad ogni ISR la rispettiva priorità.

Questa tecnica di annidamento è stata sfruttata in questo lavoro per garantire la continua acquisizione del dato in uscita dall'interfaccia impedenziometrica, come vedremo in seguito sia la struttura di filtraggio dei dati sia l'algoritmo di generazione del segnale PWM di riferimento vengono gestiti ad interrupt, durante il funzionamento quindi ci troviamo difronte ad una notevole richiesta di intervento, tuttavia la priorità deve essere garantita alla routine di filtraggio per assicurare che non vengano persi campioni, si è scelto di assegnare quindi un numero di priorità superiore alla routine di filtraggio.

In ultima analisi possiamo dire che il livello di priorità assegnato dal programmatore interviene prima del livello di priorità naturale.

Programmazione dell'interfaccia impedenziometrica

Come descritto nel paragrafo introduttivo l'interfaccia impedenziometrica viene programmata dal Software lato PC, una volta selezionati i parametri funzionali attraverso la finestra principale *Fig.8*, il Software invia tramite la seriale emulata (USB) la *config-word* che racchiude al proprio interno le informazioni necessarie alla programmazione dell'interfaccia, queste informazioni sono tuttavia da elaborare e formattare correttamente per la successiva trasmissione attraverso il modulo SPI.

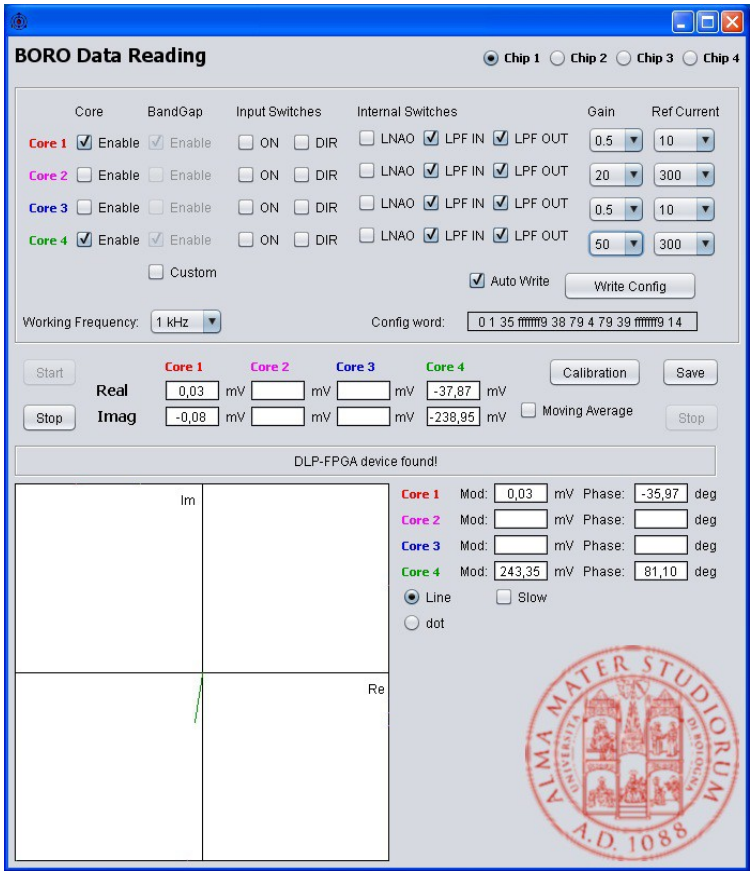


Fig.8 Software lato PC

Analizzeremo ora l'interfaccia SPI e il processo che porta alla programmazione dell'interfaccia impedenziometrica in funzione della *config-word* prodotta dal software lato PC.

SPI

Introduzione

Come precedentemente dichiarato la programmazione dell'interfaccia impedenziometrica avviene attraverso bus SPI con la trasmissione di una parola di configurazione a 64bit, a tale scopo verrà mostrato in questo paragrafo il funzionamento dell'interfaccia SPI presente all'interno del MCU, i parametri per la sua configurazione e le modalità di funzionamento.

Teoria

L' SPI o *Serial Peripherial Interface* proposta da Motorola è un sistema di comunicazione seriale tra più dispositivi, la trasmissione avviene tra un dispositivo detto *master* e uno o più dispositivi detti *slave*. Il master controlla il Bus, si occupa di generare il segnale di clock e decide quando iniziare e terminare la comunicazione.

Per come è definito il protocollo, la comunicazione SPI prevede l'uso di due linee fisicamente separate, una dedicata al flusso di dati in uscita dal master e l'altra dedicata al flusso in ingresso al master. Essendoci quindi la possibilità di trasmettere e ricevere dati in uno stesso intervallo di tempo tale comunicazione è a tutti gli effetti definita come *Full-duplex.*

In questo lavoro di tesi si fa uso di un solo dispositivo slave, ovvero l'interfaccia impedenziometrica, la comunicazione è definita quindi esclusivamente tra il master (MCU) e lo slave (interfaccia impedenziometrica).

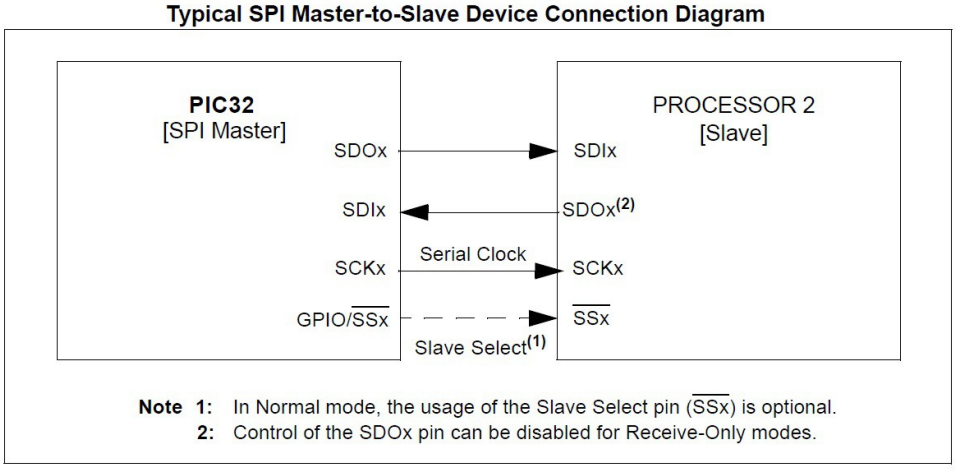


Fig.9 Bus SPI

Come possiamo vedere in *Fig.9* si riconoscono 4 linee distinte:

- **SDO** Serial Data Out, denominato anche MOSI (Master Output Slave Input)
- **SDI** Serial Data In, denominato anche MISO (Master Input Slave Output)
- **SCK** Serial Clock, denominato anche SCLK
- **SS** Slave Select, denominato anche CS (Chip Select)

Il master è incaricato di avviare la trasmissione e la presenza del segnale di SS serve a garantire l'univocità della trasmissione, infatti nel funzionamento più generale il master può comunicare con più slave, e questo segnale abilita alla ricezione solo il destinatario della comunicazione. Analizzeremo ora i vari passaggi che caratterizzano la trasmissione SPI implementata in questo lavoro di tesi.

Per prima cosa occorre precisare che ci sono diverse modalità di funzionamento per questo modulo, e per questo lavoro di tesi la modalità di funzionamento scelta e definita come *Framed Mode*, e consiste nel mantenimento continuo del clock da parte del master e dalla attivazione dello SS solo per la durata della trama.

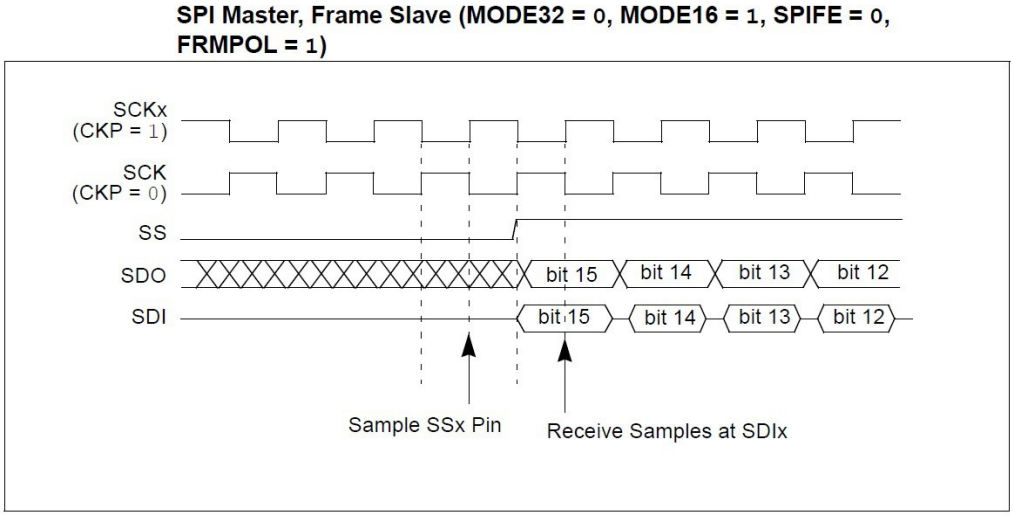


Fig.10 Diagramma temporale

Come si nota nel diagramma temporale dei segnali in *Fig.10* una volta definita la frequenza e la polarità del clock che vogliamo utilizzare, il master si occupa di mantenerlo attivo sul pin SCK in maniera indefinita, la scelta di usare questa modalità di trasmissione è dettata infatti dalla necessità di fornire in maniera continuativa il clock all'interfaccia impedenziometrica, che sfrutta questo segnale per il proprio funzionamento.

Quando inizia una trasmissione il master si occupa di portare ad 1 il valore

logico della linea SS mettendo lo slave in condizione di ascolto, questo in funzione del fatto che l'interfaccia impedenziometrica al contrario della modalità di funzionamento più comune, ha il controllo di selezione di tipo attivo-alto. A questo punto lo slave campiona ad ogni fronte di salita del clock il valore della linea SDO ed è proprio in questo istante che inizia il vero e proprio trasferimento dei dati, ad ogni ciclo di clock infatti il MSB all'interno dello shift-register del master viene trasmesso allo slave attraverso la linea SDO, il bit trasmesso entra nello shift-register dello slave in posizione LSB, il quale analogamente trasmette al master il proprio MSB, questo processo di scorrimento che si conclude con il ritorno a livello basso dello SS, persiste ad ogni ciclo di clock fino a quando non è stata trasmessa tutta la trama e i contenuti degli shift-register non si sono scambiati completamente. Come è stato già enunciato, per la programmazione dell'interfaccia impedenziometrica occorre trasferire attraverso il bus SPI una parola di 64bit, dal momento che l'interfacci SPI presente nel MCU consente trasferimenti in modalità 8bit 16bit e 32bit, si è scelto di susseguire 4 trasmissioni da 16bit.

Interfaccia SPI nella serie PIC32

Il PIC32MX795F512H possiede al proprio interno 3 interfacce SPI indipendenti e tutte di identico funzionamento a livello concettuale.

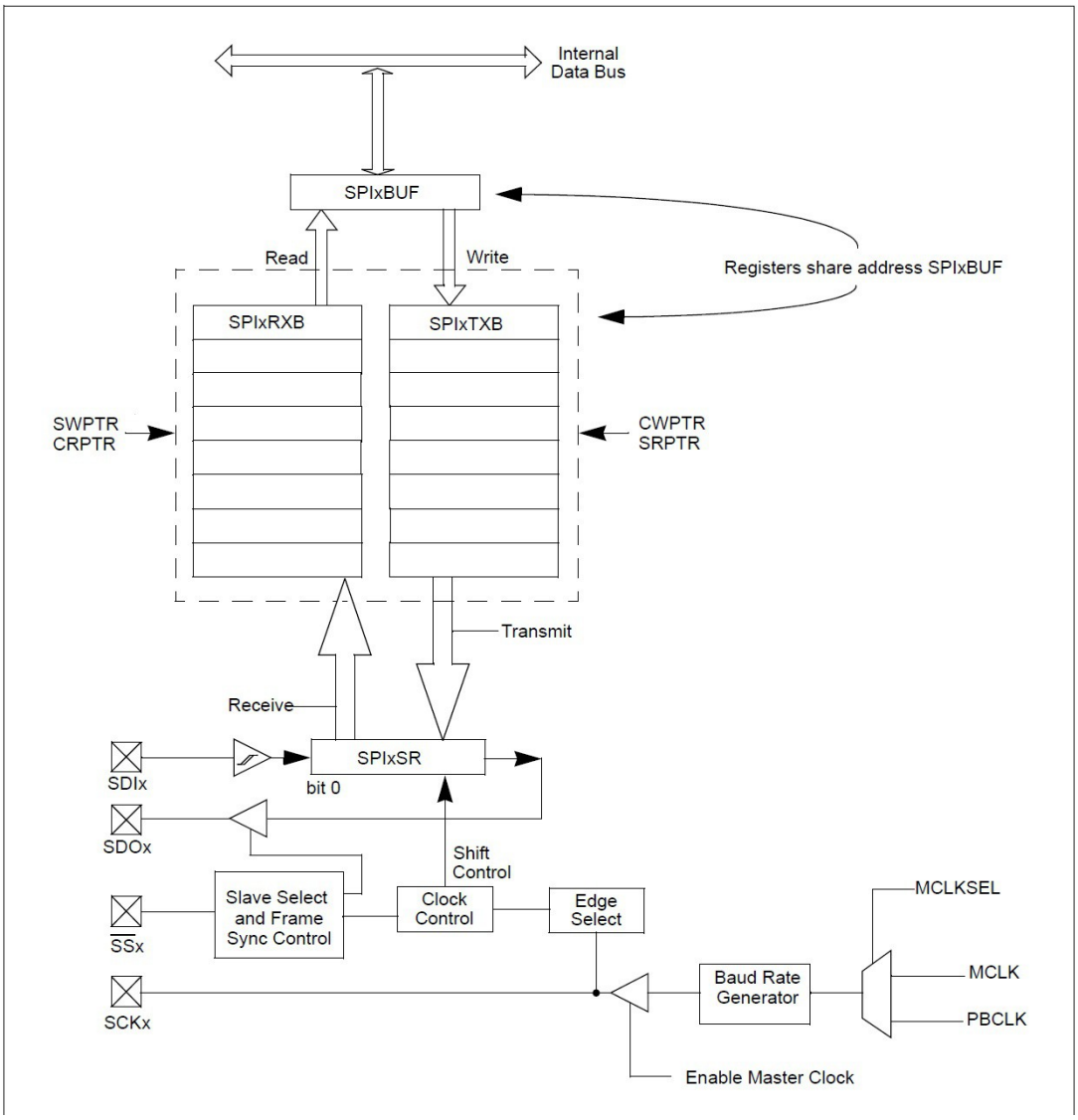


Fig.11 Schema a blocchi del modulo SPI nella serie PIC32

Ogni modulo prevede due modalità operative distinte:

- **Standard mode**: in questa modalità il dato passa attraverso un singolo registro, sia in trasmissione che in ricezione, ovvero i registri SPIxTXB e SPIxRXB
- **Enhanced mode**: in questa modalità il dato passa attraverso un registro FIFO ad 8 livelli nel caso di trasmissione a 16Bit.

I principali registri del modulo SPI *(Fig.11)*, sono:

- **SPIxBUF**: questo è il registro attraverso il quale la CPU scrive i dati da trasmettere e legge quelli ricevuti, questo registro si affaccia al bus dati.
- **SPIxCON** e **SPIxCON2**: sono due registri di configurazione attraverso i quali si definiscono tutte le caratteristiche della trasmissione.
- **SPIxSTAT** è il registro di stato della periferica, essenziale per conoscere lo stato della trasmissione, attraverso di esso è possibile conoscere le condizioni operative, consente di abilitare il modulo SPI e di decidere che tipo di interruzione generare.
- **SPIxSR** è il registro a scorrimento *(Shift Register)* attraverso il quale avviene lo scambio di dati.

Le caratteristiche di trasmissione come la frequenza di clock, la modalità di trasmissione e la lunghezza della trama di questo progetto sono univocamente imposte dalle necessità di funzionamento dell'interfaccia impedenziometrica, essa infatti per funzionare utilizza come clock di sistema il segnale SCK proveniente dal interfaccia SPI del MCU, per questa ragione i vincoli sulle specifiche dell'SPI sono:

- Presenza continua del segnale di clock alla frequenza di 500KHz.
- Segnale di selezione SS attivo alto e persistente per tutta la durata della trama.
- Lunghezza totale della trama 64bit.

Per configurare quindi il master secondo le specifiche imposte dallo slave occorre agire sui registri di configurazione **SPIxCON** e **SPIxCON2**, tuttavia per velocizzare la scrittura del codice non si è agito direttamente sui bit dei registri ma si è fatto uso della libreria fornita da Microchip, la quale attraverso la funzione di inizializzazione del modulo SPI permette di definire sintatticamente la modalità di funzionamento.

Per soddisfare la necessità riguardante la frequenza clock occorre determinare il fattore di divisione che scala opportunamente il clock di sistema ad un valore di 500Khz:

$$F_{SCK} = \frac{F_{SYSTEM\ CLOCK}}{ClkDiv} \rightarrow ClkDiv = \frac{60 * 10^6}{500 * 10^3} = 120$$

questo valore *(ClkDiv)* verrà utilizzato durante l'inizializzazione del modulo SPI. Per definire invece la modalità di trasmissione si fa uso delle opzioni offerte dalla libreria che andranno ad agire direttamente sui registri di configurazione, premettendo che in questo progetto si è fatto uso del terzo canale SPI per questioni di comodità inerenti pinout della scheda di sviluppo, mostriamo ora la procedura di inizializzazione dell'SPI presente nel codice:

```
//**********************SPI INIT*****************************
//chiusura canale SPI
SpiChnEnable(SPI_CHANNEL3, 0);  //disabilito l'SPI per inizializzarlo

//inizializzazione SPI
SpiChnConfigure(SPI_CHANNEL3,  //configurazione del modulo SPI3
SPI_CONFIG_CKP_HIGH|           //campionare sul fronte di salita
SPI_CONFIG_MSTEN|              //MCU come master
SPI_CONFIG_ON|                 //abilita modulo SPI
SPI_CONFIG_MODE16);            //modalità di trasmissione a 16 bit

SpiChnOpen(3,
SPI_OPEN_MODE16|               //trama a 16 bit
SPI_OPEN_FRMEN|                //modo a trama
SPI_OPEN_CKP_HIGH|             //campionare sul fronte di salita
SPI_OPEN_MSTEN                 //MCU come master
,120);                         //divisore Clock di sistema 60MHz/500kHz=120
```

Tuttavia la libreria fornita da Microchip non contempla tra le proprie opzioni tutte le possibili configurazioni che questa periferica può assumere, ma solo una collezione di quelle più comuni, infatti la modalità *Framed,* l'unica per la quale il segnale SCK rimane sempre attivo, non gestisce il segnale SS in modo coerente con le esigenze del nostro progetto, in particolare in questa modalità il segnale SS risulta invertito (attivo-basso) e viene mantenuto attivo solo per il primo ciclo di clock che precede la trasmissione, mentre servirebbe attivo per tutta la durata della trasmissione e sincronizzato sul primo bit utile come mostrato in *Fig.10*.
Per questo è stato necessario agire direttamente sul registro di configurazione **SPI3CON** ed impostare a livello alto i bit:

- bit29 FRMPOL, se posto ad 1 rende lo SS attivo alto, al contrario rimane attivo basso.
- bit27 FRMSYPW (Frame Sync Pulse Width Bit) , mantiene attivo lo SS per tutta la durata.
- bit17 SPIFE (SPI Frame Edge), fa coincidere il fronte di salita dello SS con il primo bit utile, che che alternativa sarebbe anticipato di un ciclo di clock.

Per fare ciò si è creata una maschera ad hoc che modifica il registro SPI3CON attraverso un operazione logica *or* , che mette a livello 1 i bit desiderati e lascia al valore originale gli altri.

```
//bit27 attivo la sincronizzazione dello SS con il primo bit
//bit29 SS attivo-alto
//bit17 durata dello SS per tutta la trasmissione
SPI3CON=(SPI3CON|0b00101000000000100000000000000000);
//attivo il canale SPI3
SpiChnEnable(SPI_CHANNEL3, 1);
```

Ora la periferica SPI è configurata correttamente, ed il master è pronto a leggere a scrivere l'interfaccia impedenziometrica, tuttavia in questo lavoro da parte del master avviene solo un processo di scrittura in quanto lo lo slave non produce

informazione utile sul canale SPI, per questo motivo nel codice l'unica funzione di libreria che è stata utilizzata è:

- **void SpiChnWriteC (SpiChannel chn, unsigned int data)**: Prende in ingresso il canale su cui trasmettere il dato e il dato da trasmettere e lo invia tramite SPI.

Programmazione dell'interfaccia

Per fare questo come prima cosa occorre analizzare il protocollo di comunicazione con il quale il Software formatta la *config-word* trasmessa al MCU.
Come è possibile vedere in *Fig.12* si tratta di una stringa di 11 byte composta da 1 primo byte di sincronismo (0x00) e da 10 byte contenenti le specifiche per il MCU e per l'interfaccia impedenziometrica.

		bit 7	**bit 6**	**bit 5**	**bit 4**	**bit 3**	**bit 2**	**bit 1**	**bit 0**
Byte 1	Sync	0	0	0	0	0	0	0	0
Byte 2	CS+ Freq	0	F2	F1	F0	Chip4	Chip3	Chip2	Chip1
Byte 3	Core4 A	Input Sw Dir	Input Sw En	LNA2 Gain	LNA1 Gain	LNA0 Gain	Iref1	Iref0	Core En
Byte 4	Core4 B	Band Gap En	-	-	-	-	LNA Out Sw	LPF In Sw	LPF Out Sw
Byte 5	Core3 A	Input Sw Dir	Input Sw En	LNA2 Gain	LNA1 Gain	LNA0 Gain	Iref1	Iref0	Core En
Byte 6	Core3 B	Band Gap En	-	-	-	-	LNA Out Sw	LPF In Sw	LPF Out Sw
Byte 7	Core2 A	Input Sw Dir	Input Sw En	LNA2 Gain	LNA1 Gain	LNA0 Gain	Iref1	Iref0	Core En
Byte 8	Core2 B	Band Gap En	-	-	-	-	LNA Out Sw	LPF In Sw	LPF Out Sw
Byte 9	Core1 A	Input Sw Dir	Input Sw En	LNA2 Gain	LNA1 Gain	LNA0 Gain	Iref1	Iref0	Core En
Byte 10	Core1 B	Band Gap En	-	-	-	-	LNA Out Sw	LPF In Sw	LPF Out Sw
Byte 11	Iref Cal.	Iref 7	Iref 6	Iref 5	Iref 4	Iref 3	Iref 2	Iref 1	Iref 0

Fig.12 protocollo Software →MCU

Per comprendere il processo di elaborazione della config-word effettuato dal MCU occorre confrontare il protocollo in *Fig.12* con il protocollo di programmazione dell'interfaccia impedenziometrica visibile in *Fig.13*.

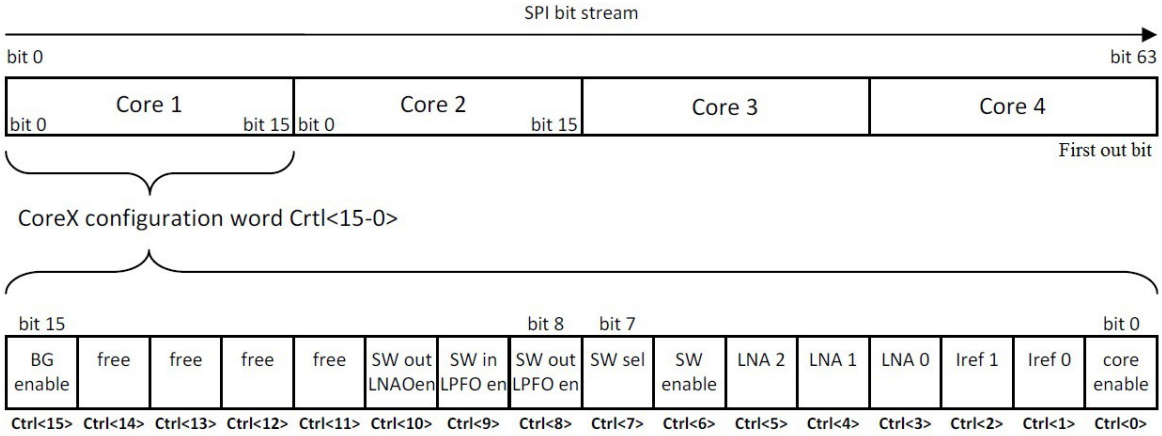


Fig.13 SPI stream interfaccia impedenziometrica

Possiamo notare che il primo byte ad essere trasmesso sul bus SPI è il byte 4 ovvero il "Core4 B" seguito dal "Core4 A" e così a seguire 3B, 3A, 2B, 2A, 1B, e 1A.
Nasce quindi l'esigenza di modificare la sequenza di byte con la quale è stata ricevuta la config-word dal PC come mostrato in *Fig.14*. Occorre tuttavia precisare che per esigenze di tempo in questo lavoro di tesi non sono state implementate tutte le opzioni di controllo possibili sull'interfaccia impedenziometrica, per questo motivo oltre al primo byte (0x00) nel processo di elaborazione vengono scartati anche il byte 2 ed il byte 11.
Per completezza occorre dire che se sfruttati questi byte danno la possibilità di:

- controllare fino a 4 interfacce impedenziometriche;
- selezionare la frequenza del segnale di riferimento a 1, 2, 4, 8 o16 Khz;
- calibrare la fase iniziale del segnale di riferimento;

in questo lavoro pertanto si è scelto di controllare una sola interfaccia impedenziometrica e con un unica frequenza di riferimento a fase iniziale nulla ed a frequenza 1Khz.

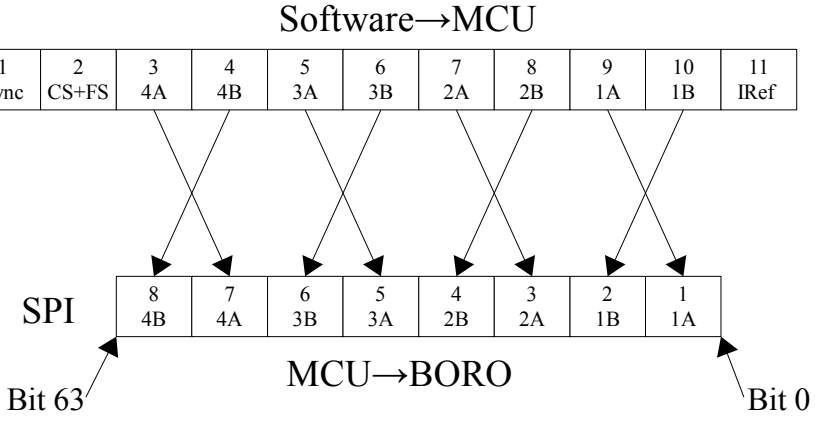


Fig.14 modifica della sequenza Byte

Analizziamo ora la sequenza cronologica con la quale opera il MCU in attesa di una config-word :

1. Il MCU controlla se sono stati ricevuti byte, altrimenti prosegue con il programma principale;
2. Se sono stati ricevuti byte controlla che siano 11 altrimenti li scarta e prosegue con il programma principale;
3. se i byte ricevuti sono 11 controlla che il primo valga 0x00, se non coincide scarta tutti i byte e prosegue con il programma principale;
4. se il primo byte vale 0x00 concatena il byte 4 al byte 3 e li trasmette su SPI, attende 10mS;
5. concatena il byte 6 al byte 5 e li trasmette su SPI, attende 10mS;
6. concatena il byte 8 al byte 7 e li trasmette su SPI, attende 10mS;
7. concatena il byte 10 al byte 9 e li trasmette su SPI.

Tutti punti precedentemente descritti si trovano nel codice all'interno di una funzione di tipo "*While(1)*" che rende ciclica la loro successione. Vediamo ora come tutti questi punti sono stati tradotti in codice:

```
/*****************************************************************
* Function:        BoroInit(void)
* Input:           void
* Output:          void
* Overview:        inizializzazione chip boro
*****************************************************************/
void BoroInit(void){
	BYTE numBytesRead;                //variabile x numero di byte
	if(mUSBUSARTIsTxTrfReady()){            //controllo che usb sia pronta
		//ottengo il numero di byte ricevuti
		numBytesRead=getsUSBUSART(USB_Out_Buffer,64);

		//controllo se ho ricevuto gli 11 byte
		if(numBytesRead == 11){

			//controllo se il primo byte è quello di sincronismo
			if(USB_Out_Buffer[0]|=0x00){
			return;
			}
			//copio solo i byte utili in un array BOROSPI[8]
			BYTE i;
			for(i=2;i<10;i++){
				BOROSPI[i-2]=USB_Out_Buffer[i];
			}
			somma_spi=0;  //variabile d'appoggio a 16bit
			//concateno e trasmetto a pacchetti di 16 bit
			somma_spi=BOROSPI[1]<<8 | BOROSPI[0];
			SpiChnWriteC(3,somma_spi);
			delay_ms(10);
			somma_spi=BOROSPI[3]<<8 | BOROSPI[2];
			SpiChnWriteC(3,somma_spi);
			delay_ms(10);
			somma_spi=BOROSPI[5]<<8 | BOROSPI[4];
			SpiChnWriteC(3,somma_spi);
			delay_ms(10);
			somma_spi=BOROSPI[7]<<8 | BOROSPI[6];
			SpiChnWriteC(3,somma_spi);
		}
	}
}//BoroInit
```

Filtraggio

Introduzione

Come è stato detto nell'introduzione l'interfaccia impedenziometrica appartiene alla categoria dei modulatori sigma-delta (ΣΔ), essa infatti traduce le grandezze di tipo analogico ai propri ingressi in segnali digitali così modulati.

Teoria

La modulazione sigma-delta è un metodo per tradurre segnali ad elevata risoluzione in segnali a bassa risoluzione tramite la tecnica di modulazione a densità di impulsi.

Questa modulazione è fortemente usata nei convertitori analogico-digitale (ADC) in quanto un convertitore che implementa questa tecnica può facilmente raggiungere risoluzioni molto elevate utilizzando tecnologie CMOS a basso costo. Un segnale di tipo sigma-delta si presenta come un treno di impulsi, questi impulsi sono tutti identici tra loro e di essi si conosce ampiezza **V** e durata **dt**, le quali rimangono sempre costanti, ciò che varia invece è l'intervallo di separazione tra di essi.

L'intervallo di separazione infatti è determinato dall'anello in retroazione in modo tale che un ingresso in tensione bassa produca un intervallo lungo tra gli impulsi, mentre un livello alto di tensione in ingresso produca un intervallo breve *Fig.15*.

L'anello di retroazione è costituito in modo tale che l'integrale dell'ingresso sia associato all'integrale del treno di impulsi.

Il conteggio finale sull'uscita rappresenta quindi la digitalizzazione del segnale in ingresso, esso si determina contando gli impulsi prodotti in un determinato periodo di tempo pari a Ndt, da qui la dicitura Σ (sommatoria). L'integrale del treno di impulsi che viene prodotto durante un intervallo di durata Ndt vale ΣVdt, pertanto il valore medio della grandezza in ingresso nel periodo considerato sarà pari a VΣ/N.

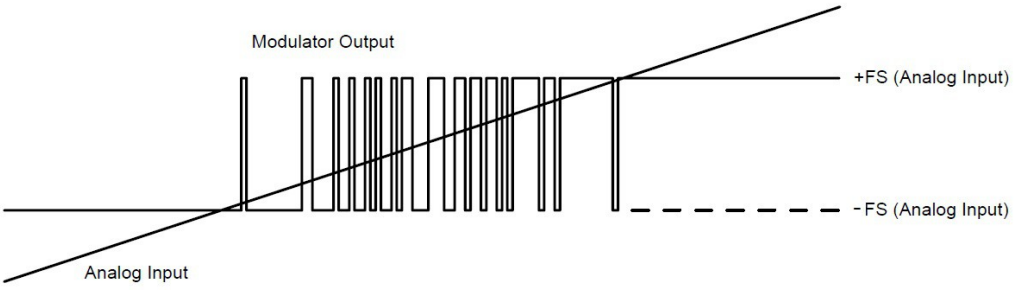


Fig.15 Modulazione sigma-delta

Il numero N di campionamenti nell'intervallo di tempo è anche detto OSR (Over Sampling Ratio) o *Fattore di decimazione*, come è intuibile un OSR maggiore porta anche ad una risoluzione maggiore, in quanto vi è un aumento della finestra di osservazione, ma questo avviene a scapito della banda in quanto occorrerà attendere la collezione di N campioni prima di poter calcolarne la media.

Spesso sono tuttavia richieste tecniche di filtraggio più elaborate di una semplice media temporale, il filtraggio di tipo $Sinc^K$, dove K è l'ordine del filtro e la cui nomenclatura richiama il risultato della trasformata di *Fourier* della funzione *Rect*, è spesso utilizzato per arrivare alla profondità di conversione desiderata, in particolare la funzione Rect che rappresenta la finestra di osservazione nel dominio del tempo produce nel dominio delle frequenze una funzione di tipo Sinc, si osserva quindi che quando è necessario avere una maggiore pendenza del primo lobo della funzione Sinc, e quindi una maggiore selettività del filtro, occorre operare una convoluzione nel dominio del tempo di più finestre di osservazione.

In particolare alcuni casi notevoli di filtraggio sono:

H (*f*)	h (*t*)
Sinc	Rect (media semplice)
$Sinc^2$	Rect * Rect
$Sinc^3$	Rect * Rect * Rect

Filtro $Sinc^3$

Come abbiamo detto precedentemente un importante tipo di filtraggio digitale per questo tipo di segnale è denominato $Sinc^K$, questo tipo di filtraggio è molto interessante a livello implementativo in quanto non richiede l'uso di moltiplicatori digitali che richiedono notevoli risorse alla CPU. Questo metodo di elaborazione consiste nell'implementare in cascata un numero K di blocchi accumulatori operanti alla frequenza di campionamento *fs*, seguiti da un numero di altrettanti blocchi differenziatori operanti alla frequenza *fs/OSR*. Una rappresentazione a blocchi di questa architettura è mostrata in *Fig.16*:

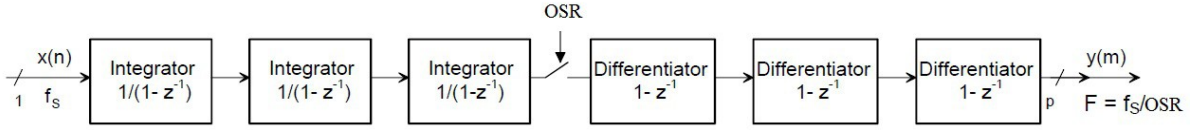


Fig.16 Topologia filtro $Sinc^3$

La funzione di trasferimento che caratterizza un filtro $Sinc^K$ è la seguente:

$$H(z) = \left(\frac{1}{OSR} \cdot \frac{1 - z^{-OSR}}{1 - z^{-1}} \right)^K$$

sostituendo Z con e^{-j} otteniamo la risposta in frequenza del filtro: $\left| H(e^{j\omega}) \right| = \left(\frac{1}{OSR} \cdot \frac{\sin(\omega \cdot OSR/2)}{\sin(\omega/2)} \right)^K$ dove $\omega = 2\pi \frac{f}{f_S}$

in *Fig.17* viene mostrato un esempio di risposta in frequenza di un filtro del terzo ordine ($Sinc^3$) corrispondente ad una struttura come quella di *Fig.16*, dove è stato utilizzato un fattore di decimazione OSR=16.

Gli zeri dello spettro sono in corrispondenza dei multipli della frequenza di decimazione. In particolare la relazione che lega la frequenza di campionamento del segnale sigma-delta e il data rate di uscita sarà:

$$DataRate = \frac{f_s}{OSR}$$

Quindi una scelta oculata del valore di OSR è molto importante anche per determinare la frequenza di taglio del filtro digitale.

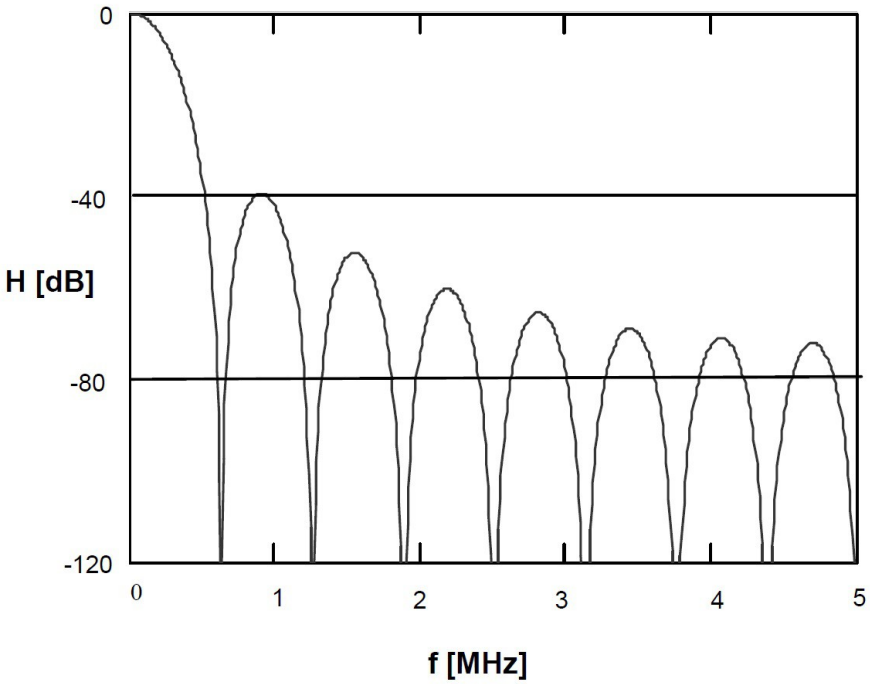


Fig.17 Risposta in frequenza di un filtro $Sinc^3$ con fs=10MHz e OSR=16

Come è stato detto la risoluzione teorica di questo tipo di conversione è funzione del parametro OSR e dell'ordine del filtro, occorre ricordare che si tratta di risoluzione teorica in quanto i bit effettivi dipendono dalle caratteristiche del convertitore, tuttavia la dimensione della parola in uscita da un filtro di ordine K è più larga di quella in ingresso di un fattore *p*, questo fattore come appena detto è legato all'ordine del filtro e al fattore di decimazione OSR come segue:

$$p = K \cdot \log_2 OSR$$

in particolare essendo la larghezza della parola in ingresso pari ad 1 bit, la dimensione della parola in uscita dal filtro sarà p bit.

Supponendo una frequenza di campionamento *fs = 10MHz* e un filtro di tipo Sinc³ è possibile confrontare i risultati per un OSR che va da 4 a 256 *Fig.18*:

Decimation	Data Rate (kHz)	Output Word Size (bits)	Filter Response f-3dB (kHz)
4	2,500.0	6	655
8	1,250.0	9	327.5
16	625.0	12	163.7
32	312.5	15	81.8
64	156.2	18	40.9
128	78.1	21	20.4
256	39.1	24	10.2

Fig.18 OSR a confronto

Implementazione

Ricordando le caratteristiche dell'interfaccia impedenziometrica descritte nel paragrafo introduttivo occorre ora spiegare con quale tecnica vengono codificate le modulazioni sigma-delta che portano le informazioni sia sulla parte reale che sulla parte immaginaria dell'impedenza oggetto di misura.

Per ogni core presente all'interno dell'interfaccia troviamo all'uscita un pin denominato **OUTx** ed un segnale di sincronismo **Sync** comune a tutti i core. La tecnica di lettura delle due parti che compongono il numero complesso, reale ed immaginario associati alla lettura dell'impedenza, sta nell'andare a campionare il segnale OUTx in corrispondenza del fronte di salita e del fronte di discesa del segnale Sync, campionando infatti l'uscita sul fronte di salita del segnale di sincronismo si ottiene un campione valido per la parte reale dell'impedenza, mentre campionando sul fronte di discesa si ottiene un campione valido per la parte immaginaria *Fig.19*.

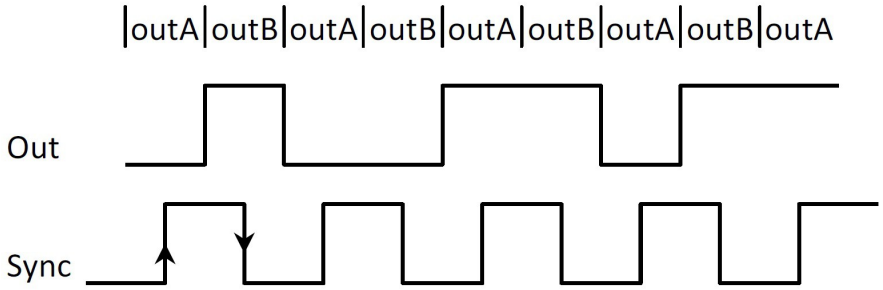


Fig.19 Campionamento segnale d'uscita

Ci si rende subito conto che per ogni segnale d'uscita occorre implementare 2 strutture di filtraggio indipendenti, e considerando che le uscite sono 4 si ottiene un totale di 8 strutture indipendenti, ci limiteremo pertanto a vedere come implementare una struttura di filtraggio che poi verrà replicata parallelamente per ogni flusso dati.

Le specifiche richieste dal chip BORO prevedono una parola di uscita di 24bit dei quali i 16 più significativi rappresentano l'effettiva conversione digitale della grandezza elettrica di cui si sta effettuando la misura, ricordando la formula che permette di calcolare il numero di bit della parola in uscita dal filtro: $p = K \cdot \log_2 OSR$ occorre quindi implementare un filtraggio di ordine 3 con un OSR pari a 256. La struttura che stiamo analizzando come mostrato in *Fig.16* può essere implementata mettendo in cascata 3 integratori e 3 derivatori, dove i blocchi integratori funzioneranno con la frequenza di campionamento *fs* prodotta dal segnale si sincronismo, mentre i blocchi derivatori funzioneranno ad una

frequenza scalata del fattore di decimazione, ovvero *fs/OSR*.

Analizzando la *Fig.16* occorre precisare che le frecce che collegano i blocchi rappresentano un trasferimento con parallelismo 24 bit e che quindi nel codice verranno mappate in registri che per la natura dell'architettura saranno a 32 bit, ma questo non rappresenta affatto un problema in quanto verranno presi in considerazione semplicemente solo i 24 bit meno significativi del registro. La realizzazione di una struttura di questo tipo risulta particolarmente agevole su dispositivi come FPGA dove si ottiene fisicamente il parallelismo necessario al corretto funzionamento, quando invece si cerca di replicare questa struttura attraverso un algoritmo occorre prestare particolare attenzione a possibili effetti indesiderati di propagazione del dato lungo la catena.

L'obbiettivo più importante di questa parte del lavoro è stato quindi quello di tradurre la struttura presentata in *Fig.16* in un algoritmo software che una volta terminato sortisca gli stessi effetti di una struttura fisica. L'algoritmo che è stato individuato per eliminare i possibili effetti di propagazione indesiderata del dato è presentato in *Fig.20*, esso viene percorso non appena viene riconosciuto un fronte. Si può notare che macroscopicamente l'algoritmo è composto da 2 blocchi, il blocco più in alto descrive la propagazione del dato attraverso la catena di integratori e viene percorsa alla frequenza *fs*, ovvero ad ogni fronte rilevato, il blocco più in basso invece descrive la propagazione del dato attraverso la catena di derivatori ed opera ad *fs/OSR* ovvero ad ogni OSR fronti. Al termine della sessione di derivazione viene portato ad 1 logico un flag che funge da variabile di sincronizzazione con il resto del programma, andando a valore logico alto infatti avvisa il *main program* che è stato prodotto un nuovo dato.

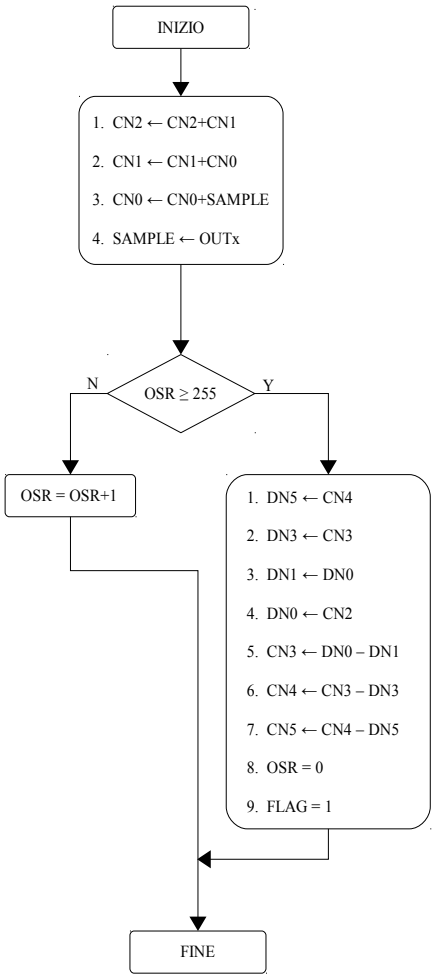


Fig.20 Algoritmo di filtraggio software

La gestione di questo algoritmo, come del resto espresso nella *Fig.16* è scandita temporalmente dal rilevamento dei fronti presenti sul segnale di sincronismo. La scelta più intelligente in osservanza della modalità di funzionamento è pertanto quella di gestire il filtraggio ad interrup, nello specifico è stato sfruttato un particolare tipo di interrupt gestito dal MCU denominato Change Notice (CN) la cui caratteristica è quella di intervenire ogni qual volta il pin a cui è assegnato cambia di stato, questo è particolarmente utile in funzione del fatto che come mostrato in *Fig.19* il campionamento avviene sia sul fronte di salita sia sul fronte di discesa. Ad ogni interrupt quindi la ISR dedicata andrà a verificare se si tratta di un fronte di salita o se si tratta di un fronte di discesa ed eseguirà rispettivamente la routine di filtraggio relativa alla parte reale o alla parte immaginaria. Quando entrambe le routine producono un nuovo dato un *"and"* logico tra i due flag consente al *main program* di capire che i dati sono pronti per essere formattati e trasmessi al PC.

Mostriamo ora come è stata implementata la ISR precedentemente descritta, per brevità viene riportata solo la parte relativa ad un singolo core in quanto per gli altri la struttura viene replicata cambiando solo gli indici:

```
/*****************************************************************
* Function:         CNInterrupt( void)
* Input:            None
* Output:           None
* Overview:         change notice interrupt
*****************************************************************/
// change notification interrupt service routine
void __ISR( _CHANGE_NOTICE_VECTOR, ipl6) CNInterrupt( void){
//campiono il segnale presente su OUT4 e lo salvo in "dato"
if(adc_data_in == 1){
dato=1;
}else{dato=0;}
//controllo il tipo di fronte avvenuto(BOROCLK=> CN11=RG9)
if ( BOROCLK == 1){
//se fronte di salita processo il dato Reale
       cn2_4R=(cn2_4R+cn1_4R);
       cn1_4R=(cn1_4R+cn0_4R);
       cn0_4R=(cn0_4R+sample_4R);
       sample_4R=dato;
       if(osr_4R>=255){           //processo di derivazione ogni OSR
                     dn5_4R=cn4_4R
                     dn3_4R=cn3_4R;
                     dn1_4R=dn0_4R;
                     dn0_4R=cn2_4R;
                     cn3_4R=(dn0_4R-dn1_4R);
                     cn4_4R=(cn3_4R-dn3_4R);
                     cn5_4R=(cn4_4R-dn5_4R);
                     osr_4R=0;
                     stampa_R=TRUE;      // variabile di sincronizzazione
       }else{osr_4R++;}    // incremento il conteggio per OSR
}else{ //se era un fronte di discesa processo dato immaginario
       cn2_4I=(cn2_4I+cn1_4I);
       cn1_4I=(cn1_4I+cn0_4I);
       cn0_4I=(cn0_4I+sample_4I);
       sample_4I=dato;
       if(osr_4I>=255){           //processo di derivazione ogni OSR
                     dn5_4I=cn4_4I
                     dn3_4I=cn3_4I;
                     dn1_4I=dn0_4I;
                     dn0_4I=cn2_4I;
                     cn3_4I=(dn0_4I-dn1_4I);
                     cn4_4I=(cn3_4I-dn3_4I);
                     cn5_4I=(cn4_4I-dn5_4I);
                     osr_4I=0;
                     stampa_I=TRUE;      // variabile di sincronizzazione
       }else{osr_4I++;}    // incremento il conteggio per OSR
}
mCNClearIntFlag();      // l'interrupt servito, resetto il bit relativo
};// CNInterrupt()
```

Trasmissione dei dati al PC

Introduzione

La possibilità di instaurare una comunicazione tra più dispositivi elettronici è sicuramente una base fondamentale dell'elettronica odierna, con l'evoluzione dell'elettronica e delle telecomunicazioni lo studio di metodi trasmissivi e di protocolli per il dialogo tra più dispositivi è stato oggetto di grande ricerca. Nel corso del tempo sono stati proposti da parte dei principali produttori di dispositivi una serie di protocolli di comunicazione, il successo sul mercato di taluni di essi, particolarmente prestanti, ha portato a processi di standardizzazione dei principali protocolli di comunicazione e di interfacciamento tra dispositivi, molti dei quali li ritroviamo oggi sui nostri PC. Anche in questo lavoro di tesi si è fatto uso di protocolli di comunicazione particolarmente affermati come USB e SPI.

In questo capitolo verrà trattato nello specifico il modulo USB descrivendone il suo utilizzo, la sua implementazione e il processo di formattazione dei dati che sta alla base della comunicazione tra il MCU e il PC.

USB

La porta USB *(Universal Serial Bus)* è uno standard di comunicazione seriale che permette il dialogo tra due dispositivi, questo protocollo è stato progettato per consentire a più periferiche di essere connesse usando una sola interfaccia standardizzata e solo alcuni tipi di connettori, questo standard inoltre supporta l'*hot-swap* cioè la possibilità di connettere dispositivi senza dover riavviare l'host.
I principali ruoli che un sistema USB può assumere sono:

- **USB host** : ciascuna comunicazione di tipo USB ha un solo host, l'host controller viene implementato spesso su PC, è una combinazione di hardware, firmware e software. Monitorizza le connessioni, le configurazioni e le rimozioni dei device, regola i flussi dati e spesso alimenta i device.
- **USB device** : possono essere dispositivi *hub* o periferiche che svolgono precise funzioni, devono gestire il protocollo USB e supportare le operazioni di configurazione, reset e fornire informazioni all'host sulle proprie caratteristiche.

La porta USB per come è stato studiato il protocollo risulta più complessa della porta seriale UART, ciononostante è stata ideata in modo che possa essere utilizzata potendo scavalcare spesso la complessità racchiusa nel suo protocollo.

Nel lavoro di tesi che seguirà non si entrerà nel merito dei dettagli associati al protocollo, si farà uso infatti di driver già pronti e della libreria Microchip MCHPFSUSB.
Dal momento che non si entrerà in merito al protocollo USB perché si farà uso di una porta seriale emulata ci occuperemo di descrivere in che modo viene utilizzata questa libreria e verranno descritte tutte le impostazioni relative al microcontrollore per il corretto funzionamento.

Oltre alle connessioni fisiche della porta USB con il sistema occorre prestare particolare attenzione alle impostazioni del microcontrollore, descriveremo ora quindi le impostazioni che permettono il corretto funzionamento della libreria Microchip.
Analizziamo come prima cosa la circuiteria di clock interna al microcontrollore e riportata in *Fig.21*:

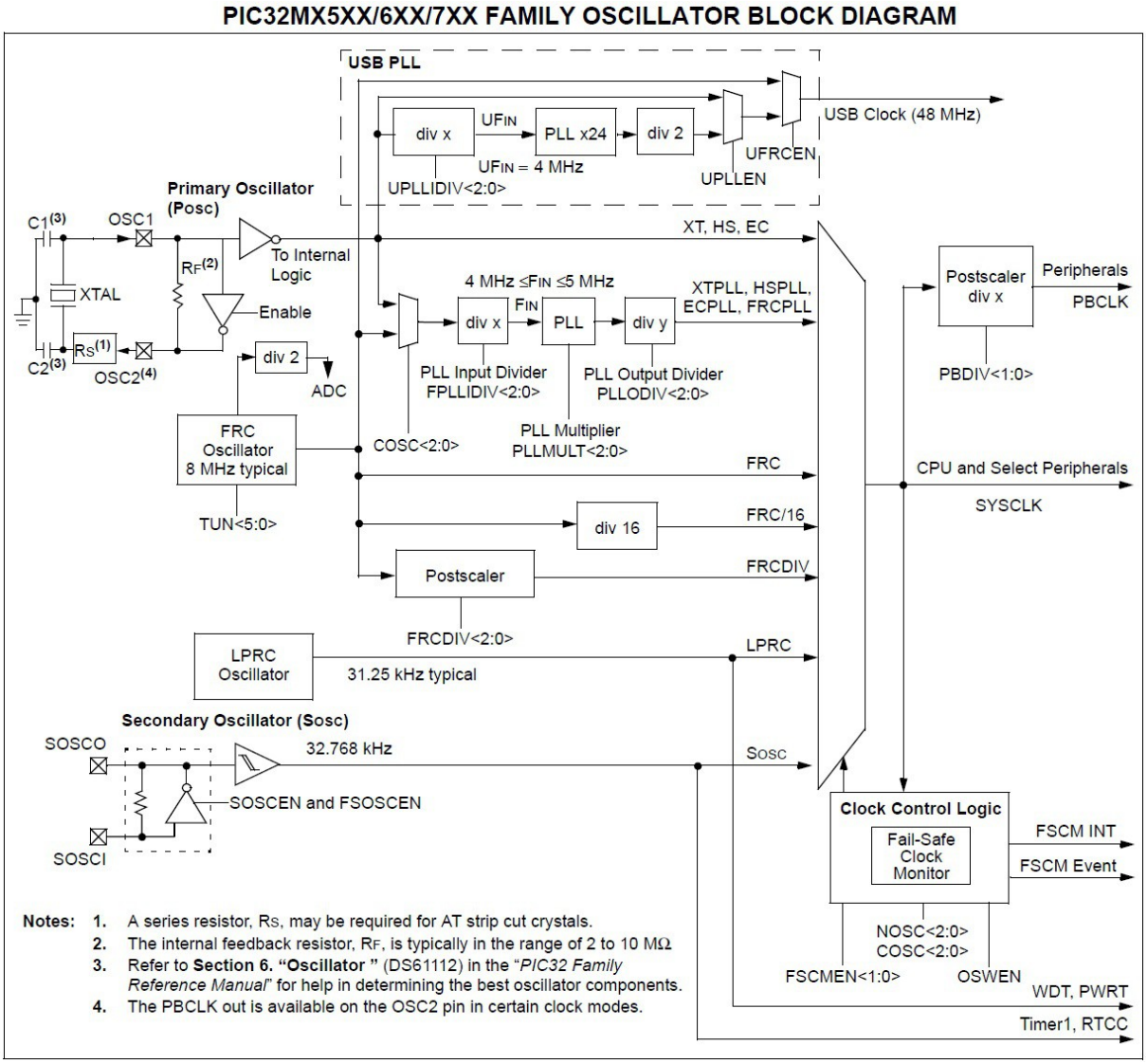


Fig.21 schema a blocchi della circuiteria di clock

Come tutti i PIC della famiglia 32Bit anche il PIC32MX795F512H dispone di due oscillatori esterni, ovvero il primario e il secondario, più un oscillatore interno ad 8MHz.

Per garantire il funzionamento della USB occorre quindi combinare correttamente questa struttura di gestione del clock in modo che all'hardware di gestione USB giunga un clock pari a 48MHz. Osservando la struttura in *Fig.21* e ricordando che disponiamo sulla nostra scheda di sviluppo di un quarzo ad 8MHz connesso all'oscillatore primario possiamo scartare l'ipotesi di utilizzare l'oscillatore interno e far giungere il clock dell'oscillatore primario al blocco *USB PLL,* questo lo si ottiene attraverso il settaggio del bit **UFRCEN**. È possibile calcolare ora il valore di divisione da assegnare ad **UPLLIDIV** coerentemente con la necessità di avere 4MHz all'ingresso del PLL, sapendo che l'oscillatore è connesso ad un quarzo ad 8MHz il valore da assegnare è 2. Inoltre occorre abilitare il bit relativo a **UPLLEN** per assicurarci di prelevare il segnale a valle del PLL.

Le impostazioni ora descritte devono essere effettuate per mezzo dei registri di configurazione, ovvero facendo uso della direttiva `#pragma` riportiamo di seguito le righe di codice utilizzate a tale scopo:

```
#pragma config UFRCEN  = ON
#pragma config UPLLDIV = DIV_2
#pragma config UPLLEN  = ON
```

La comunicazione tra PC e microcontrollore che è stata implementata in questo lavoro di tesi fa uso di una classe CDC (Comunication Device Class) che permette di trattare la porta USB come una normale porta seriale, questo permette di scrivere e leggere sulla porta USB attraverso funzioni di libreria molto simili a quelle di una semplice porta seriale.

La classe CDC permette infatti di richiamare il proprio driver all'interno del sistema operativo, per mezzo del quale il PC comunica attraverso il bus USB emulando la porta seriale, una volta connesso il dispositivo verrà infatti creata una una porta seriale COM virtuale attraverso la quale sarà possibile leggere e scrivere dati.

Funzioni di libreria

La libreria USB, ovvero il Framework, mette a disposizione delle funzioni attraverso le quali è possibile leggere e scrivere sulla porta USB come per una porta seriale, di seguito presenteremo e discuteremo le funzioni utilizzate in questo lavoro di tesi:

- **mUSBUSARTIsTxTrfReady( )**

  Questa funzione è costituita da una macro, permette di controllare se il modulo USB è pronto per poter inviare dati.

La funzione ritorna 1 se il modulo può trasmettere dati, 0 altrimenti. É molto importante non richiamare questa funzione per mezzo di un istruzione di tipo *While* in quanto si costituirebbe un istruzione di tipo bloccante che impedirebbe il corretto funzionamento del resto della libreria, e dunque impedirebbe di aggiornare lo stato delle variabili.

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

  questa funzione consente di trascrivere i dati dalla RAM al modulo USB, il puntatore ***data**, rappresenta il puntatore alla variabile o all'array in cui è contenuto il dato da trasmettere. Questa funzione deve essere utilizzata quando è nota la lunghezza dell'array, infatti il secondo parametro da passare alla funzione, rappresenta proprio la lunghezza dell'array, cioè il numero di byte che lo compongono.

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

  Questa funzione, viene utilizzata per leggere i dati dal modulo USB. La funzione prende in ingresso il puntatore alla struttura dati della variabile o dell'array in cui verranno trascritti i dati e la sua lunghezza, cioè il numero di byte che ci si aspetta siano presenti nel buffer USB o che siamo interessati a leggere. La funzione ritorna il numero di byte ricevuti, se non ce ne sono ritorna 0.

- **CDCTxService( )**

  Questa funzione deve essere richiamata almeno una volta per ogni ciclo di scrittura e serve alla gestione del Buffer di trasmissione, in particolare è necessario al corretto funzionamento di **USBUSARTIsTxTrfReady( )** , che senza il richiamo di questa funzione infatti ritornerebbe sempre 0.

Esistono inoltre altre funzioni di libreria per la seriale emulata attraverso USB che però non verranno trattate in questo elaborato perché non utilizzate nel programma. Una raccolta completa delle funzioni utilizzabili è possibile trovarla nella documentazione allegata alla libreria Microchip.

Utilizzo della libreria

Una volta installata la libreria si hanno a disposizione diversi progetti generali, il cui il nome più appropriato è quello di Framework, e che implementano le principali possibilità di utilizzo della porta USB. Questi Framework sono tuttavia molto generali, e per adattarsi ad una configurazione hardware specifica richiedono alcune modifiche. In particolare il codice presentato in questo elaborato fa uso del Framework CDC, per mezzo del quale è possibile utilizzare

l'emulazione della porta seriale attraverso USB. Analizzeremo di seguito le modifiche necessarie al fine di poter utilizzare il Framework CDC sul nostro sistema.
Come prima modifica occorre mappare le porte di I/O in funzione del nostro hardware, queste assegnazioni sono riportate nel file *HardwareProfile.h* presente nella directory del Framework, ad esempio mostriamo come è stato mappato il led che indica lo stato della connessione USB, e che sul nostro hardware è connesso al pin RD5:

```
#define mLED_1    LATDbits.LATD5
```

Una volta mappata la configurazione hardware attraverso l'uso della direttiva `#define` occorre impostare la modalità di funzionamento dell'USB, cioè occorre dichiarare se gestire il modulo USB ad Interrupt o a Polling.

Nelle prime versioni della libreria la gestione era infatti effettuata solo a Polling, rendendo l'utilizzo della libreria meno snella, mentre nelle ultime versioni si è introdotto il funzionamento ad interrupt, che è anche la modalità di funzionamento che è stata scelta per questo progetto.

Per dichiarare la modalità di funzionamento occorre accedere al file *usb_config.h* presente nella directory del Framework e scommentare solo la modalità desiderata:

```
// Select polled or interrupt-driven operation

//#define USB_POLLING

#define USB_INTERRUPT
```

Apportate queste modifiche al Framework la classe CDC è implementata sul nostro hardware e il codice se compilato è caricato sul MCU ci permette già di avere un device che collegato al PC stabilisce una comunicazione di tipo seriale emulata.
A questo punto è possibile utilizzare tutte le funzioni di libreria per i nostri obbiettivi.

Trasmissione dei dati

Per comprendere come agisce il software di cui è equipaggiato il MCU quando un nuovo dato viene reso disponibile dalla struttura di filtraggio, occorre come prima cosa valutare il protocollo da osservare per far si che il Software lato PC assimili correttamente il dato.

La trasmissione dell'informazione avviene attraverso lo scambio di una stringa composta da 25byte consecutivi come rappresentato in *Fig.22*:

Byte 1	Sincronismo 0xFF
Byte 2	Core1 Reale Byte 1 (MSB)
Byte 3	Core1 Reale Byte 2
Byte 4	Core1 Reale Byte 3 (LSB)
Byte 5	Core1 Immaginario Byte 1 (MSB)
Byte 6	Core1 Immaginario Byte 2
Byte 7	Core1 Immaginario Byte 3 (LSB)
Byte 8	Core2 Reale Byte 1 (MSB)
Byte 9	Core2 Reale Byte 2
Byte 10	Core2 Reale Byte 3 (LSB)
Byte 11	Core2 Immaginario Byte 1 (MSB)
Byte 12	Core2 Immaginario Byte 2
Byte 13	Core2 Immaginario Byte 3 (LSB)
Byte 14	Core3 Reale Byte 1 (MSB)
Byte 15	Core3 Reale Byte 2
Byte 16	Core3 Reale Byte 3 (LSB)
Byte 17	Core3 Immaginario Byte 1 (MSB)
Byte 18	Core3 Immaginario Byte 2
Byte 19	Core3 Immaginario Byte 3 (LSB)
Byte 20	Core4 Reale Byte 1 (MSB)
Byte 21	Core4 Reale Byte 2
Byte 22	Core4 Reale Byte 3 (LSB)
Byte 23	Core4 Immaginario Byte 1 (MSB)
Byte 24	Core4 Immaginario Byte 2
Byte 25	Core4 Immaginario Byte 3 (LSB)

Fig.22 protocollo MCU - PC

Analizzando la stringa si osserva come primo byte un valore esadecimale 0xFF utilizzato dal protocollo come trama di sincronismo, successivamente vengono trasmessi in sequenza i valori delle letture a 24 bit per ogni core, alternando parte reale e parte immaginaria.

Come precedentemente descritto nel paragrafo relativo al filtraggio, i valori della misura in uscita dal filtro Sinc³ sono rappresentati con una profondità di 24 bit ed immagazzinati in registri a 32bit, in quanto questa è la dimensione dell'architettura su cui stiamo operando, nasce quindi l'esigenza di sviluppare una funzione che recuperi solo i 24 bit meno significativi del registro e che li suddivida in 3 byte per la successiva trasmissione.

La struttura di filtraggio inoltre, per come è costituita, fornisce in uscita un valore compreso tra $0 \div 2^{24}+1$ proporzionale alla densità di impulsi campionati sull'ingresso, tuttavia essendo 2^{24} il massimo valore rappresentabile con 3 byte occorre prevedere un controllo di *overflow* che limiti il valore del dato in uscita al massimo valore rappresentabile.

La funzione che è stata realizzata per svolgere i compiti appena descritti prende in ingresso gli 8 valori che vengono prodotti dalle altrettante strutture di filtraggio, e opera su una stringa di 25 byte dichiarata in fase di inizializzazione e che verrà successivamente trasmessa attraverso la seriale emulata. Per prima cosa la funzione si occupa di limitare il valore dei registri in ingresso a 2^{24} :

```
/*****************************************************************************
* Function:        void OutString(int, int, int, int, int, int, int, int)
* Input:           None
* Output:          None
* Overview:        Questa funzione modifica la stringa di uscita secondo il protocollo.
*****************************************************************************/
void OutString(unsigned int num1 , unsigned int num1_i , unsigned int num2 ,
               unsigned int num2_i , unsigned int num3 , unsigned int num3_i ,
               unsigned int num4 , unsigned int num4_i){
//limite sul valore massimo che le misure possono raggiungere
if(num1>fondoscala){num1=fondoscala;};
if(num2>fondoscala){num2=fondoscala;};
if(num3>fondoscala){num3=fondoscala;};
if(num4>fondoscala){num4=fondoscala;};
if(num1_i>fondoscala){num1_i=fondoscala;};
if(num2_i>fondoscala){num2_i=fondoscala;};
if(num3_i>fondoscala){num3_i=fondoscala;};
if(num4_i>fondoscala){num4_i=fondoscala;};
```

In questo modo viene assicurato che non vi sia overflow qualora un ingresso andasse in saturazione, in particolare il valore di `'fondoscala'` è una costante e viene dichiarato in fase di inizializzazione ad un valore di 2^{24} .

Ora analizziamo la parte restante della funzione, ovvero il processo di isolamento dei byte che compongono il valore a 24bit e la successiva composizione della stringa di 25 byte che verrà trasmessa al PC:

```
string_out[0]=0xff;                          //trama di sincronismo
string_out[1]=(num1>>16);                    //bit più significativi reale core1
string_out[2]=((num1>>8) & 0x000000ff);       //bit intermedi reale core1
string_out[3]=(num1 & 0x000000ff);           //bit meno significativi reale core1

string_out[4]=(num1_i>>16);                  //bit più significativi immaginario core1
string_out[5]=((num1_i>>8) & 0x000000ff);     //bit intermedi immaginario core1
string_out[6]=(num1_i & 0x000000ff);         //bit meno significativi immaginario core1
string_out[7]=(num2>>16);                    //bit più significativi reale core2
string_out[8]=((num2>>8) & 0x000000ff);       //bit intermedi reale core2
string_out[9]=(num2 & 0x000000ff);           //bit meno significativi reale core2
string_out[10]=(num2_i>>16);                 //bit più significativi immaginario core2
string_out[11]=((num2_i>>8) & 0x000000ff);    //bit intermedi immaginario core2
string_out[12]=(num2_i & 0x000000ff);        //bit meno significativi immaginario core2

string_out[13]=(num3>>16);                   //bit più significativi reale core3
string_out[14]=((num3>>8) & 0x000000ff);      //bit intermedi reale core3
string_out[15]=(num3 & 0x000000ff);          //bit meno significativi reale core3
string_out[16]=(num3_i>>16);                 //bit più significativi immaginario core3
string_out[17]=((num3_i>>8) & 0x000000ff);    //bit intermedi immaginario core3
string_out[18]=(num3_i & 0x000000ff);        //bit meno significativi immaginario core3
string_out[19]=(num4>>16);                   //bit più significativi reale core4
string_out[20]=((num4>>8) & 0x000000ff);      //bit intermedi reale core4
string_out[21]=(num4 & 0x000000ff);          //bit meno significativi reale core4
string_out[22]=(num4_i>>16);                 //bit più significativi immaginario core4
string_out[23]=((num4_i>>8) & 0x000000ff);    //bit intermedi immaginario core4
string_out[24]=(num4_i & 0x000000ff);        //bit meno significativi immaginario core4
}//out string
```

La funzione "OutString" viene richiamata dal main program quando la struttura di filtraggio segnala attraverso i flag "stampa_R" e "stampa_I" che nuovi dati sono disponibili, entrambe i flag devono essere a valore logico 1, per questo vengono messi a confronto con una istruzione di *"and"* logico. Una volta eseguita la funzione la stringa è aggiornata con gli ultimi dati disponibili e può essere trasmessa al PC. Riportiamo di seguito la porzione del *main program* che si occupa di richiamare quando necessario la funzione "OutString" e successivamente di trasmettere la stringa :

```
if((stampa_R==1)&&(stampa_I==1)){ //sono disponibili nuovi dati?
//passo nuovi dati alla funzione
OutString(  cn5_1R, cn5_1I, cn5_2R, cn5_2I, cn5_3R, cn5_3I, cn5_4R, cn5_4I);
       if(mUSBUSARTIsTxTrfReady()){      //controllo se  modulo USB è disponibile
        putUSBUSART(string_out,25); // trasmetto la stringa
              stampa_R=0;          //azzero le variabili di sincronizzazione
              stampa_I=0;
       }
}
```

Generazione bitstream onda sinusoidale tramite PWM

In questo paragrafo verrà descritta la tecnica utilizzata per la generazione del segnale di riferimento necessario all'interfaccia impedenziometrica per la corretta lettura delle impedenze in esame.

Verranno inoltre trattate nel dettaglio tutte le periferiche utilizzate a tale scopo.

TIMER

Nel progetto descritto in questo elaborato, come già discusso nella parte introduttiva, è stato necessario implementare un sistema di generazione di una particolare forma d'onda essenziale all'interfaccia impedenziometrica per la corretta lettura del sensore, lasciando i dettagli implementativi della funzione al paragrafo relativo, in questa sezione parleremo di una parte fondamentale della struttura di questa funzione, cioè il timer. La funzione che implementa la generazione della forma d'onda deve essere in grado di modificare il duty cycle del segnale PWM ogni 20 µS, nasce quindi la necessità di un sistema di conteggio del tempo che possa generare un interrupt ogni volta che sia il momento di cambiare duty-cycle, per questo si è fatto uso di un timer interno al microcontrollore in particolare il Timer 2.

Il vantaggio di utilizzare questi timer interni al microcontrollore risiede proprio nella capacità del timer di generare un interruzione allo scadere del tempo, lasciando la CPU libera di svolgere altre mansioni per il restante tempo. Il timer è costituito da un contatore programmabile interno al MCU, esso conta un numero definito dal programmatore di impulsi che possono provenire dall'esterno o opportunamente scalati dal clock di sistema, il registro del contatore viene continuamente comparato ad un registro nel quale è stato precaricato il numero di conteggi necessari, quando i due registri si equivalgono viene generata l'interruzione.

La struttura del timer è mostrata in *Fig.23*, possiamo individuare alcuni componenti come:

- TMRx contatore del timer, incrementa ad ogni impulso proveniente dal prescaller
- PRx contiene il valore di confronto
- TxCON registro di controllo del timer

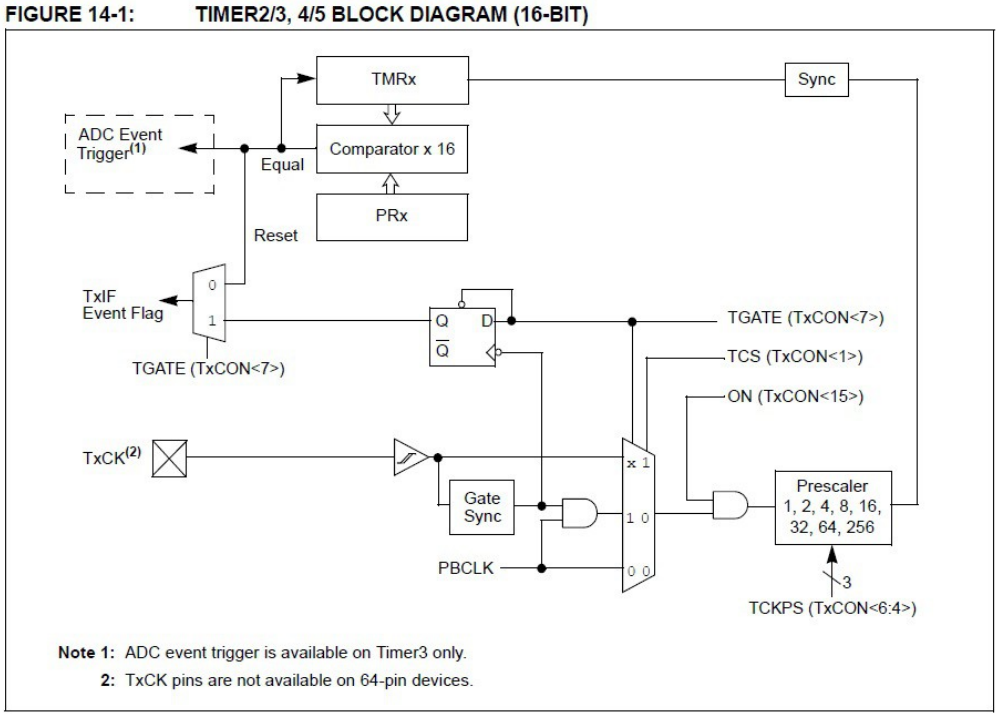


Fig.23 struttura interna del timer

Vediamo ora come è possibile configurare questa struttura per generare interrupt ogni intervallo di tempo prestabilito.

Per prima cosa si procede impostando il registro di configurazione TxCON del timer in accordo alle nostre esigenze, esso al proprio interno contiene:

- **ON** abilitazione del conteggio (bit 15);
- **SIDL** interrompe il conteggio se il MCU entra in *idle mode* (bit 13);
- **TWDIS** consente o meno di aggiornare il valore del timer in tempo reale (bit12);
- **TGATE** consente di abilitare il *Gate time accumulator mode,* in questa modalità il timer viene incrementato dal clock di sistema cioè ad ogni ciclo di istruzione (bit 7);
- **TCKPS** questi bit consentono di impostare il valore di prescaller in ingresso al timer (bit <6:4>)

Presentiamo ora i passaggi che ci permettono operativamente di produrre un conteggio che generi un iterrupt ogni 20 μS.
Per prima cosa, visto che non vi è alcun segnale esterno utile alla sincronizzazione del conteggio decideremo per l'opzione di utilizzare il clock di sistema quale base dei tempi per il conteggio.
Occorre valutare quindi il periodo del clock di sistema, in questa configurazione hardware l'oscillatore primario del MCU è connesso ad un quarzo ad 8MHz, dopodiché grazie al PLL interno e alle varie configurazioni dei prescaller, alla CPU e alle periferiche viene distribuito un clock a 60MHz, avremo quindi un periodo di clock pari a:

$$T_{clock} = \frac{1}{F_{clock}} = \frac{1}{60*10^{6}} = 16,\bar{6}\,nS$$

Avendo quindi utilizzato la sorgente interna quale base dei tempi per il contatore e sapendo quanto tempo trascorrerà tra un incremento e il successivo va ora stabilito il numero totale di conteggi che il contatore dovrà accumulare.

$$n_{conteggio} = \frac{T_{agg.PWM}}{T_{clock}} = \frac{20*10^{-6}}{16.\bar{6}*10^{-9}} = 1200$$

In questo caso, lavorando ad una frequenza di 60MHz il numero di conteggi è pari a 1200, questo numero va precaricato nel registro **PRx** *(Fig.23),* ed abilitando l'interrupt relativo al timer in uso si otterrà esattamente una routine che interviene ogni 20 nS.
In questo lavoro di tesi si è fatto uso simultaneo di due timer, entrambe i timer sono stati sincronizzati dalla stessa base dei tempi, in particolare si è usato il Timer2 programmato in abbinamento al modulo OC *(output compare)* per la generazione del PWM, ed il Timer4, come appena descritto, è stato programmato in modo da intervenire ogni 20nS per modificare il duty-cycle del PWM secondo valori prestabiliti dalla mappatura della forma d'onda di riferimento, per tutti i timer presenti nel microcontrollore il procedimento di programmazione è il medesimo.

Modulo OC (PWM)

Come è stato già descritto nell'introduzione e nel paragrafo relativo agli Interrupt, per il funzionamento dell'interfaccia impedenziometrica è necessario fornirgli un segnale di riferimento PWM che con l'evolversi nel tempo del suo valore medio descriva un segnale di tipo sinusoidale alla frequenza di 1KHz. In questo paragrafo studieremo come è possibile utilizzare il modulo OC (Output Compare) per generare un segnale PWM in maniera indipendente dal Software.
Il modulo OC viene utilizzato per generare impulsi per periodi di tempo ben precisi, in questo caso verrà utilizzato per generare una serie continua di impulsi a duty cycle variabile.

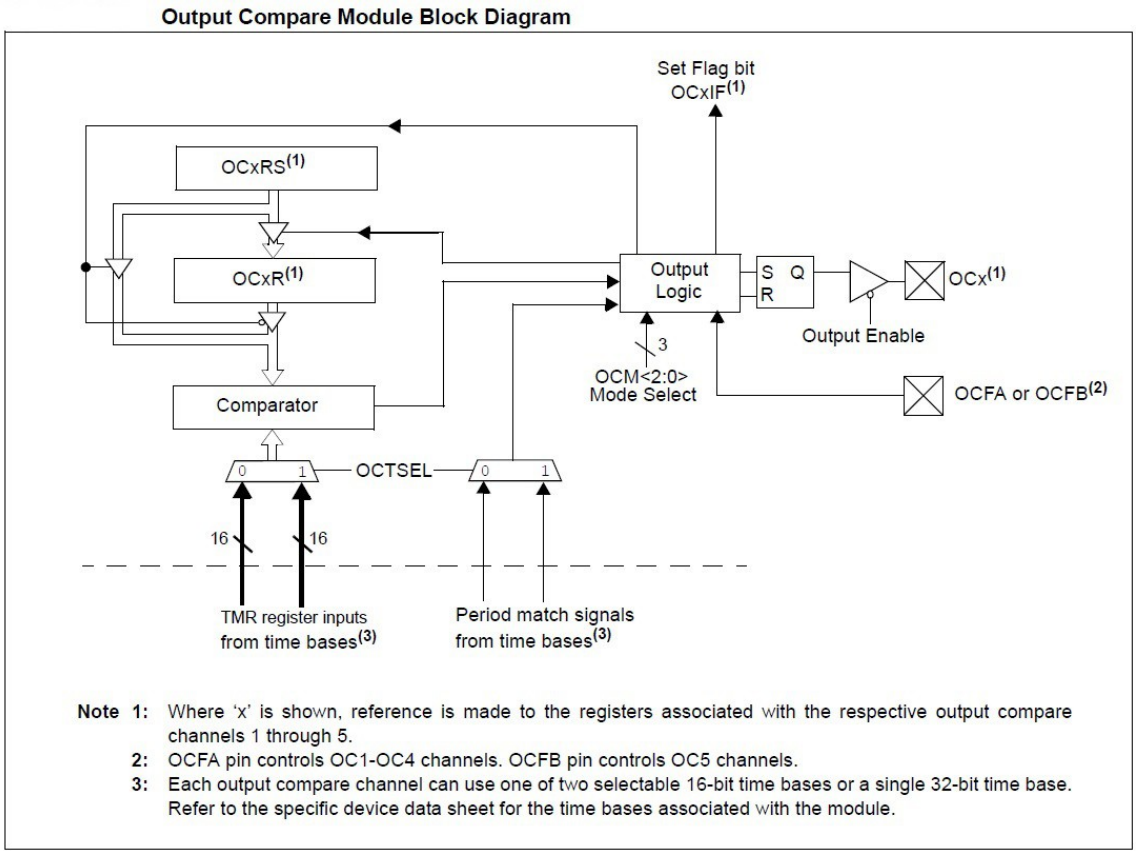


Fig.24 schema a blocchi modulo OC

Come prima cosa occorre inizializzare il modulo OC selezionando la modalità PWM, questo lo si fa agendo sul registro di configurazione OCxCON e impostano i 3 bit di controllo OCM <2:0> a '110' , ovvero si impone il funzionamento in modalità PWM senza il *fault input pin*, che non serve per i nostri scopi. Il passo successivo è quello di individuare quale timer utilizzare come base dei tempi per il segnale PWM, in questo caso il Timer2. Ricordando il motivo per il quale si è utilizzato questo modulo, occorre ora programmare il timer2 per impostare la frequenza del segnale PWM, in questo caso essendo la sinusoide da rappresentare alla frequenza di 1Khz si è scelto di avere una frequenza di PWM di almeno due ordini di grandezza maggiore, ovvero 100Khz.
Ricordando quanto descritto nel paragrafo relativo ai timer occorre calcolare il valore da inserire nel registro PR2 che rappresenta il numero di conteggi necessari per attendete un tempo pari al periodo della frequenza di nostro interesse:

$$n_{conteggio} = \frac{T_{freq\,PWM}}{T_{clock}} = \frac{10*10^{-6}}{16.\bar{6}*10^{-9}} = 600$$

Otteniamo quindi 600 cicli di clock per ogni periodo PWM, a questo punto osservando la *Fig.24* possiamo anche capire quale sia la massima risoluzione del segnale PWM così generato, infatti il registro OCxR, che rappresenta con il proprio valore il numero di cicli di clock per il quale il segnale PWM rimane a livello alto, viene comparato al registro del Timer che per come è stato programmato raggiunge al massimo un valore di 600 conteggi, possiamo quindi ottenere al massimo 600 differenti valori di duty-cycle. Questo ragionamento da anche modo di capire come aumentare la risoluzione, infatti dimezzando ad esempio la frequenza del PWM raddoppiano i possibili valori di duty-cycle.

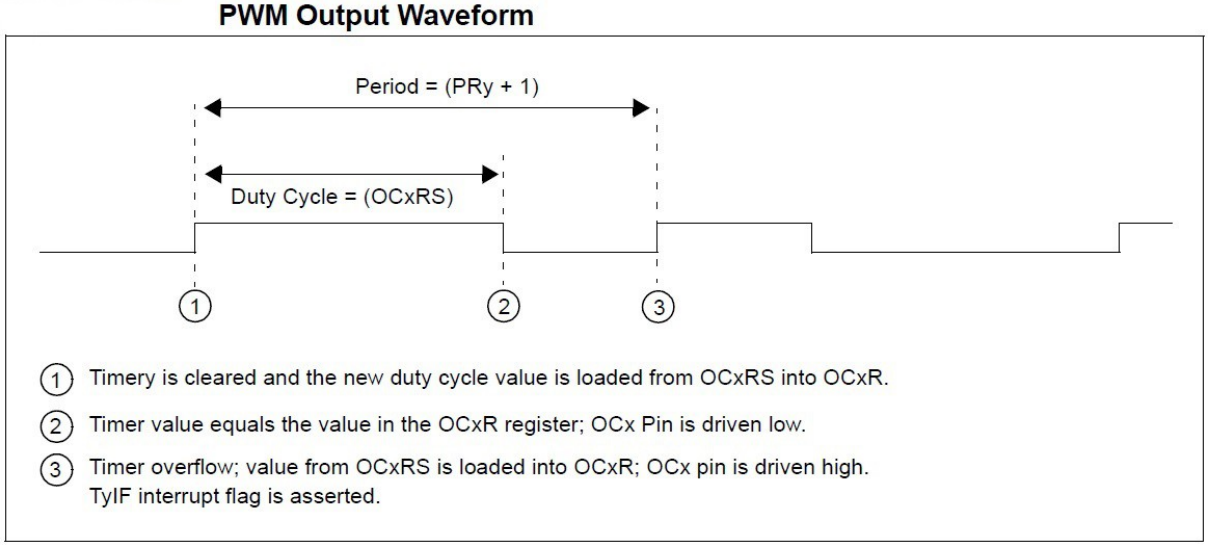


Fig.25 Forma d'onda PWM

Nella modalità PWM i registri OCxR e OCxRS lavorano in modalità master/slave, ovvero una volta attivato il modulo OC attraverso il registro OCxCON è possibile modificare il duty-cycle scrivendo solo il registro OCxRS (slave), il registro OCxR (master) viene aggiornato ricopiando il valore presente nel registro slave esclusivamente all'inizio di un nuovo periodo, questo a garantire che non avvengano *'glitch'* per un repentino cambio di duty-cycle e per lasciare al programma un intero periodo a disposizione per modificarne il valore *(Fig.25)*.

Anche per questo modulo si è fatto uso della libreria fornita da Microchip, la quale consente di inizializzare in maniera sintattica e intuitiva il modulo OC andando successivamente a programmare secondo le nostre esigenze il registro OCxCON.

Come precedentemente enunciato nel paragrafo relativo ai Timer si è fatto uso del Timer2 per la generazione del periodo PWM e del modulo OC1 per la gestione del duty-cycle, riportiamo di seguito la funzione scritta per l'inizializzazione del sistema di generazione del segnale PWM:

```
/*************************************************************
* Function:        initDA
* Input:           int frequenza del PWM: es.(100000)=100KHz
* Output:          void
* Overview:        inizializzazione modulo PWM,  Timer2
*************************************************************/
void initDA( int samplerate){
// inizializzazione OC1
// OC_ON abilita modulo OC
// OC_TIMER2_SRC selezione come sorgente il Timer2
// OC_PWM_FAULT_PIN_DISABLE non utilizzare il controllo FAULT
OpenOC1( OC_ON | OC_TIMER2_SRC | OC_PWM_FAULT_PIN_DISABLE, 0, 0);

// inizializzazione Timer2
//T2_ON abilita Timer2
//T2_PS_1_1 prescaller 1:1
//T2_SOURCE_INT utilizza sorgente interna, clock di sistema
//CLOK/samplerate conteggi prima di un overflow (600 @ 100KHz)
OpenTimer2( T2_ON | T2_PS_1_1 | T2_SOURCE_INT, 60000000/samplerate);
PR2 = (60000000/samplerate);
mT2IntEnable(0);        //disabilita gli interrupt relativi al Timer2
.
.
.
} // initDA
```

Implementazione della funzione sinusoidale

Come descritto in fase introduttiva vi è la necessità di generare un segnale di tipo PWM che descriva con l'evolvere nel tempo del proprio valore medio una funzione di tipo sinusoidale alla frequenza di 1Khz *(Fig.26)*.

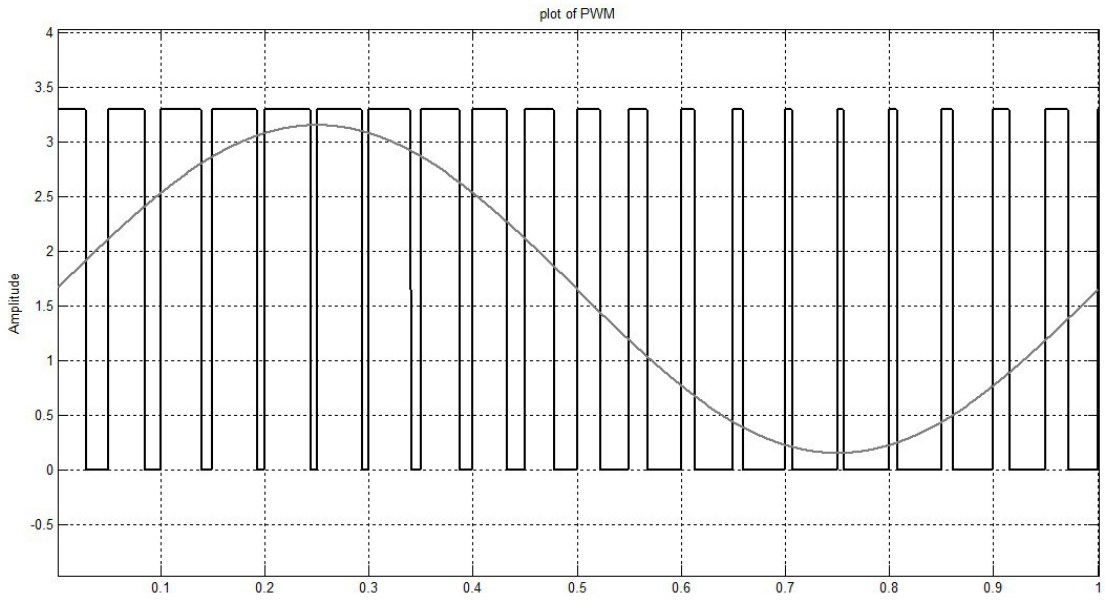


Fig.26 Implementazione di segnale sinusoidale attraverso PWM

Come mostrato in *Fig.26* la rappresentazione del segnale sinusoidale avviene attraverso la variazione controllata del duty-cycle del segnale PWM, si può infatti considerare questa tecnica una vera e propria modulazione, assumendo il segnale sinusoidale come il segnale modulante ed il segnale PWM come portante.
Per esigenze di leggibilità del grafico in *Fig.26* viene mostrato un rapporto 1:20 tra il periodo del segnale PWM e il periodo del segnale sinusoidale, tuttavia come precedentemente enunciato, per avere una maggiore distanza tra le armoniche della modulante e l'armonica della portante si è scelto un rapporto 1:100, ovvero un segnale modulante alla frequenza di 1Khz ed un segnale PWM alla frequenza di 100KHz.

Il problema principale da risolvere quando si utilizza questo tipo di tecnica consiste nell'individuare il criterio con il quale aggiornare valore di duty-cycle in funzione del segnale da rappresentare.

Dal momento che l'obbiettivo è quello di modulare il segnale PWM con una funzione di tipo sinusoidale, si presentano due approcci principali, ovvero una possibilità è quella di calcolare con le funzioni matematiche offerte dalla libreria

ad ogni periodo il valore da assegnare al duty-cycle, oppure quella di mappare una volta per tutte un numero sufficiente di campioni di un periodo del segnale sinusoidale, ed aggiornare il duty-cycle attraverso l'uso di una *look-up table* contenete i suddetti campioni.

Prove sperimentali hanno dimostrato che per quanto possano essere veloci le funzioni trigonometriche offerte dal compilatore C32, non ci sono possibilità di riuscire ad ottenere in tempo utile il valore a virgola mobile prodotto da una funzione del tipo *sin( )* e le successive moltiplicazioni e addizioni necessarie a produrre un nuovo valore di duty-cycle, il Timer4 infatti è programmato in sincronismo con il Timer2 (utilizzato per la generazione del PWM) per generare un interrupt ogni 2 periodi del segnale PWM, ovvero ogni 20µS, tempo assolutamente insufficiente per soddisfare le necessità di calcolo di un valore a virgola mobile. La scelta quindi è ricaduta sull'utilizzo di una look-up table, ovvero di una tabella salvata all'interno della memoria programma, precompilata con i valori che dovrà assumere il duty-cycle all'interno di un periodo della funzione sinusoidale, questo con lo scopo di rendere necessario solo il minimo numero di calcoli possibile all'interno della ISR del Timer4.

Il numero massimo di possibili valori assumibili dal duty-cycle, è superiormente limitato oltre che dal valore massimo assumibile dal Timer2 (legato al clock di sistema), anche dal rapporto tra la frequenza del segnale PWM e la frequenza del segnale modulante che vogliamo utilizzare. In questo specifico caso essendo il rapporto tra le 2 frequenze 1:100 si avrebbero teoricamente a disposizione 100 possibili campioni per ogni periodo della modulante, tuttavia si è scelto di dimezzare il numero dei campioni a favore della disponibilità di risorse della CPU, intervenendo così meno frequentemente con la ISR dedicata all'aggiornamento del duty-cycle, da questa idea nasce infatti anche la scelta di far intervenite il Timer4 ogni 2 cicli del Timer2.

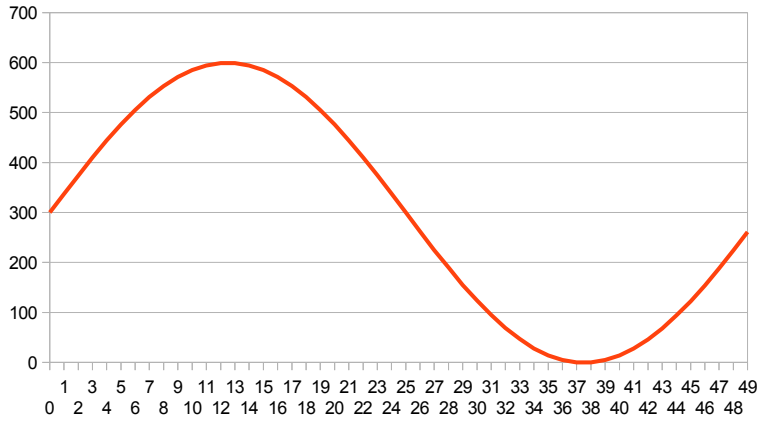
Per ottenere i valori da inserire nella tabella si è utilizzato un foglio di calcolo attraverso il quale è stata computata la seguente funzione:

```
n = offset + INT( amplitude * SIN(ROW * 2π / SAMPLE))
```

dove `SAMPLE` è il numero di campioni, `amplitude` è l'ampiezza che si desidera raggiungere con la sinusoide e `offset` è la costante additiva necessaria per non avere valori di *n* negativi, mentre la funzione `INT` restituisce solo la parte intera del valore tra parentesi. Non essendoci vincoli sull'ampiezza del segnale sinusoidale si è scelto di sfruttare tutta la dinamica del duty-cycle. Facendo riferimento ai paragrafi sul Timer e sul modulo OC ricordiamo che il Timer2 è stato programmato per generare un segnale alla frequenza di 100KHz, ed essendo il clock di sistema pari a 60Mhz, il valore di OCxRS potrà assumere valori compresi tra 0 e 600, questi valori corrispondono ai due estremi assumibili dalla modulazione ovvero un duty-cycle rispettivamente compreso tra 0% e 100%.

La formula così composta restituisce attraverso il foglio di calcolo i 50 valori da inserire nella look-up table:

0	300
1	337
2	374
3	410
4	444
5	476
6	505
7	531
8	553
9	571
10	585
11	594
12	599
13	599
14	594
15	585
16	571
17	553
18	531
19	505
20	476
21	444
22	410
23	375
24	338
25	300
26	262
27	225
28	190
29	155
30	124
31	95
32	69
33	47
34	28
35	14
36	5
37	0
38	0
39	5
40	14
41	28
42	46
43	68
44	94
45	122
46	154
47	188
48	224
49	261



$$n(i) = 300 + \mathrm{INT}(300 \cdot \sin(i \cdot 2\pi / 50))$$

Fig.27 Generazione dei valori per la look-up table

In *Fig.27* possiamo vedere nella prima colonna l'indice progressivo per l'individuazione del valore corrente e nella seconda colonna il valore che dovrà assumere il registro OCxRS. Sarà quindi necessario prevedere l'uso di una variabile di conteggio che indichi con il proprio valore la posizione corrente nella tabella e ad ogni interruzione generata dal Timer4 possa incrementare il proprio valore indicando il nuovo numero da caricare nel registro OCxRS, sarà necessario inoltre un controllo che si occupi una volta raggiunta la fine della look-up table di riposizionare l'indice alla posizione iniziale per la produzione di un nuovo periodo della sinusoide. Mostreremo ora come è stata implementata nel codice la look-up table e la ISR responsabile dell'aggiornamento del registro OcxRS.

Look-up Table:

```
/*********look-up table per la sinusoide*********************************/
const short Table[50]={300, 337, 374, 410, 444, 476, 505, 531, 553, 571,
                       585, 594, 599, 599, 594, 585, 571, 553, 531, 505,
                       476, 444, 410, 375, 338, 300, 262, 225, 190, 155,
                       124, 95, 69, 47, 28, 14, 5, 0, 0, 5, 14, 28, 46,
                       68, 94, 122, 154, 188, 224, 261};
```

ISR:

```
/*****************************************************************
* Function:        T3Interrupt
* Input:           None
* Output:          None
* Overview:        interruzzione sull' overflow del timer4
*****************************************************************/
void __ISR( _TIMER_4_VECTOR, ipl4) T4Interrupt(void){
      OC1RS = Table[count++];       //assegna il nuovo valore al modulo OC
      if(count>=50){                //controllo ripristino dell'indice
      count = 0;
      }
      // azzera interrupt flag
      mT4ClearIntFlag();
} // T4 Interrupt
```

Conclusioni

Tutte le funzioni descritte in questo elaborato sono state testate sperimentalmente sull'Hardware adibito a test *Fig.28* e verificate mediate l'utilizzo di strumenti di misura come oscilloscopio digitale e multimetro.

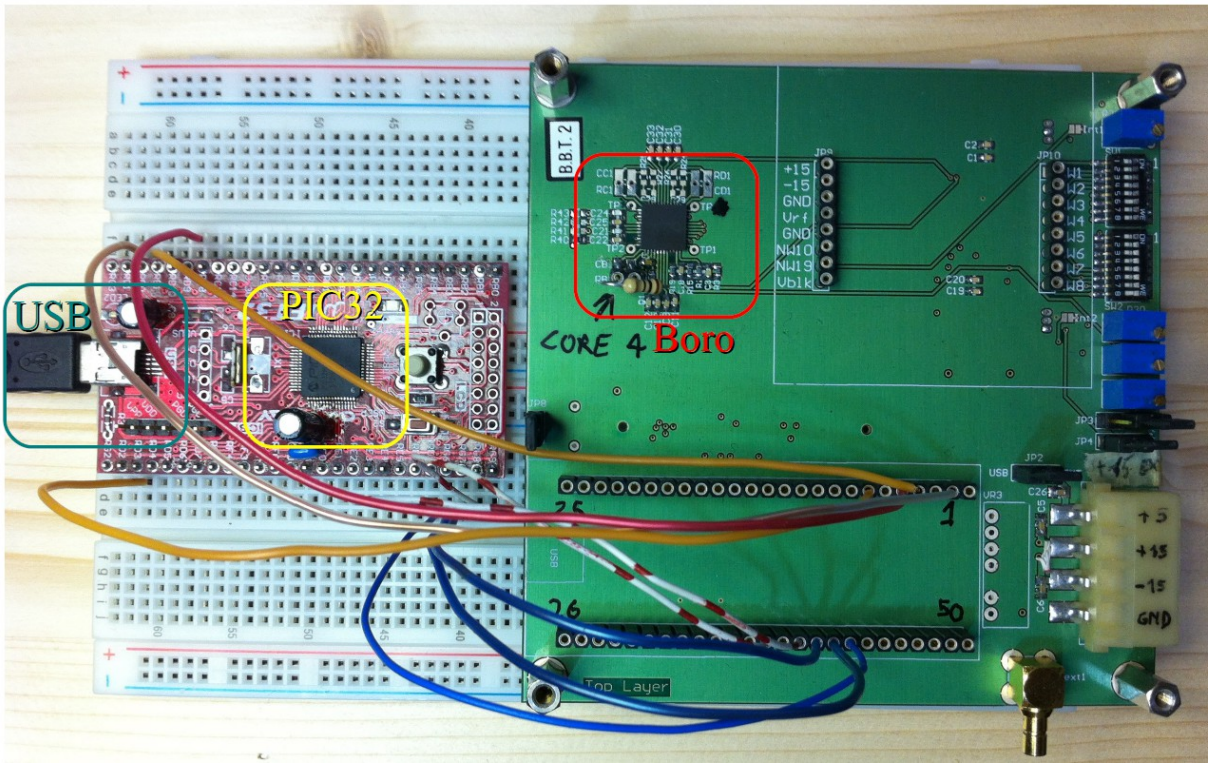


Fig.28 Hardware utilizzato per i test

In particolare è stato testato il funzionamento di:

- Comunicazione e corretta programmazione dell'interfaccia impedenziometrica attraverso bus SPI;
- Generazione segnale PWM di riferimento e verifica dell'effettiva rappresentazione del segnale sinusoidale alla frequenza di 1Khz;
- Acquisizione sincronizzata dei segnali modulati sigma delta;
- Filtraggio di tipo bassa-basso attraverso l'algoritmo Sinc^3;
- Verifica della corretta formattazione secondo protocollo della stringa trasmessa attraverso la seriale emulata;
- Implementazione della classe CDC su MCU e corretto rilevamento da parte del PC;

Il sistema così realizzato è in grado di trasferire al PC attraverso una connessione di tipo USB, le letture impedenziometriche effettuate su 4 sensori distinti, realizzando così misure con una risoluzione di 2^{16} Bit.

In Fig.29 vengono mostrate alcune letture effettuate con il chip BORO:

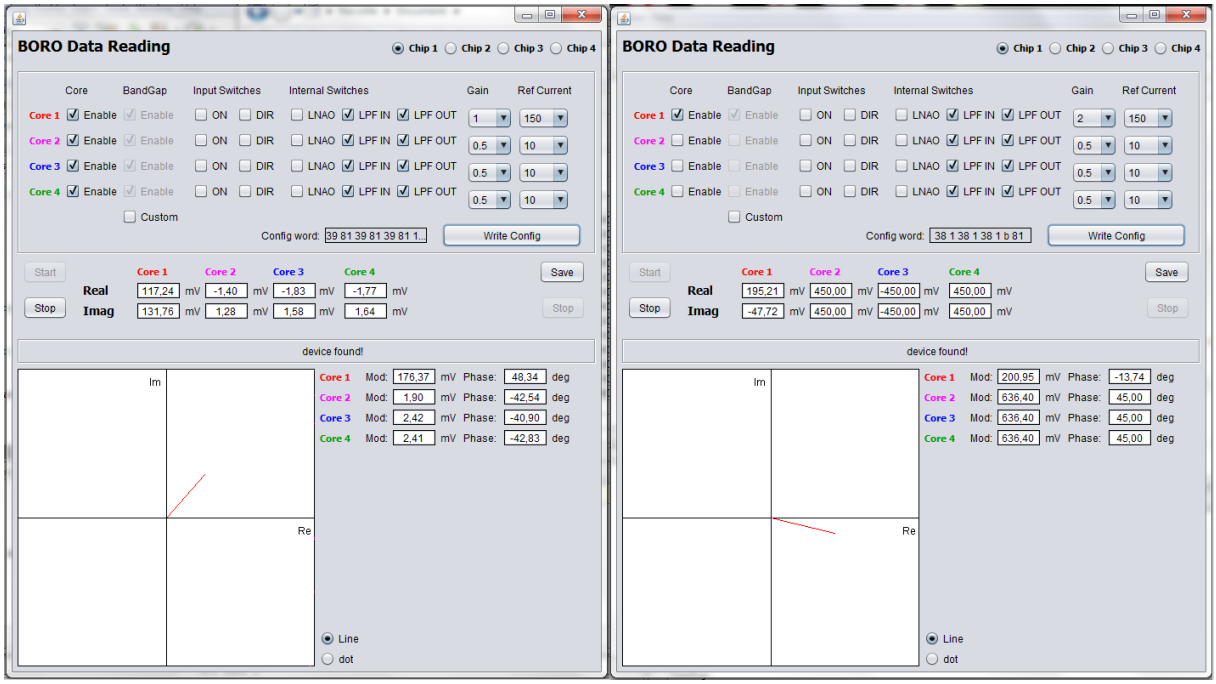


Fig.29 misure effettuate da PC tramite chip BORO

Sviluppi futuri

Questo lavoro di tesi ha verificato la realizzabilità di un sistema di misura impedenziometrico ad elevata risoluzione basato su tecnologia miniaturizzata, in particolare utilizzando lo schema mostrato al termine del paragrafo corrente è possibile alloggiare tutto il sistema su un unico PCB di dimensioni molto ridotte. Equipaggiando il circuito con 4 diversi sensori in grado di effettuare altrettante misurazioni fisiche è possibile infine ottenere un sistema di misura completo particolarmente adatto per applicazioni dove è richiesto un limitato ingombro e ristretti consumi di energia.

Una rappresentazione tridimensionale del sistema miniaturizzato alloggiato su un unico PCB è mostrato il *Fig.30* e in *Fig.31*:

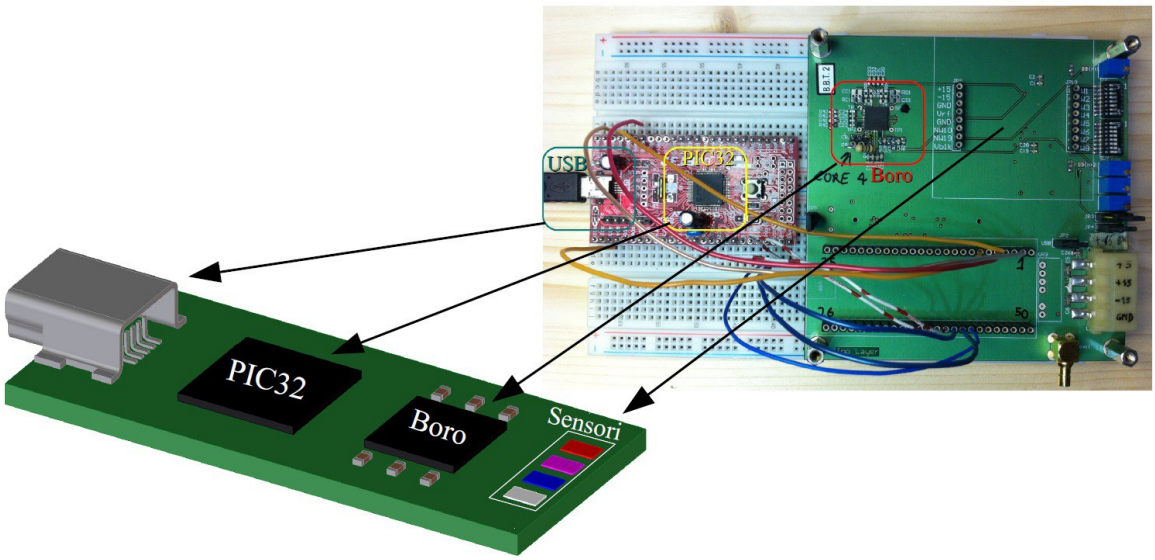


Fig.30 Render sistema completo

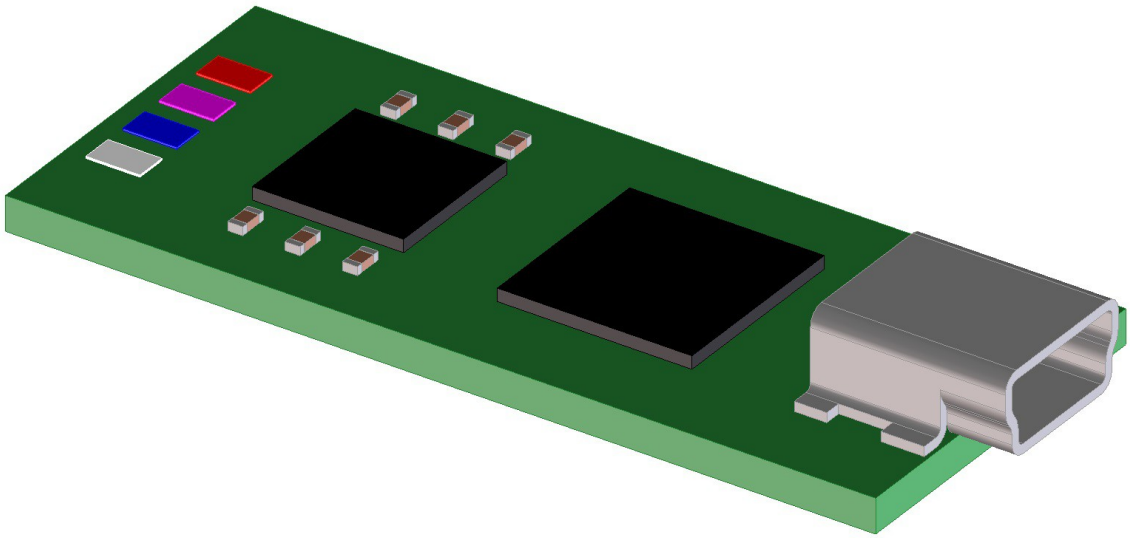


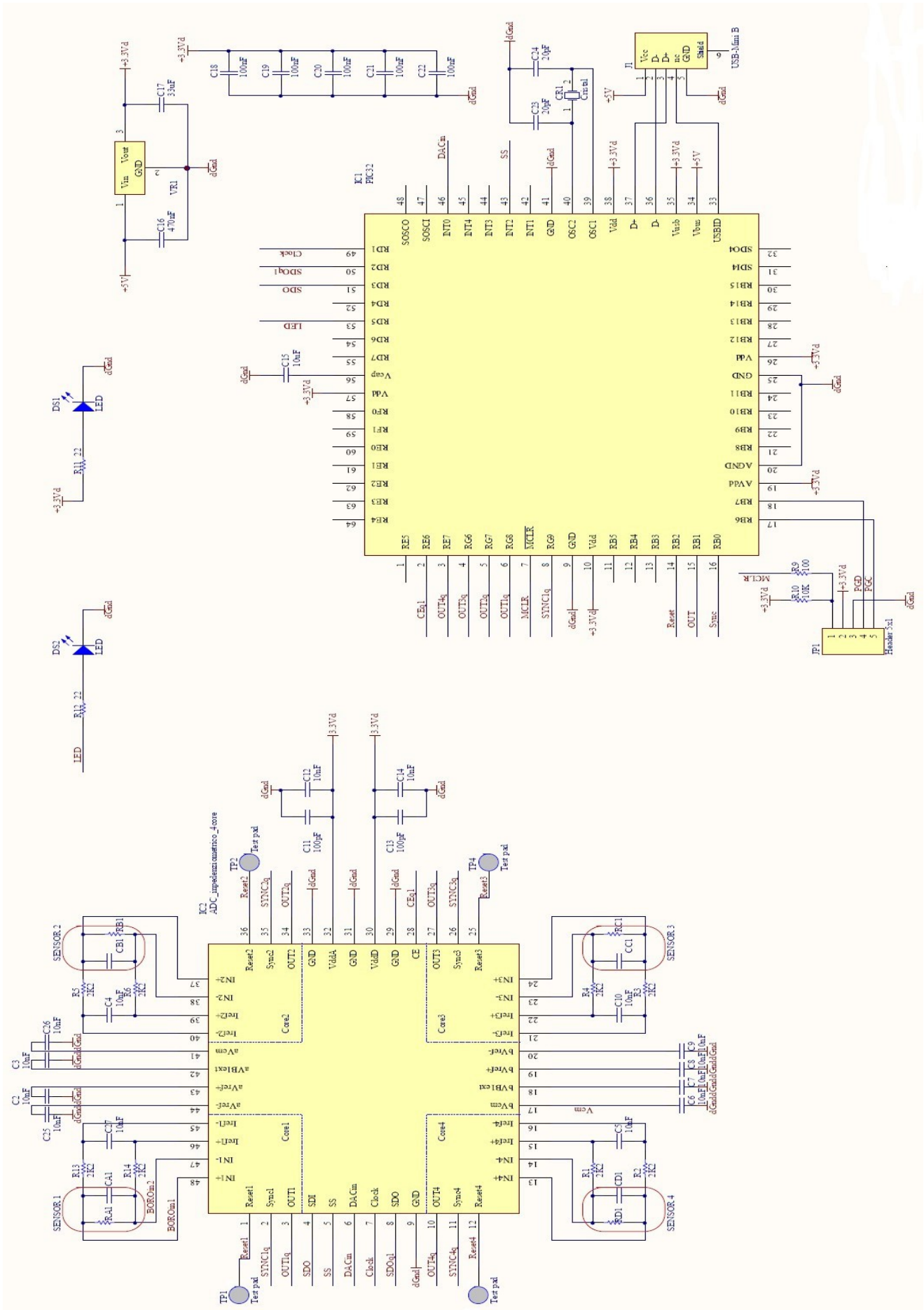
Fig.31 Render sistema completo

Possibili sviluppi sono sicuramente legati anche al software, in particolare sono implementabili diverse funzioni, quali:

- Funzione di modalità *"Sleep"* , è possibile ottimizzare i consumi del dispositivo introducendo una funzioni che risvegli il processore solo quando il Software lato PC richiede una misurazione, questa funzione è assolutamente auspicabile nel caso in cui il sistema venga alimentato a batteria;
- Implementazione della possibilità di scegliere la frequenza di riferimento prodotta dal segnale PWM attraverso le indicazioni fornite dal secondo byte della config word prodotta dal PC, questo garantisce una maggiore versatilità del sistema nell'adattarsi a sensori che presentano impedenze molto differenti tra loro;
- implementazione della possibilità di definire una fase iniziale del segnale sinusoidale di riferimento sfruttando l'ultimo byte della config-word prodotta dal PC, questo permette di azzerare la fase iniziale che nel caso più generale è funzione dell'istante di accensione.
- Introdurre la possibilità di rendere il sistema periodicamente o continuativamente di tipo stand-alone, ovvero implementare la possibilità di effettuare misurazioni indipendentemente dalla connessione al PC, memorizzando le misurazioni fatte nella memoria RAM per una successiva trasmissione o nella memoria Flash presente all'interno del MCU funzionando cosi da vero e proprio *data logger.*

In ultima analisi possiamo notare che il codice scritto in questo lavoro di tesi occupa circa il 5% della memoria programma e il 1,3% della memoria RAM, lasciando cosi ampio spazio per revisioni future del codice oppure alla migrazione verso un dispositivo di fascia più bassa e di profilo energetico di tipo *Low-Power.*

Schema elettrico



Riferimenti bibliografici

- *Microchip, PIC32MX795F512H Family Data Sheet, www.microchip.com*
- *Microchip, PIC32 Family Reference Manual, Sect. 08 Interrupts, www.microchip.com*
- *Microchip, PIC32 Family Reference Manual, Sect. 06 Oscillators, www.microchip.com*
- *Microchip, PIC32 Family Reference Manual, Sect. 14 Timers, www.microchip.com*
- *Microchip, PIC32 Family Reference Manual, Sect. 16 Output Compare , www.microchip.com*
- *Microchip, PIC32 Family Reference Manual, Sect. 23 Serial Peripheral Interface , www.microchip.com*
- *Microchip, PIC32 Family Reference Manual, Sect. 27 USB On-The-Go , www.microchip.com*
- *Gaetano Iuculano, Domenico Mirri, Misure Elettroniche, Cedam, 2004*
- *Mauro Laurenti, Tutorial - L'interfaccia SPI, www.laurtec.com*
- *Texas instrument Application Report SBAA094 – June 2003*
- *Boro datasheet 48 pin v4 - UNIBO*