ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA IN INGEGNERIA ELETTRONICA,
INFORMATICA E TELECOMUNICAZIONI

Titolo dell'elaborato

SVILUPPO DI UN FIRMWARE PER UN SISTEMA DI SENSING IMPEDENZIOMETRICO A BASSA POTENZA

Tesi di laurea in
ELETTRONICA DEI SISTEMI DIGITALI

RELATORE
Prof. Aldo Romani

CANDIDATO
Tommaso Quadrelli

CORRELATORI
Dott. Marco Crescentini
Dott. Giulia Luciani

Sessione Febbraio/Marzo
Anno Accademico 2015/2016

Sommario

Introduzione

Il progetto vuole realizzare un sistema ultra low power, in grado di monitorare variabili fisiche quali temperatura e conducibilità dell'acqua nelle profondità marine in autonomia, per una durata complessiva di due anni. Il salvataggio dei dati raccolti nel periodo di utilizzo avrà come fine ultimo lo studio dei cambiamenti climatici relativi all'ambiente marino. Volendo collocare il sistema di monitoraggio sul dorso di pesci o in profondità oceaniche non facilmente accessibili è necessario garantire dimensioni ridotte e un funzionamento autonomo duraturo al termine del quale sarà possibile scaricare i dati raccolti. Nel tentativo di rispettare la specifica relativa al ciclo di lavoro autonomo del sistema è stato importante adottare una politica rigorosa riguardante i consumi estremamente ridotti, senza però venir meno alle ulteriori specifiche di progetto, riportate in dettaglio nei paragrafi successivi.

Dalla progettazione circuitale alla realizzazione del firmware, passando per una minuziosa scelta della componentistica a minor consumo, ho avuto la possibilità di dar vita all'intero progetto in autonomia anche grazie al supporto tecnico, nonché morale, del dottor. Marco Crescentini, confrontandomi con tutti gli aspetti e le problematiche che la realizzazione di un simile progetto porta con se.

Il software realizzato prende spunto da una tesi di laurea [1] nella quale si è sviluppato un sistema di acquisizione dati ad alta risoluzione da un'interfaccia impedenziometrica. Nell'elaborato il processo di acquisizione avveniva in maniera continuativa fino al raggiungimento di un numero prestabilito di campioni i quali venivano filtrati real-time ed inviati al computer tramite un collegamento USB. Il flusso operativo di tale tesi è risultato fonte di ispirazione per la realizzazione di un firmware volto al campionamento, al filtraggio ed al salvataggio dei dati raccolti provenienti da interfacce di sensing integrate. Il software realizzato nel precedente lavoro di tesi si differenzia dal seguente elaborato in quanto per la natura del sistema che vogliamo realizzare abbiamo necessità di acquisire i valori ad intervalli regolari di tempo,

non in maniera continuativa, effettuare un filtraggio dei campioni ad acquisizione completata, non durante tale fase, ed infine non inviare i dati al PC tramite USB ma salvarli in memoria.

Obiettivo

Il progetto nasce con lo scopo di realizzare un sistema di monitoraggio ad alta risoluzione completamente autonomo e con un ciclo di vita di due anni finalizzato allo studio delle variabili ambientali dell'habitat marino. Lo scopo di questa tesi è quello di realizzare un prototipo del sistema finale, il quale permetta di studiare e testare il flusso di operazioni, avendo la possibilità di agire con correzioni o miglioramenti sul firmware e sullo schema elettrico prima della realizzazione del circuito finale. Per queste ragioni nella scheda di test realizzata sono presenti numerosi test-pad, jumper e switch: per permettere un monitoraggio più agevole sui segnali, per avere la possibilità di rilevare gli effettivi consumi e per poter effettuare un corretto debug avendo la possibilità di controllare selettivamente le varie parti del circuito.

Il firmware ha il compito di provvedere all'acquisizione dei dati dalle interfacce di sensing, eseguire un filtraggio dei suddetti rilevamenti ed infine salvare le misurazioni su una memoria esterna. Al fine di limitare il consumo dell'intero processo il firmware deve gestire l'abilitazione delle interfacce relativamente al loro periodo di utilizzo, provvedendo inoltre a ridurre i consumi nella fase di attesa tra i vari rilevamenti utilizzando una modalità di funzionamento a risparmio energetico.

In Figura 1 viene mostrato lo schema a blocchi del circuito in questione: il sensore fornisce all'interfaccia impedenziometrica uno strumento per la lettura dei valori relativi alle grandezze fisiche rilevate i quali vengono poi campionati dalla stessa interfaccia. Due blocchi di commutatori permettono la mutua selezione del dispositivo di controllo da utilizzare per elaborare i campioni: PIC18F46K80 della Microchip (del quale si è sviluppato il firmware in questa tesi) o un modulo FPGA. L'unità di controllo utilizzata, tra le due disponibili, è incaricata di gestire la routine di operazioni riguardante il salvataggio dei campioni prodotti dall'interfaccia impedenziometrica, di effettuarne il filtraggio e di scriverne in memoria i relativi dati ottenuti. Questo flusso di istruzioni non dev'essere eseguito continuamente ma periodicamente nel tempo. Il calcolo del periodo tra un ciclo e quello successivo è spiegato e ricavato in dettaglio nei capitoli successivi.

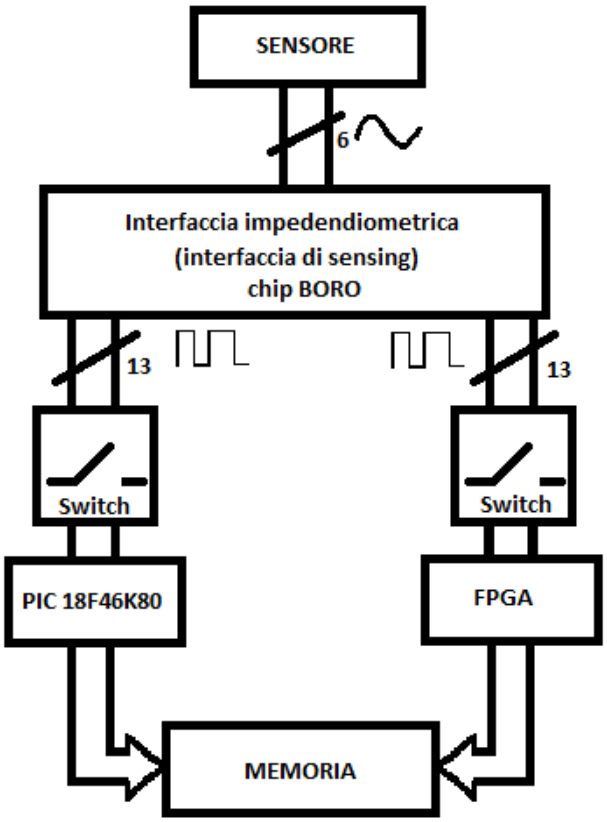


Figura 1 - Schema a blocchi circuito

Introduzione al chip impedenziometrico

Nella realizzazione di questo progetto si è fatto uso di un interfaccia impedenziometrica a 4 canali implementata su un chip realizzato dal gruppo di ricerca del Prof. Marco Tartagni [2]. Al chip è stato associato dagli stessi sviluppatori il nome in codice “BORO”, nella seguente tesi si farà spesso sua menzione utilizzando tale pseudonimo.

Si tratta di un integrato ad alta precisione e a basso consumo, composto da 4 interfacce d’impedenza gestite da altrettanti core. L’interfaccia è basata su un approccio totalmente digitale basato su una demodulazione di tipo ΔΣ in grado di raggiungere i 15 bit di risoluzione. Il tutto è implementato in una tecnologia CMOS a 0.35µm e occupante un’area di $9mm^2$ [2].

In Figura 2 si può notare in giallo lo schema a blocchi costituente un singolo core del chip, in verde la struttura di controllo (in figura vi è l'FPGA, sostituibile con un microcontrollore). Lo studio di questa figura è utile per prendere conoscenza di quali siano i segnali di comunicazione tra Boro e l'unità di elaborazione. Sono presenti, in ordine a partire da sinistra, il segnale necessario al chip per effettuare la lettura dell'impedenza, i segnali di controllo basati su comunicazione SPI che permettono una corretta programmazione del core, il clock necessario a generare la fase digitale ed infine il segnale di sincronismo SYNC relativo al segnale OUT sul quale viene trasmesso il risultato del campionamento. Il chip è stato progettato per ricevere in ingresso un pattern di valori generato tramite il software Matlab simulante un segnale modulato ΔΣ. Il segnale viene quindi convertito in un bit-stream in corrente tramite il convertitore V/I e filtrato attraverso un filtro passa basso in corrente al fine di eliminare gli errori causati dalla generazione del ΔΣ. Il successivo elemento è l'amplificatore LNA a guadagno variabile (per una maggiore flessibilità d'utilizzo) il quale effettua una lettura di tipo differenziale del DUT (valore d'impedenza da misurare), segue poi un filtro capacitivo che elimina il rumore. Il blocco BPDS non è altro che un convertitore ΔΣ passa banda il quale effettua il campionamento del segnale analogico di tensione in uscita dall'amplificatore ad una frequenza 4 volte maggiore (quindi 4 campioni ogni periodo) e provvede a sfasare tali campioni di $\pi/2$ gli uni dagli altri. Si ottengono così due segnali in quadratura di fase i quali vengono demodulati e riportati in banda base. I relativi bit-stream ottenuti vengono impacchettati in un unico segnale e portati all'esterno sul pin OUT. Il blocco BGR (Band Gap References) ha il compito di generare i livelli di tensione, ottenuti da valori fisici indipendenti dalla temperatura, utilizzati per i limiti di soglia nell'ADC, nell'LNA e per generare il riferimento in tensione utilizzato in tutto il chip. La sommaria descrizione relativa al funzionamento del chip Boro è utile per acquisire una maggiore conoscenza dei segnali ne-

cessari alla sua programmazione e per comprendere la natura dei segnali generati e portati in uscita.

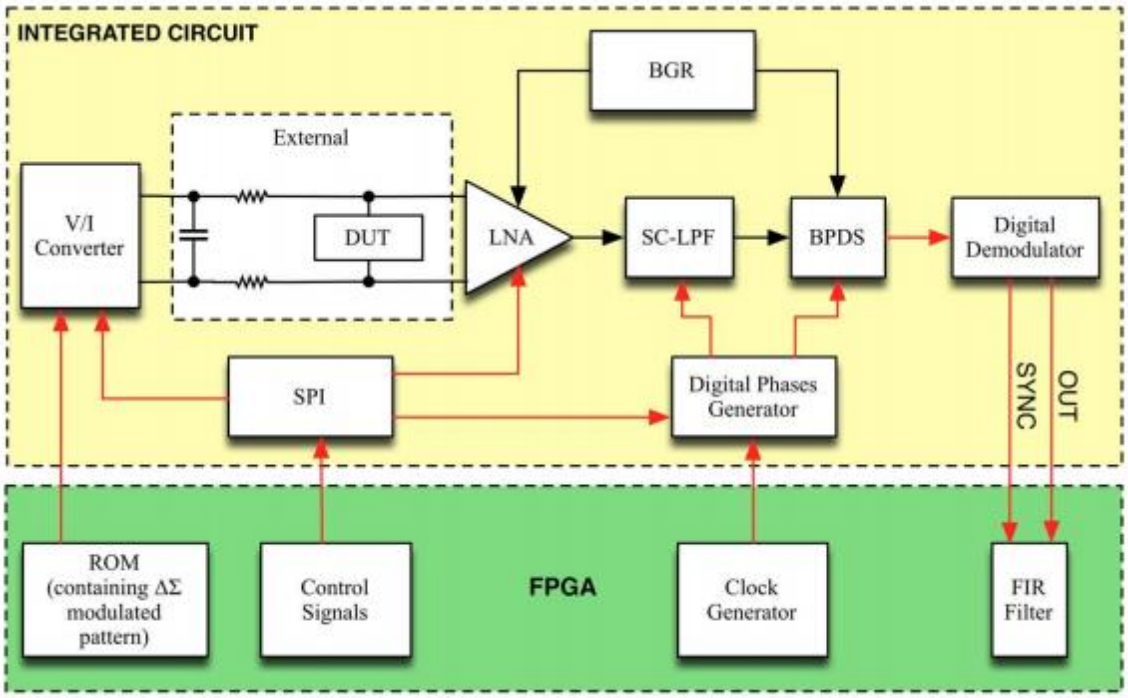


Figura 2 - Schema a blocchi chip BORO da [2]

La sua applicazione abbinata a sensori CTD (Conductivity-Temperature-Depth), sensori in grado di rilevare simultaneamente conduttività/temperatura/profondità del liquido all'interno del quale sono immersi, costituisce un'ottima soluzione applicativa per il monitoraggio di riserve marine.

Specifiche di progetto

Le principali specifiche di progetto riguardano essenzialmente i consumi e le dimensioni:

- Dev'essere garantito un consumo estremamente ridotto, che permetta un funzionamento continuativo per due anni in mare aperto utilizzando come fonte di alimentazione una batteria. Il consumo del chip con tutti e 4 i core in funzione risulta di 5,8 mW [2], è pertanto necessario che il consumo apportato dal sistema di controllo non sia rilevante rispetto tale valore. Le dimensioni devono inoltre essere minime

ed il costo dell'intero sistema dev'essere basso, questi vincoli ci portano ad orientarci verso batterie a bottone con capacità comprese tra i 1000mAh e i 1500mAh.

- L'elaborato finale non deve superare le dimensioni di 60x30 mm (ingombro stimato a partire dalle dimensioni dei componenti minimi necessari usando package adeguati) al fine di assicurare un applicazione non invasiva del circuito sul dorso di pesci o su boe.
- Al fine di misurare le più piccole variazioni e i minimi cambiamenti con estrema precisione è necessaria un'elevata risoluzione: grazie all'approccio totalmente digitale basato su una demodulazione ΔΣ del chip Boro ci è permesso raggiungere 15 bit di risoluzione [2].

La scelta della componentistica si basa sul rispetto di queste prioritarie specifiche le quali ci portano a raggiungere dei compromessi legati alla natura fisica dei singoli componenti.

Alimentazione

Al fine di assicurare un funzionamento di due anni non è conveniente effettuare i campionamenti continuamente ma in maniera periodica in modo da ridurre al minimo i consumi. Questa scelta progettuale è realizzabile in quanto i parametrici fisici monitorati sono caratterizzati da costanti di tempo tipicamente elevate, perciò i cambiamenti dei valori avvengono lentamente. Il monitoraggio continuo non risulterebbe efficiente in quanto: si otterrebbero molte misurazioni simili, provocherebbe un consumo maggiore ed una saturazione della memoria più rapida. La scelta del periodo tra un rilevamento e quello successivo è risultato di un calcolo le cui variabili sono la capacità di alimentazione utilizzata e il consumo dell'intero circuito. Si ottiene quindi un periodo di tempo ponderato tra: la necessità di effettuare letture ravvicinate per rilevare ogni minimo cambiamento di stato e la necessità di ridurre i consumi non eccedendo nella quantità di dati raccolti. A seguito

di questo studio si raggiunge quindi un compromesso tra la capacità di alimentazione necessaria e il periodo tra i campionamenti.

Effettuando rilevamenti ogni 125 secondi la capacità necessaria al funzionamento biennale dev'essere di almeno 1000 mAh (Figura 3).

Measurement Time	1,00	s
Measurement Current	6	mA
Standby Current	0,01	mA
Battery Capacity	1000	mAh
Deployment Length	24	months
Deployment Length	720	days
Deployment Length	17280	hours
Measurement Period	125,1	s

Figura 3 - Calcolo del periodo dei rilevamenti

Il chip Boro per funzionare necessita di 3,3 V fissi [6], valore di tensione al di sotto del quale non bisogna mai scendere per evitare il malfunzionamento o addirittura lo spegnimento dell'interfaccia.

Fonti di alimentazione che si contraddistinguono per le loro ridotte dimensioni, in grado di offrire capacità e voltaggi simili a quelli richiesti, sono batterie di tipo "bottone", le migliori in base al rapporto prestazioni/dimensioni. La scelta ricade sulla batteria TADIRAN BATTERIES SL-889/P la quale, con i suoi 5 cm^3 di ingombro volumetrico, garantisce una capacità di 1000 mAh e 3,6 V per un periodo di funzionamento a 3 mA di circa 300 ore [3]. Considerando il periodo di funzionamento in fase di misurazione a 6 mA, stimato di 1 s, ed utilizzando un periodo tra le misurazioni di 125 secondi, nella durata complessiva di due anni il circuito sarà in funzione per 497'664 s, per un totale di 5,76 giorni complessivi di funzionamento (circa 138 ore, valore molto inferiore delle 300 ore considerate dal grafico). Nel restante periodo di tempo il sistema si troverà in una fase a risparmio energetico. In Figura 4 viene riportata la curva di scarica della batteria nel tempo.

Non avendo a disposizione dati riguardanti il funzionamento del chip a fronte di variazioni della tensione di alimentazione, non sappiamo se il fisiologico calo della batteria possa provocare variazioni di prestazioni. Per questo motivo, dovendo realizzare la test-board del progetto finale, oltre all'utilizzo della batteria come fonte di alimentazione, permetto di alimentare il circuito con una fonte esterna come un generatore di tensione il quale ci permette di analizzare il comportamento del circuito simulando un calo di tensione della batteria nel tempo.

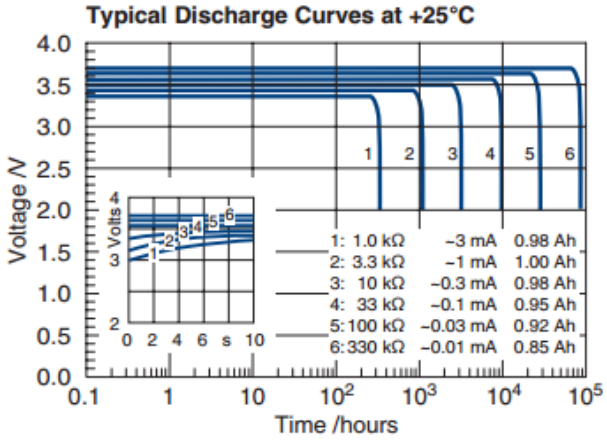


Figura 4 - Grafico Voltage/Time da [3]

Il Sensore

Il sensore utilizzato per i rilevamenti delle variabili marine è l'LFS 117 dell'IST il quale, oltre ad avere dimensioni contenute, grazie ai suoi 6 terminali è in grado di effettuare due differenti tipi di letture: una di temperatura, tramite il sensore di temperatura interno PT1000, ed una di conducibilità del liquido nel quale è immerso. Fattori influenzanti nella scelta di questo sensore sono [4]:

- Ridotte dimensioni: 16.9 x 9.9 x 0.65 mm.
- Basso assorbimento di corrente: 0.3 mA in fase di misura.
- Ampio range di temperature di lavoro garantito.
- Eccellente stabilità a lungo termine.

Technical Data

Operating temperature range:	-50 °C to +150 °C	
Conductivity range:*	0.2 mS/cm to 200 mS/cm	
Cell constant:*	typical 0.435 1/cm at 1.4 mS/cm	
Temperature sensor:*	Pt1000	
Measurement frequency range:	100 Hz bis 3 kHz	
Maximum supply voltage (electrodes):	< 0.7 V_{pp} (Electrolysis of the analyte has to be avoided)	
Characteristics curve:	3850 ppm/K	
Measuring current[3]:	0.3 mA	
3) Selfheating must be considered		
Temperature sensor accuracy (dependent on temperature range):*		IST AG reference
	DIN EN 60751 F0.3	B
	DIN EN 60751 F0.6	C
Connection:*	Pt/Ni wires, Ø 0.2 mm	
	Cu/Ag wires, PTFE insulated, AWG 30	

Figura 5 - Technical Data sensor da [4]

Il Microcontrollore

Per la scelta del microcontrollore è necessario seguire ulteriori specifiche oltre a quelle più generali già menzionate le quali permettano di comunicare con l'interfaccia e con la memoria, di programmare correttamente il chip BORO e di assicurarne un corretto funzionamento. La scelta ricade sul microcontrollore ad 8bit PIC18f46k80 della Microchip il quale è in grado di rispettare e soddisfare le seguenti necessità:

- Il consumo in fase di lavoro dev'essere estremamente ridotto ed essendo il PIC18F46K80 un microcontrollore di tipo Nano Watt Technology permette un *low power consumption*: 300 nA tipici di consumo per il WDT e 13 nA di consumo tipici per la Sleep mode [5].
  Inoltre prevede una modalità di risparmio energetico per limitare al massimo i consumi durante la fase di attesa tra i rilevamenti denominata *Sleep Mode*. Durante questa fase viene reso insensibile il programma alla maggior parte degli interrupt e vengono arrestati tutti i clock all'infuori di un oscillatore interno a basso consumo. Il risveglio dalla *Sleep Mode* può avvenire in diverse modalità, per la realizzazione di questo progetto sfruttiamo la presenza di un *Watchdog Timer* il quale, al termine di un conteggio prefissato, è in grado di "svegliare" il programma facendolo tornare in modalità di lavoro.
- Per la comunicazione e la programmazione del chip e della memoria è necessario che il microcontrollore disponga di un modulo SPI che

permetta di operare come Master e di poter gestire simultaneamente due Slave selezionabili attraverso differenti *Chip Select* (CS/SS); il PIC18F incorpora 4 differenti moduli SPI.

- 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):
  $Z = R + jX$ [ohm]
  Per garantire l'esatta lettura dell'impedenza, oltre alla corretta lettura del modulo, è necessario porre particolare attenzione alla fase del vettore rappresentativo associato, 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 [1]. Inoltre per poter effettuare la lettura dell'impedenza è necessario "stimolarla", tale stimolo è stato previsto come un segnale modulato ΔΣ, riconducibile per semplicità ad un segnale modulato PWM. Questa esigenza viene soddisfatta dalla presenza di un modulo *Capture/Compare/PWM* abbinato all'utilizzo di un timer interno che svolge tale compito.

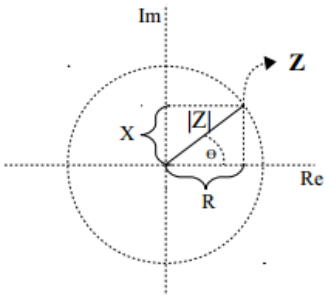


Figura 6 - Impedenza come n complesso da [1]

La Memoria

Per la scelta della capacità della memoria è importante effettuare una stima dei dati che verranno raccolti nei due anni di operatività. L'operazione di filtraggio dei campioni permette di effettuare una media dei rilevamenti ed ottenere, per ogni core dell'interfaccia impedenziometrica, un valore della lunghezza di 16 bit. Ad ogni ciclo di lettura quindi, considerando tutti e 4 i core, verranno prodotti 64 bit di dati da salvare. Conoscendo l'intervallo di tempo tra le letture (125 secondi) ottengo il numero dei cicli che verranno effettuati nei due anni di lavoro: 503'194 letture. Sapendo che ad ogni ciclo vengono prodotti 64 bit è facile ottenere la quantità di dati prodotti e, di conseguenza, la capacità della memoria necessaria al relativo salvataggio.

La quantità di dati che si otterrà sarà pari a 32'292'864 bit, ovvero 4,04 MB, il valore di capacità della memoria dovrà essere pertanto maggiore di tale valore.

Dopo una dettagliata ricerca si è individuata la memoria MEMORY S25FL164K-64Mb, la quale soddisfa le richieste progettuali. Per la scelta di questo componente mi sono orientato su memorie di tipo flash, più veloci e a minor consumo rispetto a tipologie differenti: il consumo di 5,6 mA di tale memoria [8] in fase di scrittura è il valore più basso individuato rispetto a memorie di medesima capacità che presentano invece consumi notevolmente maggiori, come ad esempio la SDRAM IS45S16400F della ISSI [9] che consuma ben 110 mA. La capacità richiesta di 8 MB risulta essere il fattore decisivo per la scelta della memoria: a parità di consumo le memorie disponibili presentano capacità ridotte e non sufficienti per la realizzazione del progetto. Tale memoria unisce infatti la capacità richiesta di 8 MB con le ridotte dimensioni del package e i bassi consumi in fase di scrittura. Inoltre integra un modulo SPI necessario alla comunicazione con il microcontrollore ed una modalità di funzionamento a riposo denominata *Deep Power Down* nella quale i consumi vengono estremamente ridotti.

Il microcontrollore PIC18F46K80 in dettaglio

Per la realizzazione del progetto utilizzo il microcontrollore PIC18F46K80 della Microchip appartenente alla famiglia dei PIC18F66K80, in quanto in grado di rispettare tutte le specifiche progettuali richieste. La sua architettura ad 8 bit non permette un utilizzo fluido del codice sviluppato nella precedente tesi di laurea essendo questo realizzato per un microcontrollore con architettura interna a 32 bit. Ciò comporta una differente gestione interna dei dati e l'impossibilità di utilizzare librerie fornite da Microchip specifiche per microcontrollori a 32bit. E' necessario pertanto adattare il programma alla nostra struttura quando possibile, mentre è obbligatoria una completa riscrittura del codice quando vengono utilizzate funzioni esclusive per strutture a 32 bit.

Per l'utilizzo corretto del PIC sono necessari alcuni componenti esterni di contorno ed inoltre, per facilitare la fase di test e debug, sono stati inseriti nel progetto led, test point e jumper:

- Oscillatore al quarzo da 8 MHz.
- Regolatore di tensione a 3,3 V.
- Condensatori di filtro per le capacità.
- Connettore per la programmazione ICSP.
- Connettore EUSART.
- Led per il controllo dei segnali di Reset dei 4 core di Boro.
- Jumper collegati all'alimentazione per testare l'assorbimento di corrente.

Il microcontrollore scelto per questo progetto presenta varie caratteristiche utili allo sviluppo del codice:

- Ha un'architettura ad 8 bit.
- E' provvisto di un *Extended Watchdog Timer* (WDT) con un periodo programmabile da 4 ms a 4194 ms: ideale per generare un overflow ogni 125 s come previsto.
- Incorpora 4 moduli *Capture/Compare/PWM*: necessari per la generazione del segnale PWM di riferimento per il chip Boro.

- Supporta tutti e 4 i moduli SPI: per il nostro utilizzo è indispensabile la presenza del modulo che permetta di utilizzare il microcontrollore come master e che riesca a comandare più slave.
- Ha 64 kB di *Program Memory* e 3’648 kB di *Data Memory*.
- E’ provvisto di un modulo EUSART per il collegamento al PC tramite un adattatore EUSART-USB.
- Dispone di 3 modalità di funzionamento principali *Primary Run Mode, Idle Run mode, Sleep Mode*: ideali per impostare una configurazione a minor consumo.
- PLL (Phase-Locked Loop) interno in grado di moltiplicare fino a x4 il clock esterno: utile per generare un clock a frequenza maggiore da utilizzare nella fase di scrittura in memoria per velocizzarne il processo e limitarne il periodo di operatività (questa modalità di funzionamento sarà da testare a progetto ultimato effettuando dei monitoraggi di potenza e constatando quale sia il compromesso più efficiente).

Il PIC18F46K80 è disponibile in due configurazioni: il 40-Pin PDIP e il 44-Pin TQFP [5]. Per il progetto scelgo di utilizzare il 40-Pin PDIP per un più facile montaggio sul circuito essendo questa una scheda di test; per il progetto finale invece è preferibile utilizzare la configurazione 44-Pin TQFP il

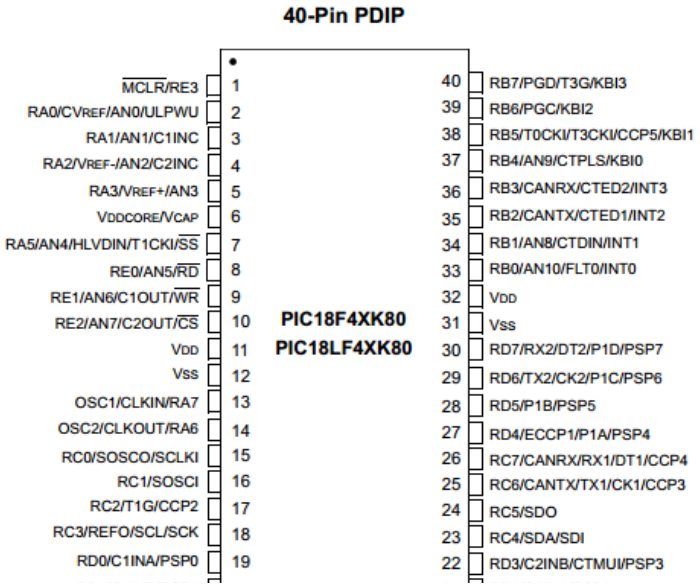


Figura 7 - Piedinatura PIC18F46K80 da [5]

cui ingombro è notevolmente inferiore. In Figura 7 viene mostrata la piedinatura riferita alla configurazione scelta.

In Figura 8 viene riportato l'assorbimento di corrente nella *Primary Run Mode* (*PRI_RUN_MODE*) ad una frequenza di 4 MHz. La prima colonna di valori rappresenta il consumo tipico mentre la seconda quello massimo. Non essendo disponibile il consumo del microcontrollore alla frequenza di lavoro dell'oscillatore esterno utilizzato nel progetto (8 MHz) si è stimato un assorbimento di corrente circa del doppio di quello di seguito raffigurato. Con un'alimentazione di 3,3 V, ad una temperatura campione di 25 °C e con un valore di oscillazione di 8 MHz, il consumo previsto è circa di 1,17 mA.

PIC18FXXK80	566	1020	µA	-40°C	V_{DD} = 3.3V[5] Regulator Enabled
	585	1020	µA	+25°C	
	590	1020	µA	+60°C	
	595	1120	µA	+85°C	
	600	1220	µA	+125°C	

Figura 8 - Consumo Primary Run Mode da [5]

Come si evince dalla Figura 9 il consumo in *Sleep Mode* da parte del microcontrollore, alimentato a 3,3 V e ad una temperatura campione di 25 °C risulta essere solamente di 230 nA.

PIC18FXXK80	200	700	nA	-40°C	V_{DD} = 3.3V (**Sleep** mode) Regulator Enabled
	230	800	nA	+25°C	
	320	1050	nA	+60°C	
	510	1500	nA	+85°C	
	5	9	µA	+125°C	

Figura 9 - Consumo Sleep Mode da [5]

Riporto infine il consumo tipico relativo al funzionamento del Watchdog: 0,6 µA, visibile in Figura 10.

PIC18FXXK80	0.6	3	µA	-40°C to +125°C	V_{DD} = 3.3V Regulator Enabled

Figura 10 - Consumo Watchdog Timer da [5]

Il consumo di riferimento scelto è quello relativo alla temperatura di 25 °C in quanto simile al valore riscontrabile in ambiente marino.

Progetto Hardware

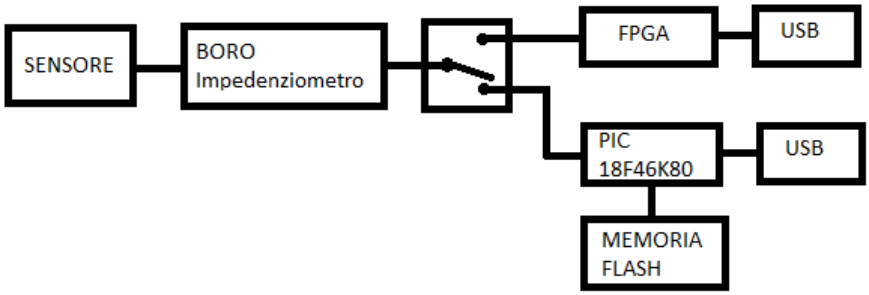


Figura 11 - Schema a blocchi del progetto

Analizziamo il sistema in dettaglio nella sua parte hardware, spiegando il funzionamento delle singole parti e le scelte effettuate in fase di progettazione.

La parte del sistema relativa al rilevamento delle variabili da monitorare è affidata al sensore precedentemente descritto: l'LFS 117 dell'IST, in grado di effettuare simultaneamente le letture di temperatura e conducibilità del liquido nel quale è immerso. Il Sensore, di tipo resistivo, è in grado di modificare la sua resistenza in maniera proporzionale alle variabili fisiche misurate: le letture effettuate sui valori assunti corrispondono agli ingressi del chip il quale, grazie ai suoi 4 core, è in grado di effettuare simultaneamente 4 letture distinte.

Due core del chip Boro sono interessati al campionamento di altrettante variabili ottenute dal sensore, gli altri due core non ancora utilizzati sono lasciati liberi con l'obiettivo di servirsene in fase di test per il rilevamento della temperatura del chip stesso tramite un sensore TP1000 al fine di studiare e correggere possibili errori di lettura della temperatura causati dal suo surriscaldamento.

Grazie ad una comunicazione tra l'unità di elaborazione e il chip avente luogo su un bus di tipo SPI si ottiene l'inizializzazione del chip grazie alla quale si ha la possibilità di impostare i parametri di funzionamento descritti nel capito "Introduzione al chip impedenziometrico".

Essendo Boro composto da 4 core indipendenti, esso dispone di 4 linee dati indipendenti OUT[1-4] in uscita dall'interfaccia abbinate ad altrettanti segnali di sincronismo SYNC[1-4]. Queste 8 linee sono dirette verso il microcontrollore il quale è incaricato di effettuare il campionamento delle uscite in accordo con i relativi segnali di sincronismo. Il microcontrollore è dunque incaricato di salvare i campioni provenienti in maniera continuativa dai canali d'uscita relativi alle letture effettuate e accumularne un numero sufficiente al fine di ottenere i 4 dati. Al termine del periodo d'acquisizione effettua un elaborazione dei dati raccolti secondo un algoritmo di filtraggio di tipo $Sinc^3$ descritto successivamente, provvede poi a salvare tali risultati ottenuti nella memoria esterna sempre grazie all'ausilio di una comunicazione su bus SPI.

In fase di progetto si è valutata la possibilità di sostituire il PIC con un modulo FPGA per lo studio di sviluppi futuri. Con l'ausilio di interruttori manuali si ottiene la mutua selezione dei collegamenti tra PIC-memoria/ FPGA-memoria e PIC-Boro/ FPGA-Boro. La scelta di interruttori manuali del tipo deep switch piuttosto che di tipo analogico non è casuale:

Pro switch manuali

- Non essendo previsti segnali di controllo e di abilitazione per gli switch manuali non vi è il rischio di imbattersi in errori dovuti alla presenza di correnti parassite.
- Con switch di tipo manuale non è necessario alcun tipo di controllo da parte del microcontrollore in quanto l'azionamento è manuale.

Contro switch analogici

- Con interruttori di tipo analogico vi è il rischio di generare errori causati da correnti parassite provocate dai segnali di controllo.
- Switch analogici prevedono, oltre alle piste per i segnali di controllo, anche piste per l'alimentazione, quindi un aumento dei collegamenti nel circuito e un ingombro totale maggiore.

Progetto Software

Forniamo ora le linee guida del progetto software che hanno portato alla realizzazione del firmware ed analizziamo step-by-step le istruzioni che l'unità di elaborazione, nel nostro caso il microcontrollore, deve compiere per arrivare all'obiettivo prestabilito. Nei sotto-capitoli successivi analizzeremo invece come tali istruzioni sono state realizzate, entrando maggiormente nel dettaglio commentando il codice relativo.

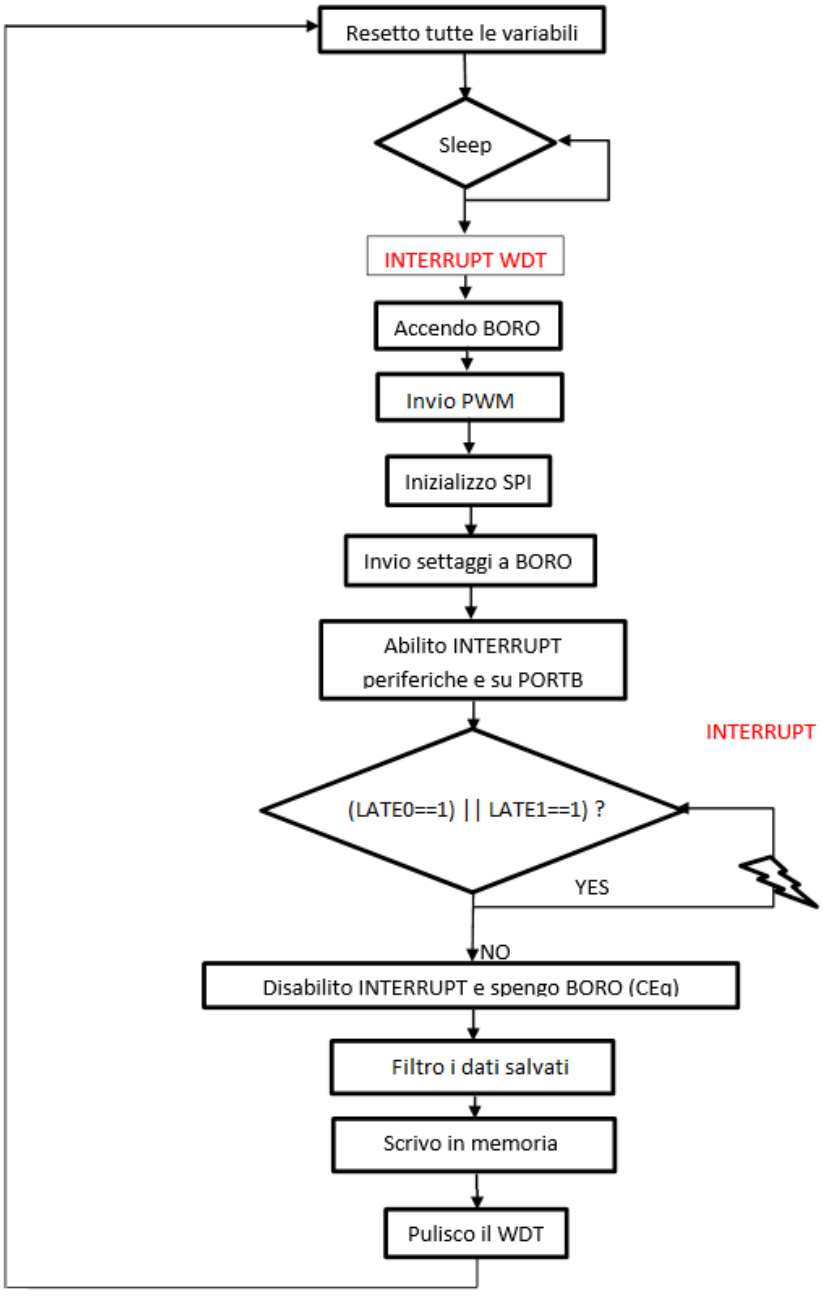


Figura 12 - Diagramma di flusso main program

Il codice è scritto in linguaggio C utilizzando l'ambiente di lavoro fornito gratuitamente dal produttore denominato MPLAB IDE (Integrated Development Environment), facendo uso del compilatore C18 ed utilizzando il Pickit 3 della Microchip per la programmazione della scheda. L'esigenza principale nella realizzazione del firmware è quella di rendere il flusso operativo il più efficiente possibile dal punto di vista energetico, limitando i consumi al minimo. Dovendo effettuare il salvataggio dei campioni relativi ai 4 canali ad intervalli regolari di 125 s la normale modalità di funzionamento del microcontrollore (*PRI_RUN_MODE*) per la fase di attesa tra i rilevamenti risulta essere altamente dispendiosa non essendoci operazioni da eseguire. Si sfrutta quindi la modalità di lavoro a risparmio energetico prevista dal PIC18F46K80 denominata *Sleep Mode*. Durante questa fase il microcontrollore non esegue istruzioni, arresta tutte le fonti di clock, disabilita le periferiche e rimane in attesa di essere svegliato dall'unica risorsa abilitata disponibile, ovvero il *Watchdog Timer*. Questo timer, gestito da un clock a basso consumo interno al PIC, permette di contare fino al periodo di tempo programmato al termine del quale, andando in overflow, riabilita il normale flusso operativo grazie ad un wake-up event: un interrupt. A questo punto il programma abilita ed inizializza Boro, genera il segnale PWM necessario al il chip per la lettura, accumula i campioni generati dall'interfaccia impedenziometrica, li filtra ed infine li salva in memoria. Al termine di questo ciclo, dopo aver resettato le variabili utilizzate e ripristinato il *Watchdog Timer*, il microcontrollore ritorna nella modalità di risparmio energetico *Sleep Mode* in attesa di ricominciare la sequenza di istruzioni. In Figura 12 viene mostrato il diagramma di flusso relativo al main program del microcontrollore: si può notare la sequenza di operazioni precedentemente descritte successive all'uscita dalla fase di Sleep. In Figura 19 e Figura 18, invece, sono raffigurate le routine di gestione degli interrupt nelle quali il microcontrollore svolge le seguenti funzioni:

- Riconosce l'arrivo di un nuovo campione proveniente dal chip e provvede a salvarlo, facendo distinzione che si tratti di un dato relativo alla parte reale o immaginaria, fino al raggiungimento del numero di campioni stabilito per un ciclo di lettura (Figura 19).

- Riconosce l’overflow del timer 4 e provvede a cambiare il duty-cycle al segnale PWM (Figura 18).

Main program

Per rendere il flusso di istruzioni ciclico si è fa uso della funzione `while(1){…}` la quale permette di ripetere le istruzioni inserite tra paretesi graffe all'infinito. Le prime istruzioni eseguite sono dedicate alla fase di sleep, di fatti viene abilitato l'interrupt sul watchdog e azzerato il timer ad esso associato con l'istruzione `CLRWDT`(), dopodiché si manda il programma in modalità a risparmio energetico tramite l'istruzione `SLEEP()`.

Sleep Mode

Il microcontrollore PIC18F46K80 dispone di una modalità di lavoro a risparmio energetico apposita per quelle applicazioni che necessitano di un consumo ridotto nelle fasi di attesa: il consumo previsto durante la fase di sleep è di 230 nA, al quale si aggiunge il consumo relativo al funzionamento del Watchdog Timer che in tale fase risulta essere di 600 nA. Si entra in *Sleep Mode* resettando il bit IDLEN del registro OSCCON ed eseguendo l'istruzione SLEEP(). Ciò provoca lo spegnimento di tutti i clock all'infuori dell'oscillatore LF-INTOSC (Low Frequency Internal Oscillator) oscillatore interno a bassa frequenza, nel caso in cui il Watchdog sia abilitato. In questo modo l'unico oscillatore sarà l'LF-INTOSC il quale provvederà a gestire il timer del Watchdog incrementandolo ad ogni fronte. Per far uscire il programma dalla *Sleep Mode* e farlo tornare alla modalità di lavoro prestabilita è necessario un wake-up event, un evento che lo "svegli". Nel progetto in questione l'evento che permette l'uscita dalla modalità a risparmio energetico è l'interrupt del *Watchdog-Timer*: il clock interno provvede a incrementare il contatore fino al raggiungimento del valore programmato, l'overflow sul conteggio permette al programma di proseguire con le istruzioni ed uscire dalla fase di attesa. La gestione del periodo del timer è affidata ad un postscaler che divide il clock in periodi multipli, configurabile attraverso i bit WDTPS<4:0> del registro di configurazione CONFIG2H. Assegnando a tali bit i valori 0b01111 ottengo un fattore di divisione del periodo di 1:32,768, ovvero un periodo di 131,072 s [5]. In fase di progetto è stato inizialmente calcolato un periodo di 125 secondi, valore però non rigoroso ma flessibile e quindi sostituibile con quello fornito dal postscaler. Per abilitare e disabi-

litare il watchdog timer è necessario semplicemente modificare il bit SWDTEN del registro WDTCON: a livello logico alto lo abilito mentre per spegnerlo basta portarlo a livello logico basso. Infine per resettare il timer ad esso relativo si invoca la funzione CLRWDT(), funzione contenuta nella libreria "pic18.h" fornita da Microchip.

Inizializzazione chip BORO

Una volta attesi 131 secondi l'overflow sul timer genera un interrupt il quale permette di tornare in modalità *Primary_run_mode* e proseguire con il ciclo di istruzioni. L'istruzione immediatamente successiva riguarda l'abilitazione di Boro: invocando la funzione *void BoroInit(void)* provvedo ad accendere il chip e ad inizializzarlo tramite l'invio dei dati relativi alla configurazione desiderata.

L'inizializzazione dell'interfaccia impedenziometrica è stata prevista dagli sviluppatori del chip attraverso la trasmissione di una parola di configurazione della lunghezza di 64 bit su un bus di comunicazione di tipo SPI dove il microcontrollore ricopre il ruolo di master. Il bit-stream di dati si compone di 4 istruzioni della lunghezza di 16 bit ciascuno, una per ogni core. In Figura 2**Errore. L'origine riferimento non è stata trovata.**, nel capitolo "Introduzione al chip Boro", è mostrato lo schema a blocchi del chip nel quale è visibile la parte relativa alla comunicazione in questione. In Figura 13 viene

mostrata la suddivisione dell'istruzione complessiva nelle sue 4 parti ognuna delle quali configura un core consentendo di:

- Abilitare il core.
- Abilitare il Bandgap-Reference.
- Scegliere il guadagno dell'amplificatore LNA: x 0.5/1/2/5/10/20/50/100.
- Scegliere la corrente di riferimento necessaria al convertitore V/I: 10/150/300/1000 µA.

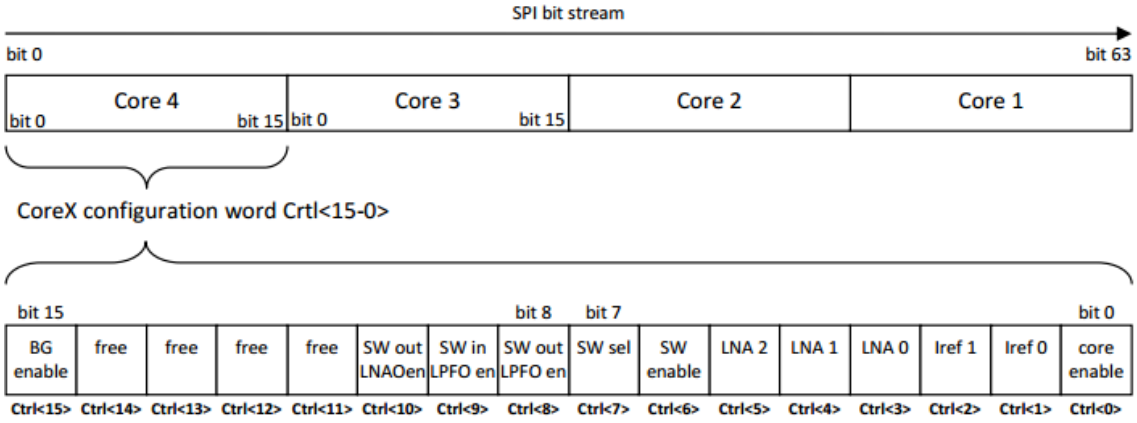


Figura 13 - BORO SPI module da [6]

Il chip Boro per funzionare necessita di un segnale di clock a frequenza costante di 500 kHz [2], sfruttiamo quindi il pin SCK (Serial ClocK) del microcontrollore sul quale è presente il clock del modulo SPI e impostiamo la frequenza di comunicazione al valore desiderato. Per generare il clock alla frequenza di 500 kHz (F_{BORO}) determino il fattore di divisione (CK_{DIV}) da utilizzare per scalare opportunamente il clock di sistema (F_{OSC}=$F_{SYSTEM:}$ frequenza del sistema, fornita dal quarzo esterno a 8 MHz):

$$F_{BORO} = \frac{F_{SYSTEM}}{CK_{Div}} \rightarrow CK_{Div} = \frac{8 \cdot 10^6}{500 \cdot 10^3} = 16$$

Operando sugli opportuni registri del modulo SPI lo si configura:

- Abilito il sincronismo sulla serial port: `SSPCON1bits.SSPEN=1;`

- Imposto come ruolo del dispositivo nella comunicazione quello di Master e il fattore di divisione per ottenere la frequenza a 500 kHz (Fosc/16): `SSPCON1bits.SSMP=0b0001;`
- Seleziono il livello logico basso per lo stato di idle del clock: `SSPCON1bits.CKP=0;`
- La trasmissione dei dati avviene in corrispondenza del fronte positivo del clock: `SSPSTATbits.CKE=0;`
- I dati in ingresso sono campionati a metà del tempo dei dati in uscita: `SSPSTATbits.SMP=1;`

Al fine di modificare la velocità di scrittura dati a seconda del dispositivo utilizzato (chip o memoria), ad ogni inizio di comunicazione imposto la frequenza di lavoro del modulo SPI.

Una volta aperto il canale SPI provvedo ad abilitare lo slave con il Chip Select, semplicemente impostando a 0 il registro del pin relativo: `LATAbits.LATA5=0`. La gestione del CS dell'interfaccia è opposta al consueto utilizzo, infatti in questo caso l'abilitazione dello slave Boro viene effettuata portando a valore logico alto il CS; la ragione risiede nella natura del chip il quale è stato sviluppato con la seguente specifica.

Per inviare i dati con il modulo SPI è sufficiente caricare nel registro del master SSPBUF il dato e questo verrà inviato bit per bit ad ogni fronte positivo di clock. La funzione di scrittura consente di inviare variabili da 1 Byte e per questo motivo nella configurazione dell'interfaccia impedenziometrica la parola di codice da 64 bit viene suddivisa in 8 parole di codice da 1 Byte ciascuno, relativamente BOROSPI_CORExa e BOROSPI_CORExb (con x=[1-4]) per tutti e 4 i core la cui configurazione è assegnata in fase di inizializzazione.

Il codice relativo all'inizializzazione dell'interfaccia impedenziometrica si compone in questo modo:

```
TRISAbits.TRISA5=0;             //Setto il CS di BORO (output)
LATA5=1;                        //attivo il CS (alto per atti-
varlo)

SSPBUF = BOROSPI_CORE4a;        //1° Byte del CORE4 (MSB)
```

```
while (!SSPSTATbits.BF);            //SSPBUF è pieno?
Delay10KTCYx(2);                    //Ritardo di 10ms

SSPBUF = BOROSPI_CORE4b;            //2° Byte del CORE4 (LSB)
while (!SSPSTATbits.BF);
Delay10KTCYx(2);

SSPBUF = BOROSPI_CORE3a;            //1° Byte del CORE3 (MSB)
while (!SSPSTATbits.BF);
Delay10KTCYx(2);

SSPBUF = BOROSPI_CORE3b;            //2° Byte del CORE3 (LSB)
while (!SSPSTATbits.BF);
Delay10KTCYx(2);

SSPBUF = BOROSPI_CORE2a;            //1° Byte del CORE2 (MSB)
while (!SSPSTATbits.BF);
Delay10KTCYx(2);

SSPBUF = BOROSPI_CORE2b;            //2° Byte del CORE2 (LSB)
while (!SSPSTATbits.BF);
Delay10KTCYx(2);

SSPBUF = BOROSPI_CORE1a;            //1° Byte del CORE1 (MSB)
while (!SSPSTATbits.BF);
Delay10KTCYx(2);

SSPBUF = BOROSPI_CORE1b;            //2° Byte del CORE1 (LSB)
while (!SSPSTATbits.BF);
Delay10KTCYx(2);

LATA5=0;                            //Disattivo il CS
```

Dopo aver copiato il dato nel registro del master attendo il suo completo invio prima di procedere con il dato successivo: con l'istruzione `while (!SSPSTATbits.BF)` effettuo un controllo sul flag del registro SPI il quale viene settato a livello logico 1 non appena il contenuto del registro SSPBUF è stato completamente trasmesso. Inserendo infine un ritardo di 10 ms tra l'invio di un dato e l'altro assicuro l'assenza di conflitti in fase di trasmissione. In Figura 14 è visibile il segnale BOROSPI_CORE4b, ovvero 0xB1 (valore impostato in fase di programmazione), inviato tramite SPI al chip.

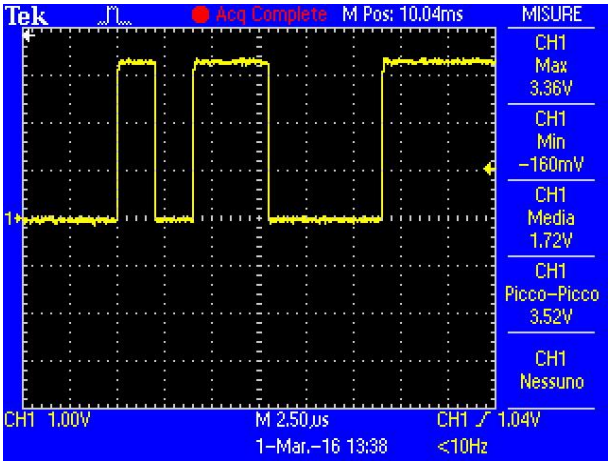


Figura 14 - Segnale BOROSPI_CORE4b inviato

Generazione segnale PWM

Successivamente, invocando la funzione *void Pwm(void)*, provvedo a generare il segnale per la lettura dell'impedenza da parte del chip.

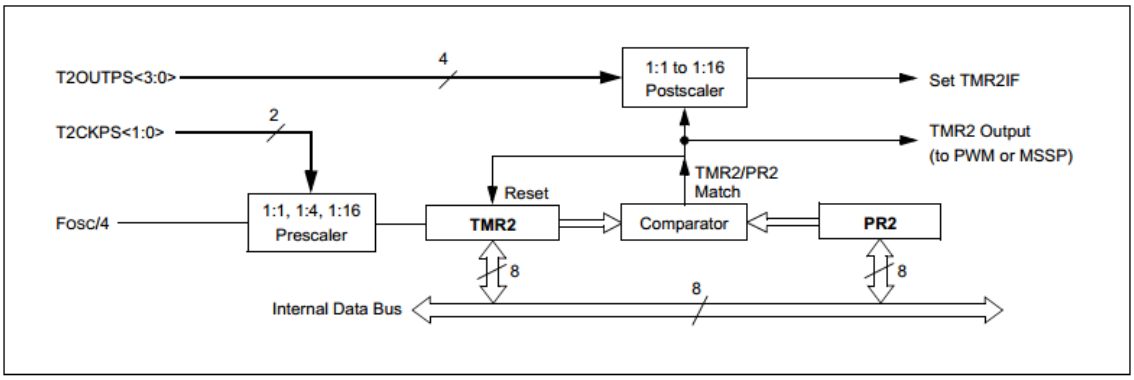


Figura 15 - Diagramma a blocchi Timer2 da [5]

Un segnale modulato PWM è simile ad uno ottenuto tramite una modulazione di tipo ΔΣ, segnale previsto dagli sviluppatori del chip. Per ragioni di semplicità realizziamo uno stream di dati, più piccolo (solamente 50 valori) rispetto a quello utilizzato durante la realizzazione del chip, modulato PWM anziché utilizzare un segnale ΔΣ di dimensioni molto maggiori. Otteniamo ciò generando un segnale periodico ed andando a modificare il valore del duty-cycle ogni 2 periodi di clock.

Realizzare tale segnale è possibile grazie all'ausilio del modulo integrato *Capture/Compare/PWM* presente nel microcontrollore abbinato ad un timer utilizzato come base dei tempi. Il timer in questione è il Timer2 la cui struttura interna è visibile in Figura 15: il registro PR2 rappresenta il registro

relativo al periodo da assegnare al timer ed infatti ad ogni ciclo di clock il comparatore confronta il registro TMR2, che incrementa periodicamente, con il registro PR2 precaricato con il valore da raggiungere; l'uguaglianza produce l'output e l'azzeramento del TMR2. Visto il suo funzionamento provvediamo a programmarlo per impostare la frequenza del segnale PWM, in questo caso volendo rappresentare una sinusoide alla frequenza di 1 kHz prendiamo un valore di frequenza per il segnale di almeno due ordini di grandezza maggiore, ovvero 100 kHz ($T_{PWM} = \frac{1}{100\ KHz} = 10\ \mu s$).

La base dei tempi utilizzata per il funzionamento del Timer2 è il clock esterno il quale fornisce un segnale alla frequenza di 8 MHz ($T_{CK} = \frac{1}{8\ MHz} = 125\ ns$). Stabiliamo quindi il numero di conteggi effettuati dal contatore al fine di scalare il clock di sistema ($n_{conteggio}$), quindi individuiamo il valore con cui precaricare il PR2:

$$n_{conteggio} = \frac{T_{PWM}}{T_{ck}} = \frac{10 \cdot 10^{-6}}{125 \cdot 10^{-9}} = 80$$

scriviamo quindi `PR2= 8000000/80`.

Otteniamo così un segnale periodico sulla base dei tempi del Timer2.

Per generare la forma d'onda richiesta è necessario che il programma sia in grado di modificare il duty-cycle del segnale ogni 20 µsec, ovvero ogni due periodi del segnale generato dal Timer2, risultato ottenibile attraverso l'ausilio di un altro timer interno al microcontrollore. Il contatore utilizzato è il Timer 4 il quale, non appena raggiunto il valore precaricato nel registro PR4, (non appena termina il conteggio prestabilito), va in overflow producendo un interrupt che ci consente di assegnare un differente valore al duty-cycle. Utilizzando come sorgente dei tempi sempre il clock esterno ad 8, procediamo ad individuare il valore da caricare in PR4 al fine di ottenere l'interrupt ogni 20µs come desiderato, scalando anche in questo caso il clock di sistema:

$$n_{conteggio} = \frac{T_{interrupt}}{T_{ck}} = \frac{20 \cdot 10^{-6}}{125 \cdot 10^{-9}} = 160$$

scriviamo quindi `PR4= 8000000/160` o `PR4= PR2*2`.

Si fanno quindi uso di due timer: il Timer2 abbinato al modulo PWM per la generazione del segnale e il Timer4 per intervenire ogni due periodi del PWM, ovvero ogni 20 µs, codificando il duty-cycle del segnale secondo dei valori prestabiliti dalla mappatura della forma d'onda di riferimento. I valori di duty-cycle da assegnare al segnale, al fine di ottenere il risultato desiderato, sono prestabiliti e salvati all'interno di una array di 50 elementi: Ta-

ble[50]. Per una descrizione più dettagliata riguardo i valori salvati ed utilizzati per la generazione del segnale PWM si rimanda il lettore all'elaborato dell'Ing. Martini [1].

Per ottenere ciò è necessario programmare opportunamente i due timer agendo sui rispettivi registri interni:

```
//TIMER2
PR2=8000000/80;                          //buffer per TMR2
T2CONbits.T2OUTPS=0b0000;                //postscaler 1:1
T2CONbits.TMR2ON=1;                      //abilito il timer2
T2CONbits.T2CKPS=0b00;                   //prescaler 1:1
PIE1bits.TMR2IE=0;                       //interrupt OFF

//TIMER4
PR4=2*PR2;                               //buffer per TMR4
(Doppio di PR2, intervengo 1 volta ogni 2 cicli)
T4CONbits.T4OUTPS=0b0000;                //postscaler 1:1
T4CONbits.TMR4ON=1;                      //abilito il timer4
T4CONbits.T4CKPS=0b00;                   //prescaler 1:1
PIE4bits.TMR4IE=1;                       //Interrupt ON
```

Il modulo PWM interno al PIC18F è attivabile e programmabile operando sui registri CCPxCON e CCPTMRS:

```
CCP1CONbits.CCP1M=0b1100;          //scelgo il modulo PWM
CCPTMRSbits.C1TSEL=0;              //modulo basato sul Ti-
mer2
```

Il periodo del PWM si basa sul valore precaricato nel registro PRx a seconda del timer utilizzato. Per settare il duty-cycle del segnale invece occorre operare sul registro CCPRxL: il valore da attribuire in questo caso lo impostiamo ogni 20 µs grazie all'interrupt generato dal Timer4.

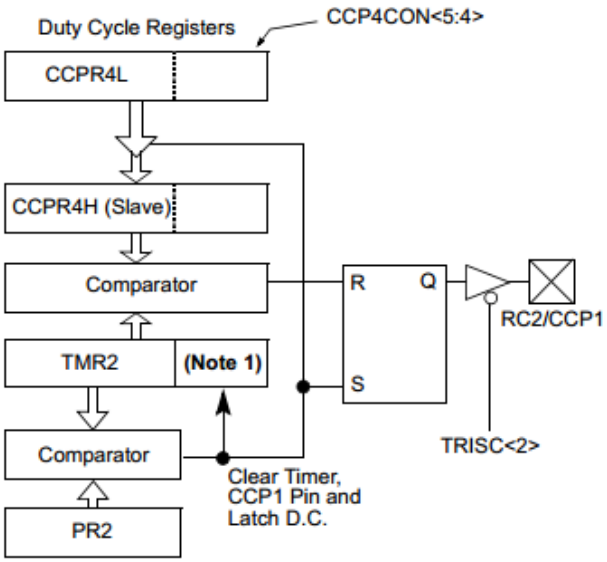


Figura 16 - Composizione interna modulo PWM da [5]

Riportato in Figura 17 viene mostrato il segnale PWM generato dal micro-controllore e osservato tramite l'utilizzo di un oscilloscopio sul pin DACIN del chip.

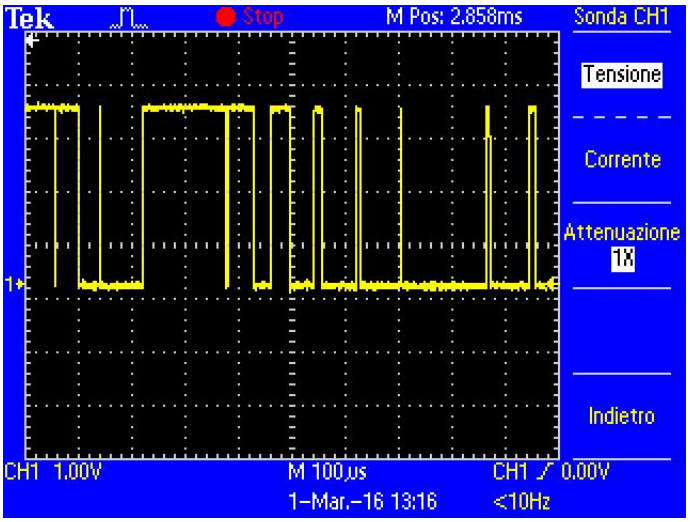


Figura 17 - Segnale PWM in ingresso al chip

ISR per il Timer 4

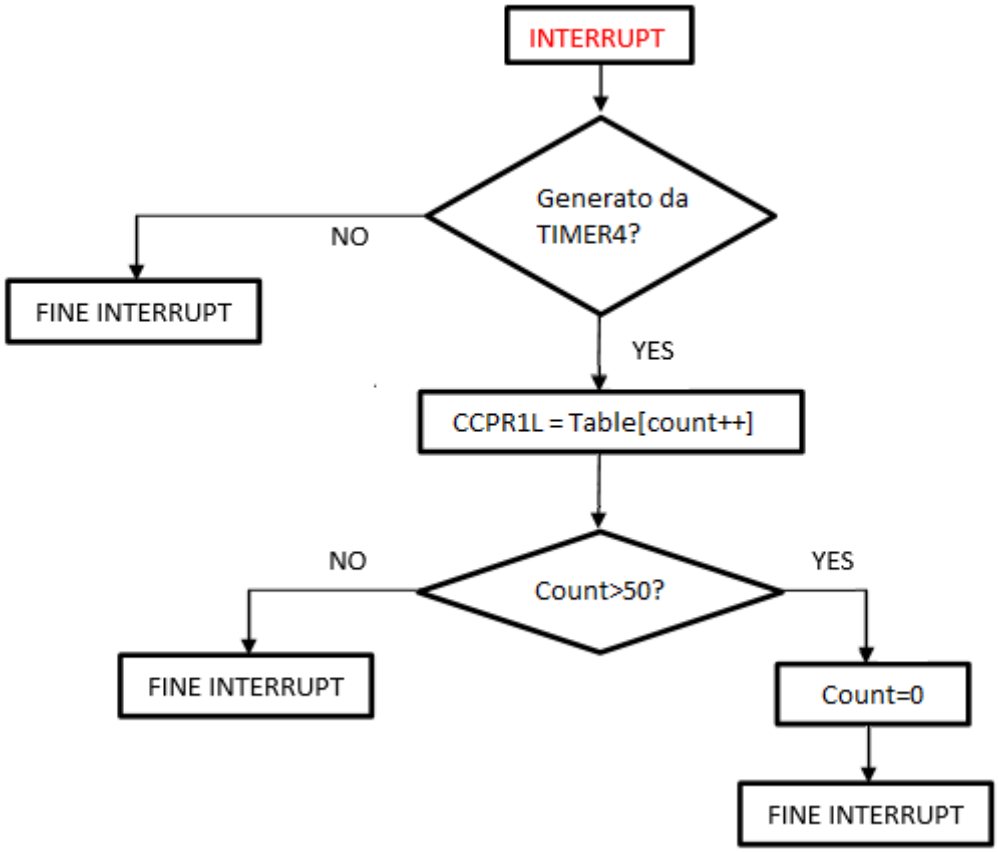


Figura 18 - Diagramma di flusso della gestione interrupt Timer 4

Provvediamo a realizzare una routine di servizio (ISR = Interrupt Service Routine) in grado di occuparsi dell'interrupt generato dal Timer 4. All'avvenimento di ogni interrupt viene invocata in automatico la funzione *void isr(void)* la quale provvede a gestire tali interruzioni. Al suo interno viene effettuato un controllo sulla natura dell'interrupt per capire quale evento sia stato la causa scatenante e poter quindi operare di conseguenza; tale controllo viene fatto grazie a dei flag associati ad ogni interrupt. Nel caso del Timer4 il flag relativo all'interrupt è il TMR4IF nel registro PIR4: questo in automatico viene settato ad 1 ogni qualvolta si manifesta l'evento.

Di conseguenza provvediamo a modificare il duty-cycle del segnale PWM andando a cambiare il valore precedente contenuto nel registro CCPR4L at-

tribuendogli un valore della tabella e tenendo traccia della posizione all'interno di essa con la variabile *count* la quale ci permette di ricominciare dal primo elemento ogni volta che viene raggiunto l'ultimo:

```
if(PIR4bits.TMR4IF==1){            //Interrupt Timer4?
     CCPR5L = Table[count++];      //Cambio il duty

     if(count>=50){                //finiti gli elementi
     count = 0;                    della tabella ricomincio
     }
     PIR4bits.TMR4IF=0;            //resetto il flag TIMER4
}
```

La tabella contenente i 50 valori assunti dal duty-cycle del segnale viene dichiarata globalmente nel modo seguente:

```
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};
```

Abilitazione Interrupt

Ora che il chip Boro è acceso e funzionante possiamo attivare gli interrupt, i quali permettono di rilevare il segnale di SYNC incaricato di comunicare la presenza dei dati presenti sui 4 canali dell'interfaccia impedenziometrica. La funzione invocata è *void InterruptInit(void)*:

```
     TRISBbits.TRISB6=1;      //setto RB6 come Input

     //abilito l'interrupt su RB6 (SYNC4) e disabilito tutti
     gli altri
     IOCBbits.IOCB4=0;
     IOCBbits.IOCB5=0;
     IOCBbits.IOCB6=1;
     IOCBbits.IOCB7=0;

     INTCONbits.GIE=1;   //abilito l'interrupt globale
     INTCONbits.PEIE=1;  //abilito l'interrupt delle perife-
     riche
     INTCONbits.RBIE=1;  //abilito gli interrupt on change
     (su RB<7-4>)
     RCONbits.IPEN=0;    //disabilito i livelli di priorità
     degli interrupt

     PIE4bits.TMR4IE=1;  //abilito l'interrupt del TIMER4
```

Operando singolarmente sui bit del registro IOCB abilito l'interrupt on change solo sul pin RB6 dov'è presente il segnale SYNC e disabilito tutti gli altri.

Provvedo poi ad abilitare: l'interrupt globale, l'interrupt delle periferiche e l'interrupt on change settando rispettivamente i bit GIE, PEIE e RBIE del registro INTCON. Abilito poi l'interrupt relativo al timer 4 settando ad 1 il bit TMR4IE. Per concludere disabilito i livelli di priorità degli interrupt, non necessari per il progetto.

Il programma rimane quindi in attesa che il salvataggio dei campioni sia ultimato; tale processo viene spiegato in dettaglio nel capitolo successivo.

Campionamento dati e Interrupt On Change

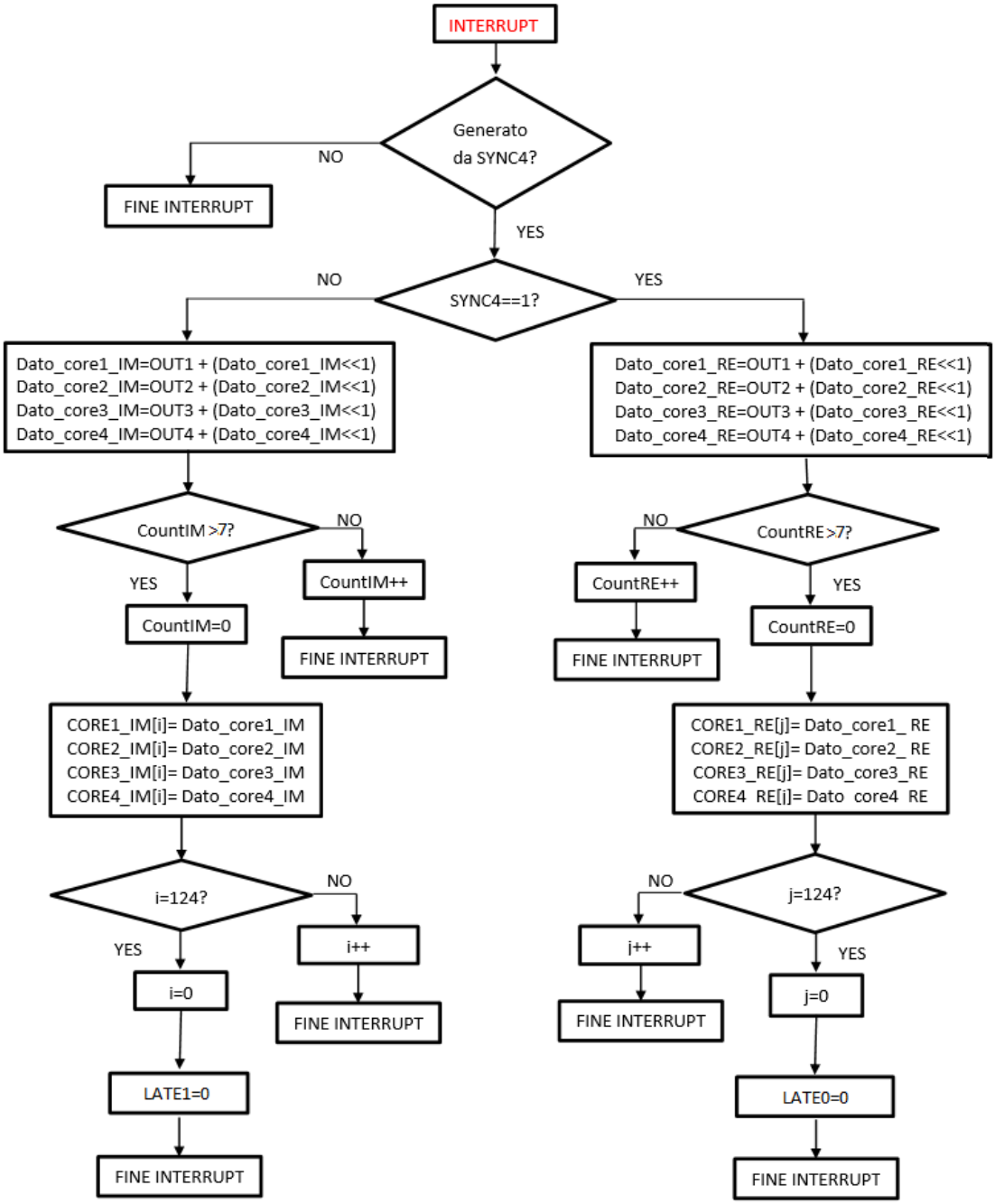


Figura 19 - Diagramma di flusso della gestione interrupt di BORO

Per ogni core presente all'interno dell'interfaccia troviamo in uscita un pin denominato OUTx ed un segnale di sincronismo SYNC comune a tutti i core (per tale ragione ci si mette in ascolto solo di un segnale di sincronismo, il SYNC4). La lettura della parte reale e della parte immaginaria, componenti

il numero complesso rappresentativo dell'impedenza misurata, viene scandita dai fronti di salita e discesa del segnale SYNC. L'importanza di questo segnale è evidente in Figura 20 dove viene mostrato il campionamento del segnale d'uscita e il corrispondete fronte del segnale di sincronismo: esso è stato progettato con uno sfasamento rispetto al segnale d'uscita ottenendo così un cambio di fronte del SYNC esattamente a metà del semiperiodo del dato; in questo modo è possibile campionare con facilità parte reale ed immaginaria dell'impedenza. La tecnica di campionamento consiste pertanto nel rilevare il tipo di fronte del segnale SYNC e salvare il segnale OUTx, ottenendo così i campioni validi per la parte reale ed immaginaria del dato.

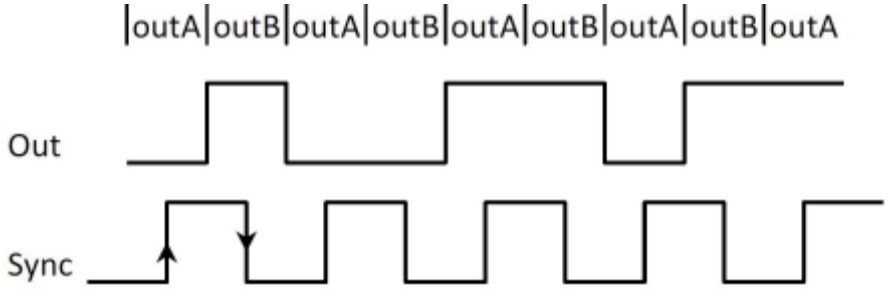


Figura 20 - Campionamento segnale d'uscita da [6]

Per realizzare il campionamento ad ogni fronte del segnale di sincronismo ci viene in aiuto un particolare interrupt integrato dal microcontrollore denominato "Interrupt On Change". Questa particolare funzione è attivabile solo sui pin RB<4-7> ed è in grado di generare un interrupt ogni qualvolta il valore alla porta di uno dei pin commuta rispetto lo stato precedente. Abilitando questo particolare interrupt al pin relativo al segnale di sincronismo SYNC siamo in grado di rilevare sia i fronti positivi che negativi e processare i dati opportunamente. Essendo il segnale di sincronismo comune a tutti e 4 i core sfruttiamo la presenza del segnale SYNC4 sul pin RB6 attivando l'interrupt on change ed abilitandolo solo su tale pin. In questo modo, ad ogni commutazione del segnale di sincronismo, possiamo campionare i dati provenienti da tutti i core.

Il programma rimane ora in attesa dell'avvenuta acquisizione dei campioni, condizione che permette il proseguimento della sequenza di istruzioni. Ogni volta che l'interrupt viene generato si interrompe il normale flusso operativo

e si provvede a servire l'interruzione attraverso una routine specifica. Per prima cosa viene effettuato un controllo della causa di interruzione: ad ogni interrupt è associato un flag il quale viene impostato al valore logico 1 in automatico dal microcontrollore ogni volta che viene generato. Il flag relativo all'interrupt on change è il bit RBIF del registro INTCON. Dopo questo controllo si provvede a riconoscere il tipo di fronte, se di salita o discesa, per effettuare il campionamento della parte reale o immaginaria. Tale controllo viene effettuato sul valore alla porta del segnale SYNC4: se la porta RB6 si trova a valore logico 1 si provvede a processare la parte reale, se si trova al valore logico 0 quella immaginaria. Al termine del singolo campionamento si resetta il flag riportandolo a 0 ed uscendo dalla routine rimanendo in attesa del campione successivo.

Per la generazione dei dati ottenuti dal salvataggio dei campioni predisponiamo 8 array di tipo *unsigned char* di 125 elementi ciascuno (125 elementi della lunghezza di 1 Byte), due array per ogni core, uno relativo alla parte reale ed uno per quella immaginaria:

- CORE1_RE[124], CORE1_IM[124]
- CORE2_RE[124], CORE2_IM[124]
- CORE3_RE[124], CORE3_IM[124]
- CORE4_RE[124], CORE4_IM[124]

Per il salvataggio dei campioni all'interno degli otto array sfruttiamo altrettante variabili (*unsigned char*) utilizzate come degli *shift register*:

- dato_core1_RE, dato_core1_IM
- dato_core2_RE, dato_core2_IM
- dato_core3_RE, dato_core3_IM
- dato_core4_RE, dato_core4_IM

Queste variabili fungono da registri nei quali vengono salvati in successione 8 campioni consecutivi provenienti da Boro; al termine dell'acquisizione i registri vengono copiati in un elemento dell'array e svuotati per essere nuovamente riutilizzati per comporre il byte successivo. Il processo a luogo fino al completamento dell'array, ovvero dopo la creazione di 125 elementi.

In Figura 21 viene riportata una raffigurazione della modalità di campionamento e salvataggio dei dati appena descritta. La situazione mostrata è relativa solamente al Core1 mentre nel progetto questa struttura di salvataggio dati viene replicata per tutti i core dell'interfaccia.

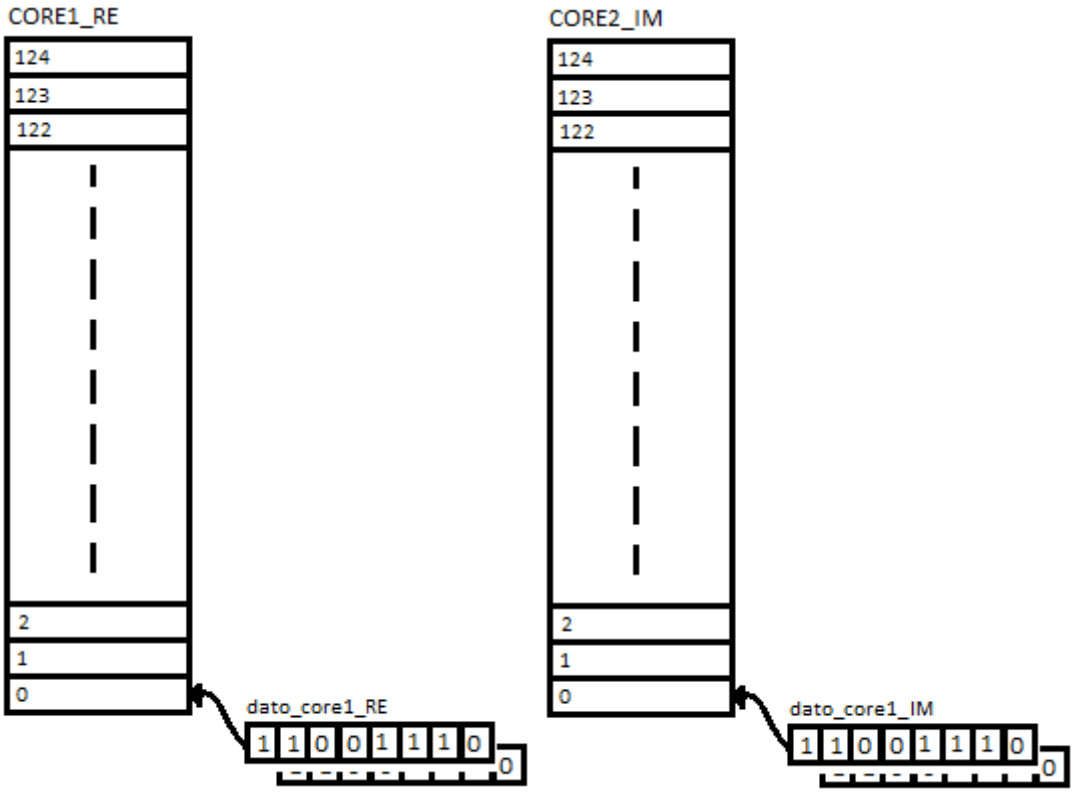


Figura 21 - Campionamento dati (SOLO CORE 1)

Per comunicare al programma il completamento degli array, al raggiungimento del 125° elemento viene settato un flag a valore logico 0. I due flag utilizzati per rappresentare la fine dell'acquisizione della parte reale ed immaginaria non sono altro che due registri: LATE0 e LATE1. La condizione di uscita dal loop di attesa è rappresentata da una funzione logica XOR dei due flag: `(LATEbits.LATE0==1) || (LATEbits.LATE1==1)`. Il risultato di tale logica risulta essere differente da 1 (condizione per la quale si ha l'uscita dal ciclo while) esclusivamente quando entrambi i registri sono a valore logico 0, ovvero quando entrambe le parti dell'impedenza sono state acquisite correttamente.

L'istruzione che permette al microcontrollore di rimanere in attesa del completo campionamento è quindi la seguente:

```
while((LATEbits.LATE0==1) || (LATEbits.LATE1==1));
```

Il codice che realizza il corretto campionamento e l'acquisizione dei dati nei registri e successivamente nei rispettivi array è riportato qui di seguito:

```
if(INTCONbits.RBIF==1){    //Uno tra RB<7-4> è cambiato di
stato?
     if(SYNC4==1){         //FRONTE ↑= salvo la parte RE

          dato_core1_RE= OUT1 + (dato_core1_RE<<1);
          dato_core2_RE= OUT2 + (dato_core2_RE<<1);
          dato_core3_RE= OUT3 + (dato_core3_RE<<1);
          dato_core4_RE= OUT4 + (dato_core4_RE<<1);
          if(countRE>7){ //se ho completato il registro a
                         8bit salvo il dato nell'array
               countRE=0;
               CORE1_RE[j]=dato_core1_RE;
               CORE2_RE[j]=dato_core2_RE;
               CORE3_RE[j]=dato_core3_RE;
               CORE4_RE[j]=dato_core4_RE;
               if(j==125){    //array ultimato?
                    j=0;
                    LATEbits.LATE0=0;   //setto il flag di
                              fine acquisizione
               }else{
                    j++; //altrimenti incremento l'indice
               }
          }else{         //Se countRE è !=7 lo incremento
               countRE++;
          }
     }else{              //FRONTE ↓= salvo la parte IM

          dato_core1_IM= OUT1 + (dato_core1_IM<<1);
          dato_core2_IM= OUT2 + (dato_core2_IM<<1);
          dato_core3_IM= OUT3 + (dato_core3_IM<<1);
          dato_core4_IM= OUT4 + (dato_core4_IM<<1);
          if(countIM>7){      //se ho completato il regi-
                              stro a 8bit salvo il dato
                              nell'array

               countIM=0;
               CORE1_IM[i]=dato_core1_IM;
               CORE2_IM[i]=dato_core2_IM;
               CORE3_IM[i]=dato_core3_IM;
               CORE4_IM[i]=dato_core4_IM;
               if(i==125){    //se sono arrivato alla fine
                              dell'array
                    i=0;
```

```
                    LATEbits.LATE1=0;   //setto il flag di
               fine acquisizione
               }else{
                    i++; //altrimenti incremento l'indice
               }
          }else{         //Se countIM è !=7 lo incremento

               countIM++;
          }
     }
     INTCONbits.RBIF=0;     //resetto il flag dell'interrupt
}
```

Il primo controllo riguarda il flag relativo all'interrupt che è stato generato per comprendere se si tratta dell'Interrupt On Change o di un altro tipo di interruzione. Il controllo successivo riguarda il valore del segnale SYNC4 distinguendo tra fronte di salita e di discesa, processando quindi la relativa parte dell'impedenza. I valori presenti alle porte OUT[1-4] vengono salvati nelle variabili ad 8 bit utilizzate come registri le quali vengono shiftate di una posizione a sinistra per fare in modo che i dati vengano aggiunti nella prima posizione a destra come LSB. Si esegue poi un controllo su una variabile, *countRE* per la parte reale e *countIM* per quella immaginaria, incaricata di tener traccia del numero di campioni salvati nel registro: se questa risulta essere > 8 significa che il registro è stato riempito ed è necessario copiare il suo contenuto all'interno dell'array. Se il registro invece non è ancora completo si incrementa la variabile designata al conteggio dei campioni, si resetta il flag relativo all'interrupt e si interrompe la routine.

Se il registro contenente i campioni risulta essere completo è necessario copiarlo in un elemento del relativo array: dopo aver resettato la variabile di conteggio dei campioni si copia il contenuto del registro nell'elemento j-esimo/i-esimo dell'array relativo alla parte reale/immaginaria del dato. Come in precedenza, viene effettuato un controllo su una variabile designata al conteggio degli elementi dell'array, *j* per la parte reale e *i* per quella immaginaria, la quale tiene traccia del numero di elementi salvati: se risulta essere uguale a 124 (cioè 125 elementi) significa che l'ultimo salvataggio

effettuato ha provveduto ad ultimare l'array, pertanto viene resettato il registro LATE0/LATE1 per comunicare l'avvenuto completamento del campionamento. In caso contrario si procede invece incrementando la variabile, resettando il flag dell'interrupt e interrompendo la routine. Ad acquisizione ultimata si avrà:

Lettura Core 1: 125 x8 campioni per la parte Reale, 125 x8 campioni per la parte Immaginaria.
Lettura Core 2: 125 x8 campioni per la parte Reale, 125 x8 campioni per la parte Immaginaria.
Lettura Core 3: 125 x8 campioni per la parte Reale, 125 x8 campioni per la parte Immaginaria.
Lettura Core 4: 125 x8 campioni per la parte Reale, 125 x8 campioni per la parte Immaginaria.

La scelta di utilizzare registri ad 8 bit e di realizzare degli array composti da elementi di tipo *char* (variabili di 8 bit) è stata condizionata dall'architettura interna ad 8 bit del microcontrollore utilizzato. Tale scelta di salvataggio risulta essere vantaggiosa anche per la comunicazione tramite in quanto le funzioni di scrittura e lettura operano su dati della lunghezza di 1 Byte ed è pertanto semplice operare sui singoli elementi degli array senza dover effettuare delle concatenazioni o delle suddivisioni dei dati.

Infine, siccome il chip è stato progettato con la necessità di ricevere un segnale periodico di 500 kHz fissi per tutto il periodo di funzionamento [2], facciamo uso del clock del modulo SPI utilizzato per sincronizzare l'invio dei dati. Basterà semplicemente caricare ripetutamente il registro di invio dell'SPI SSPBUF con dei dati (0x00) in modo da ottenere la generazione del clock desiderato per il loro invio; disabilitiamo il CS del chip per renderlo insensibile ai dati ricevuti in modo da sfruttare il solo clock. Questa operazione viene eseguita durante tutto il periodo di attesa del completo salvataggio dei campioni.

L’istruzione immediatamente successiva all’uscita dal loop di attesa disabilita gli interrupt on change per rendere il programma insensibile al cambiamento di stato del segnale di sincronismo e provvede a spegnere Boro per un maggior risparmio energetico.

Filtraggio dati

I dati acquisiti sono ora di dimensioni ragguardevoli rispetto alla capacità totale della memoria per poter essere salvati per due anni, inoltre l’interfaccia impedenziometrica utilizzata appartiene alla categoria dei modulatori delta-sigma perciò necessitano un filtraggio particolare. Un modulatore di questo tipo è il cuore dei convertitori ADC [7] ed è un metodo per tradurre segnali ad alta risoluzione in segnali a bassa risoluzione tramite l'uso della modulazione a densità di impulsi. Un segnale analogico applicato all’ingresso del convertitore viene campionato più volte con una tecnica denominata sovracampionamento, il valore rilevato viene confrontato con quello precedente e successivamente integrato. L’uscita di un modulatore si presenta come un treno di impulsi identici di ampiezza V e durata dt, ciò che varia è l’intervallo di separazione tra di essi. Questo è dovuto alla presenza dell’anello in retroazione negativa il quale fa in modo 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, Figura 22.

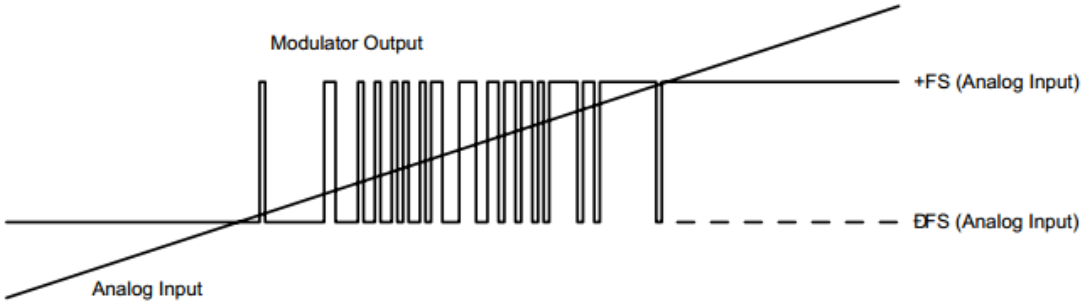


Figura 22 - Output modulatore delta sigma da [7]

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. Il numero N di campioni è anche detto Over Sampling Ratio o OSR e rappresenta il fattore di decimazione; ovviamente un aumento dell'OSR comporta un aumento della risoluzione in quanto viene aumentata la finestra di osservazione a discapito della banda occupata in quanto bisognerà attendere un numero di campioni maggiori prima di poterne calcolare la media.

Una tecnica di filtraggio come la media temporale a finestra mobile è però spesso insufficiente e vengono richieste soluzioni più elaborate come un filtraggio di tipo $Sinc^K$, dove K rappresenta l'ordine del filtro, spesso utilizzato per arrivare alla profondità di conversione desiderata. Questo tipo di filtraggio consiste nell'implementare K blocchi accumulatori in cascata operanti alla frequenza di campionamento f_s, seguiti da altrettanti blocchi differenziatori operanti alla frequenza f_s/OSR [1]. In Figura 23 è mostrato lo schema a blocchi rappresentativo il filtro $Sinc^3$, si nota come il segnale filtrato in uscita abbia la frequenza dipendente dal fattore di decimazione OSR:

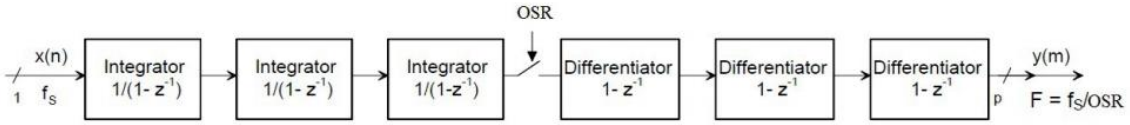


Figura 23 - Schema a blocchi filtro Sinc3 da [7]

La risoluzione del segnale in uscita dal filtro è funzione del parametro OSR e dell'ordine K del filtro e la dimensione della parola in uscita da un filtro di ordine K risulta essere più larga di quella in ingresso di un fattore p: $p = K \cdot log_2 OSR$ [7].

Supponendo una frequenza di campionamento f_s=10 MHz e un filtro di tipo $Sinc^3$ è possibile confrontare i risultati per un OSR che va da 4 a 256 Figura 24.

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

Figura 24 - OSR e Output Word size a confronto da [7]

L'implementazione di tale filtro la otteniamo invocando la funzione *void Filtraggio(void)*:

- scorriamo tutti i campioni contenuti negli array grazie all'utilizzo di due cicli *for* (la denominazione delle variabili righe e colonne fa intendere che sono stati considerati come matrici) e dando i valori uno ad uno in ingresso ai tre blocchi sommatori il cui funzionamento è simulato da tre variabili:

```
//righe
for(int riga=0;riga<124; riga++){
//colonne
for(int colonna=0; colonna<8; colonna++) {
     //Out 3° blocco
     cnbis_4=(cnbis_4+cn2_4);
     //Out 2° blocco
     cn2_4=(cn2_4+cn1_4);
     //Out 1°  blocco
     cn1_4=(cn1_4+delta1_4);
     //dato IN
     delta1_4=CORE1_RE[riga] & (0x01<<colonna);
}
}
```

- segue poi la cascata dei tre blocchi differenziatori i quali però operano alla frequenza f_s/OSR per ciò teniamo traccia del valore di OSR e abilitiamo la catena solamente quando si è raggiunto il fattore desirato, in questo caso OSR=200 (scelta progettuale):

```
if(osr_4>200){           //raggiunto il valore di OSR?
     dn5_4=cn4_4;        //blocco derivatori
```

```
        dn3_4=cn3_4;
        dn1_4=dn0_4;
        dn0_4=cnbis_4;
        cn3_4=(dn0_4 - dn1_4);
        cn4_4=(cn3_4 - dn3_4);
        cn5_4=(cn4_4 - dn5_4);
        osr_4=0;
    }else{
        osr_4++;}       //incremento il conteggio per OSR
    }
```

La scelta di eseguire tale procedura a seguito del campionamento dei dati scorrendo tutti i campioni copiati negli array anziché effettuare il filtraggio real-time è dettata dalla volontà di limitare i consumi del chip Boro, lo si abilita solo per il periodo di campionamento e non appena i dati dai relativi core sono stati acquisiti lo si disabilita.

Scrittura in memoria

La memoria S25FL164K supporta due moduli SPI: il modulo 0 (0,0) e il modulo 3 (1,1); per il progetto in questione utilizziamo lo 0. La differenza principale riguarda il clock utilizzato per la comunicazione: nella modalità 0 adottata il clock nello stato di idle è a valore logico basso e la trasmissione avviene sul fronte di salita.

La scrittura dei dati avviene come ultima istruzione del programma, a seguito del completo campionamento dei dati, dopo aver provveduto a disabilitare gli interrupt delle periferiche, disabilitato l'interfaccia impedenziometrica e filtrati i dati acquisiti. La funzione *void Memory(void)* invocata nel main program contiene la sequenza di istruzioni da eseguire per la scrittura in memoria.

Per soddisfare la necessità di eseguire una scrittura veloce evitando di mantenere accesa la memoria per un periodo di tempo eccessivo, utilizziamo una frequenza di clock maggiore rispetto alla velocità di configurazione di Boro. La massima utilizzabile dal modulo SPI è data da: $\frac{F_{OSC}}{4} = \frac{8\ MHz}{4} = 2\ MHz$. Basterà quindi modificare i bit SSMP del registro SSCON1 impostando la frequenza di clock a Fosc/4: `SSPCON1bits.SSMP=0b0000`.

La configurazione del modulo SPI è la prima ad essere eseguita ed operando su registri imposto:

- MODE_00: CKE=1 e CKP=0, questa modalità di funzionamento imposta lo stato di idle del clock a valore logico basso e comunica che la trasmissione avviene sul fronte di salita del segnale di clock.
- SPI_FOSC_4: imposto il microcontrollore come il master della comunicazione SPI e imposto la frequenza di clock alla frequenza massima ovvero Fosc/4=2MHz.

```
SSPCON1bits.SSPM = 0b0001;  // SPI Master mode e clock = FOSC/4
(con 8MHz= 2MHz)

SSPCON1bits.CKP = 0;        //Stato di idle del clock livello
logico basso
SSPSTATbits.CKE = 0;        //Trasmissione sul fronte di salita
del clock
```

Con queste istruzioni imposto il microcontrollore come Master nella comunicazione e la frequenza di scrittura a 2 MHz, imposto lo stato di idle a livello logico basso e la trasmissione dei dati sul fronte di salita del clock.

Ogni istruzione inviata alla memoria dev'essere preceduta dall'attivazione del CS settandolo a valore logico basso, il quale deve poi essere riportato a valore logico alto non appena l'invio risulta ultimato [8]. Ogni comunicazione pertanto sarà preceduta e poi seguita da un'attivazione e una disattivazione del CS ottenuta operando sul registro del pin utilizzato per questa necessità: registro LATD0 del pin RD0.

In merito al risparmio energetico la memoria in esame vanta una modalità di standby particolare: la *Deep-Power-Down*. In questa fase l'assorbimento di corrente scende fino ad un valore di 2 µA, riducendo enormemente il normale assorbimento di corrente in stand-by di 15 µA. Risulta quindi indispensabile a termine del ciclo di salvataggio impostare questa modalità di risparmio energetico. L'abilitazione di questa fase, così come altre istruzioni, avviene attraverso l'invio di comandi specifici alla memoria che permettono di operare sui registri interni [8]. In Figura 25 e Figura 26 vengono riassunti tali comandi e la loro relativa codifica in codice esadecimale. Notiamo che il

comando in grado di portare la memoria nella modalità desiderata è denominato *Deep Power-down* ed è codificato con il valore 0xB9, mentre l'unica istruzione in grado di riabilitare la memoria è l'istruzione *Release Power Down* codificata con il valore 0xAB. Per poter eseguire entrambi i comandi basterà inviare alla memoria il valore esadecimale corrispondente al comando da eseguire, preceduto e seguito rispettivamente dall'attivazione e disattivazione del CS.

Command Name	BYTE 1 (Instruction)	BYTE 2	BYTE 3	BYTE 4	BYTE 5	BYTE 6
Deep Power-down	B9h					
Release Power down / Device ID	ABh	dummy	dummy	dummy	Device ID (1)	
Manufacturer/ Device ID (2)	90h	dummy	dummy	00h	Manufacturer	Device ID
JEDEC ID	9Fh	Manufacturer	Memory Type	Capacity		
Read SFDP Register	5Ah	00h	00h	A7–A0	dummy	(D7–D0, ...)
Read Security Registers (3)	48h	A23–A16	A15–A8	A7–A0	dummy	(D7–D0, ...)
Erase Security Registers (3)	44h	A23–A16	A15–A8	A7–A0		
Program Security Registers (3)	42h	A23–A16	A15–A8	A7–A0	D7–D0, ...	

Figura 25 - Comandi memoria 1 da [8]

Una volta attivata la memoria, prima di inviare i dati da salvare, è necessario abilitarne la scrittura. In Figura 26 individuiamo il comando *Write Enable* la cui istruzione in valore esadecimale risulta essere 0x06. A inizio comunicazione pertanto inviamo tale comando alla memoria per impostare l'inizio della scrittura, successivamente inviamo un byte alla volta i dati. Per l'invio dei dati alla memoria distinguiamo due casi: nel primo consideriamo i dati non filtrati e quindi la comunicazione tra microcontrollore e memoria prevede l'invio di tutti gli elementi dell'array, il secondo caso prevede l'invio dei dati filtrati. L'invio dei dati non filtrati risulta essere utile in fase di debug per monitorare il corretto salvataggio dei campioni provenienti da Boro, ed inoltre per verificare l'invio dei dati alla memoria escludendo possibili errori causati dal filtro.

Nel caso dell'invio dei dati non filtrati ci avvaliamo di cicli *for(…;…;…)* utili per scorrere tutti gli elementi di un elenco. Essendo la comunicazione SPI basata sullo scambio bit per bit dei dati contenuti nei registri di Master e Slave, è necessario attendere che la comunicazione sia avvenuta correttamente prima di procedere all'invio di nuovi dati pertanto, a seguito dell'invio

di ogni singolo Byte, inserisco un controllo sul registro SSPBUF e un ritardo della durata di 10ms.

```
LATDbits.LATD0=0;              //attivo il CS#
SSPBUF = Release;              //Invio 0xAB per attivare la memoria
while (!SSPSTATbits.BF);       //SSPBUF ancora pieno? Aspetto
Delay10KTCYx(2);               //ritardo di 10ms
LATDbits.LATD0=1;              //disattivo il CS#


LATDbits.LATD0=0;              //attivo il CS#
SSPBUF = WriteEnable;          //Invio 0x06 per attivare la memoria
while (!SSPSTATbits.BF);       //SSPBUF ancora pieno? Aspetto
Delay10KTCYx(2);               //ritardo di 10ms
LATDbits.LATD0=1;              //disattivo il CS#


LATDbits.LATD0=0;              //attivo il CS#
for(int k=0;k==124;k++){       //invio tutti i byte del
SSPBUF = CORE1_RE[k];          //Core1_re con il ciclo for
while (!SSPSTATbits.BF);}      //SSPBUF ancora pieno? Aspetto
LATDbits.LATD0=1;              //disattivo il CS#
…
…
LATDbits.LATD0=0;              //attivo il CS#
SSPBUF = DeepPowerDown;        //invio 0xB9 per la Deep Power down
while (!SSPSTATbits.BF);       //SSPBUF ancora pieno? Aspetto
LATDbits.LATD0=1;              //disattivo il CS#
```

Il ritardo è realizzato sfruttando la funzione messa a disposizione da Microchip contenuta nella libreria “delays.h”. In particolare la funzione Delay10KTCYx(2) mi permette di realizzare un ritardo della durata di 10’000*2 volte il TCY (tempo di ciclo per istruzione) che a $\frac{F_{OSC}}{4} = \frac{8\,MHz}{4} = 2\,MHz$ è di 0.5 µs e quindi: $10'000 \cdot 2 \cdot 0.5 \cdot 10^{-6} = 10\,ms$.

Nell’esempio di codice sopra riportato viene mostrato l’invio dei Byte costituenti l’array relativo alla parte reale del core1, tale costrutto è poi ripetuto per tutti gli array utilizzati.

Command Name	BYTE 1 (Instruction)	BYTE 2	BYTE 3	BYTE 4	BYTE 5	BYTE 6
Read Status Register-1	05h	SR1[7:0] (2)				
Read Status Register-2	35h	SR2[7:0] (2)				
Read Status Register-3	33h	SR3[7:0] (2)				
Write Enable	06h					
Write Enable for Volatile Status Register	50h					
Write Disable	04h					
Write Status Registers	01h	SR1[7:0]	SR2[7:0]	SR3[7:0]		
Set Burst with Wrap	77h	xxh	xxh	xxh	SR3[7:0] (3)	
Set Block / Pointer Protection	39h	A23–A16	A15–A10, x, x	xxh		
Page Program	02h	A23–A16	A15–A8	A7–A0	D7–D0	
Sector Erase (4 kB)	20h	A23–A16	A15–A8	A7–A0		
Block Erase (64 kB)	D8h	A23–A16	A15–A8	A7–A0		
Chip Erase	C7h/60h					
Erase / Program Suspend	75h					
Erase / Program Resume	7Ah					

Figura 26 - Comandi memoria 2 da [8]

Per quanto riguarda invece l'invio dei dati già filtrati il procedimento è sostanzialmente simile, questa volta invece di scorrere tutti gli elementi dell'array sarà sufficiente inviare i dati contenuti nelle variabili *dato* in uscita dal filtro.

```
SSPBUF = dato & 0xFF          //Invio i primi 8bit
while (!SSPSTATbits.BF);      //SSPBUF ancora pieno? Aspetto
Delay10KTCYx(2);              //ritardo 10msec

SSPBUF = dato & 0xFF00        //Invio i secondi 8bit
while (!SSPSTATbits.BF);      //SSPBUF ancora pieno? Aspetto
Delay10KTCYx(2);              //ritardo 10msec

SSPBUF = dato & 0XF0000       //Invio gli ultimi 4 bit
while (!SSPSTATbits.BF);      //SSPBUF ancora pieno? Aspetto
Delay10KTCYx(2);              //ritardo 10msec
```

SPI

La Serial Peripheral Interace o SPI, è un sistema che permette la comunicazione tra un dispositivo denominato *master* e uno o più dispositivi detti *slave*. Quattro sono i segnali che entrano in gioco in una comunicazione di tale tipo:

- SCLK: Serial Clock
- SDI/MISO: Serial Data Input, Master Input Slave Output
- SDO/MOSI: Serial Data Output, Master Output Slave Input
- CS/SS: Chip Select o Slave Select

La trasmissione dei dati si basa sul funzionamento di registri a scorrimento (Figura 27): ogni dispositivo, sia master che slave, è dotato di un registro a scorrimento interno dal quale i bit vengono emessi sul canale d'uscita (SDO/MOSI) e contemporaneamente immessi nel canale d'ingresso (SDI/MISO). Tale trasferimento di dati avviene su due linee fisicamente separate, una dedicata all'ingresso e una all'uscita. Nel seguente progetto di tesi si richiede al microcontrollore di comunicare con due slave: la memoria esterna alla quale vengono inviati i dati da salvare, e il chip Boro il quale invece riceve il bit stream di configurazione iniziale, entrambi abilitabili grazie al relativo Slave Select (SS/CS).

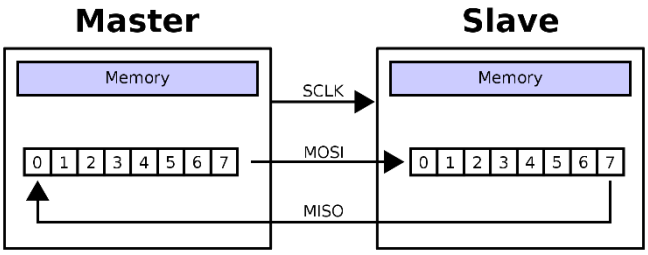


Figura 27 - Master e Slave del modulo SPI

In Figura 28 viene mostrata la struttura interna del modulo SPI nella quale sono visibili i principali registri utilizzati:

- SSPCON1: è il registro di controllo del modulo SPI, interamente leggibile e scrivibile. Tramite la configurazione di questo registro si può selezionare la modalità di funzionamento del microcontrollore (se ma-

ster o slave), la frequenza di clock e la sua polarità nonché la possibilità di utilizzare l'overflow sul byte in ricezione e il controllo di collisione.

- SSPSTAT: è il registro di stato del modulo SPI nel quale i 6 bit meno significativi sono di sola lettura, mentre solo i 2 più significativi sono sia di lettura che di scrittura. Questi ultimi due permettono di selezionare l'istante di tempo del campionamento e su quale fronte effettuarlo.
- SSPBUF: è il registro attraverso il quale il microcontrollore scrive i dati da trasmettere e legge quelli ricevuti; questo registro si affaccia al bus dati. Per caricarvi il dato da inviare è sufficiente scrivere la seguente riga di comando: SSPBUF=dato.
- SSPSR: è il registro a scorrimento (Shift Register) utilizzato per lo shift del dato in ingresso o in uscita; attraverso di esso avviene lo scambio dei dati bit per bit.

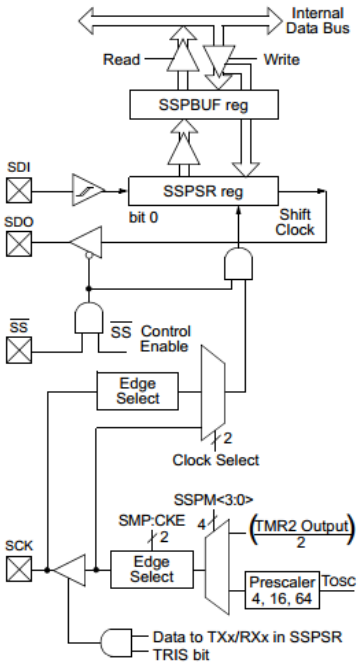


Figura 28 - Schema a blocchi del modulo SPI da [5]

Nel progetto è previsto l'utilizzo del modulo SPI per la comunicazione con due differenti slave i quali hanno necessità di frequenza di clock differenti, per tale motivo ad ogni apertura di connessione si effettua la configurazione del canale secondo le specifiche richieste.

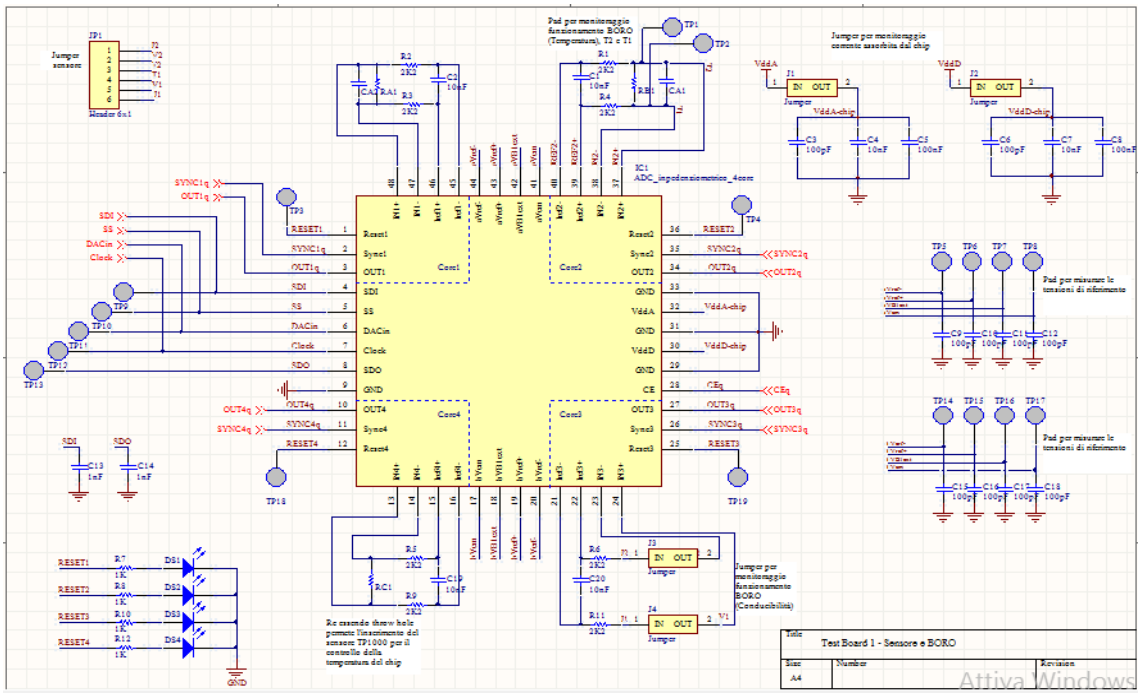
Conclusioni

Il progetto, nato dalla volontà di ottenere una scheda di test nella quale poter studiare e testare il funzionamento di tutte le componenti avendo la diretta possibilità di effettuare modifiche al firmware, è passato attraverso differenti fasi.

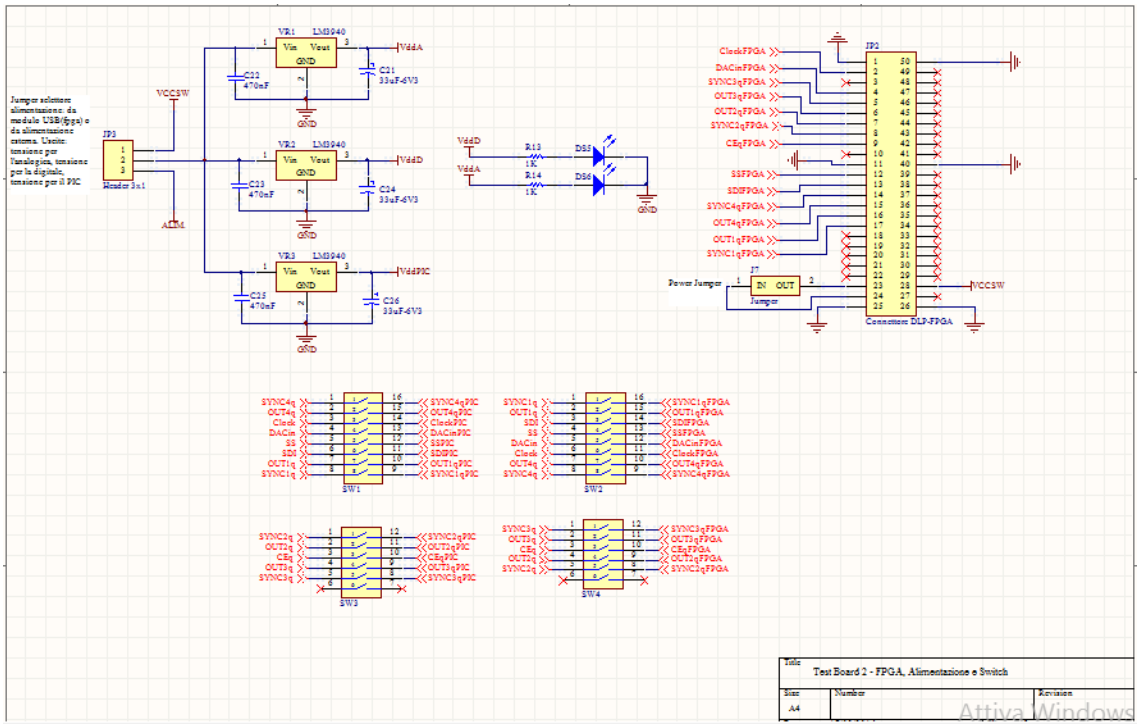
Schematici

In primo luogo è stato disegnato lo schematico del circuito con l'ausilio del programma "Altium Designer Winter"; tre tavole costituiscono l'intero progetto:

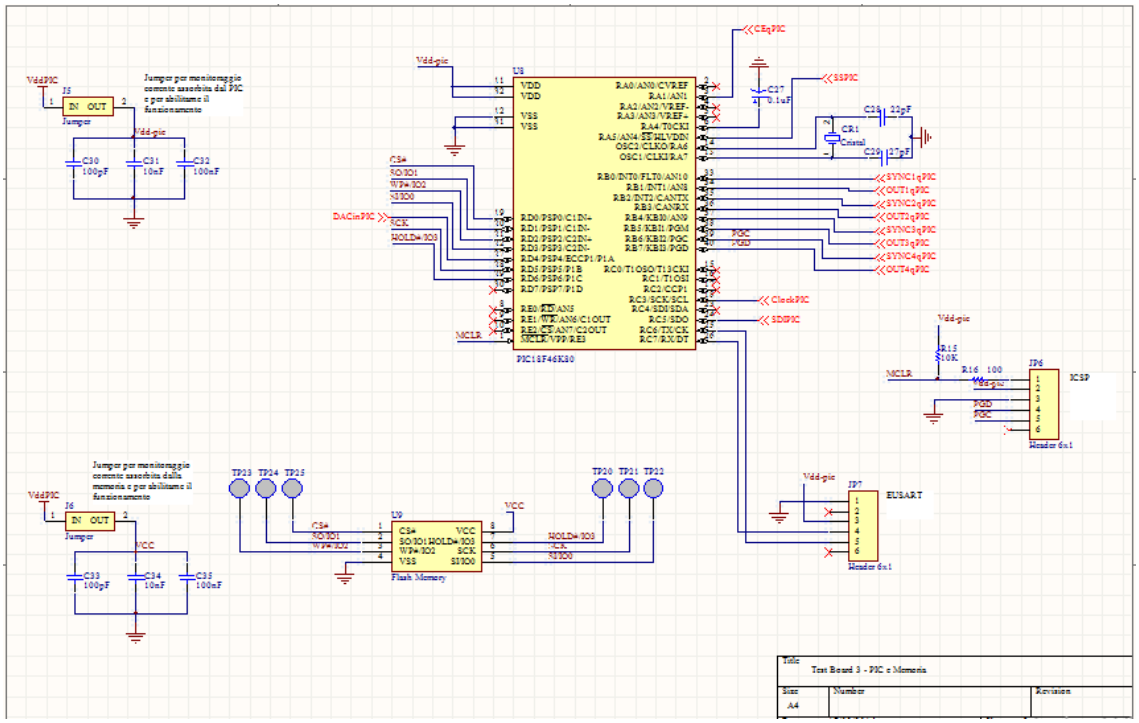
1. La prima tavola comprende il sensore e il chip Boro con annessa circuiteria di controllo come led, test-point (per prelevare agevolmente i segnali da monitorare) e jumper (usati sia per abilitare/disabilitare selettivamente dei componenti, sia per poter monitorare l'assorbimento di sezioni del circuito).



2. Nella tavola numero 2 sono presenti gli switch utilizzati per effettuare la mutua selezione di PIC/FPGA, il modulo FPGA con annessa la circuiteria per il suo funzionamento e un selettore per l'alimentazione: o tramite FPGA collegato al PC con un collegamento USB (VSCSW) o tramite alimentatore esterno (ALIM).



3. Nell’ultimo schematico vi è la parte riguardante il PIC con la necessaria circuiteria di funzionamento, la memoria e i collegamenti esterni: ICSP e EUSART.



Circuito stampato

In seguito alla progettazione del circuito elettrico ho provveduto a disegnare il layout relativo agli schematici, ponendo particolare attenzione alle dimensioni della scheda finale cercando di disporre i componenti in maniera ordinata e logica al fine di non lasciare spazi inutilizzati. E' durante la collocazione dei componenti all'interno della scheda che ho preferito l'utilizzo di switch manuali anziché analogici: in precedenza il numero superiore di collegamenti comportava una gestione delle piste più disordinata e un ingombro totale maggiore rispetto a quello ottenuto con switch manuali. In Figura 29 viene proposto il top layer del circuito finale.

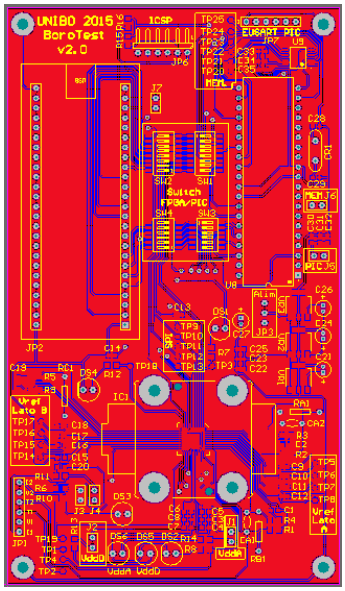


Figura 29 - Layout circuito

L'accurata gestione dello spazio ha portato ad avere un ingombro totale del circuito di 149x85 mm e come si può notare dalla Figura 30 la disposizione dei componenti e lo spazio utilizzato risulta essere efficiente. Nell'elaborato in figura è visibile un'area quadrata nella zona centrale delimitata da 4 fori contornati da aree circolari bianche: questa individua la posizione per il collocamento del socket che ci permette un utilizzo temporaneo del chip Boro. Il socket infatti assicura il posizionamento del chip QFN48 sul circuito evitando di effettuare saldature, sono necessarie solo 4 viti per assicurare la pressione e il contatto tra la scheda e il plug-in. Si è comunque proceduto saldando il chip sulla scheda dopo aver verificato la corretta presenza dei segnali provenienti dal PIC diretti a Boro ed i valori di alimentazione nominali richiesti dal chip.

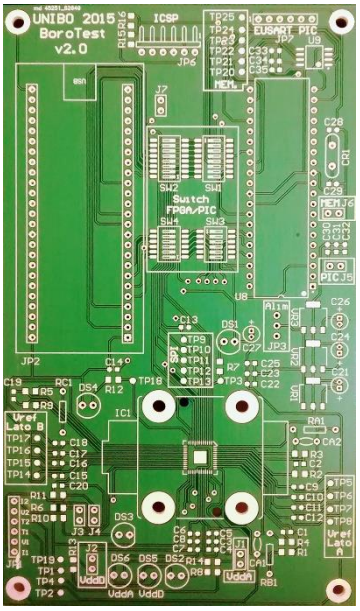


Figura 30 - Circuito stampato senza componenti

Una volta realizzato il circuito si è provveduto poi a completarlo inserendovi e saldando i componenti rimanenti; il risultato finale è visibile in Figura 31

dove vengono evidenziati i dispositivi di maggior interesse. Al momento dello scatto della fotografia risulta essere assente il sensore, sostituito temporaneamente da un resistore.

Prima di procedere con il collocamento del microcontrollore nel circuito si è preferito programmarlo su bread-board per permettere una più rapida fase di debugging. I componenti utilizzati nel circuito di debug comprendono solo quelli necessari al funzionamento del PIC18F per cui vi è il collegamento ICSP per la sua programmazione (grazie al Pickit 3 della Microchip), il circuito oscillante per la generazione del clock esterno e alcuni led sulle uscite d'interesse. In questa fase mi sono avvalso dell'utilizzo di led per controllare la presenza di segnali e per monitorare il corretto flusso delle operazioni seguenti:

- Controllo sul segnale di accensione/abilitazione del chip.
- Controllo sui segnali di Slave Select del chip e della memoria.
- Verifica della corretta uscita dalla fase di sleep.
- Verifica della completa acquisizione dei campioni.

E' bastato associare all'avvenimento di uno degli eventi da controllare l'accensione di un led ad esso dedicato.

Per simulare invece il bit stream di dati provenienti da Boro ho utilizzato un generatore di funzione programmato a 2 kHz, medesima frequenza di uscita dei dati dal chip, mentre per verificare il corretto salvataggio dei campioni negli array ho provveduto ad inviare alla memoria i dati non filtrati e ad osservarli con l'utilizzo di un oscilloscopio. A programmazione ultimata ho potuto installare il microcontrollore sulla scheda finale.

Sviluppi Futuri

Questo lavoro di tesi termina con l'implementazione del microcontrollore programmato nella scheda e con la verifica della presenza dei degnali inviati al chip: si verifica la corretta comunicazione delle parole di inizializzazione

del chip da parte del PIC sul canale SPI ed il corretto invio del segnale PWM che viene generato.

Sarà ora possibile implementare nel circuito il chip avviando una serie di procedure di test riguardanti il corretto funzionamento dell'intero progetto, il consumo di potenza e le prestazioni. Si avrà la facoltà di poter effettuare modifiche al firmware se ritenute necessarie al fine di verificare differenti possibilità per il raggiungimento di prestazioni migliori, al fine ultimo di ottenere un ulteriore risparmio energetico. La conclusione del progetto verrà attuata con l'implementazione del sistema realizzato per questa tesi in un formato miniaturizzato, raggiungendo così l'obiettivo principale alla base dell'intero programma. In Figura 31 abbiamo la possibilità di esaminare il circuito ottenuto dalla produzione di questa tesi di laurea.

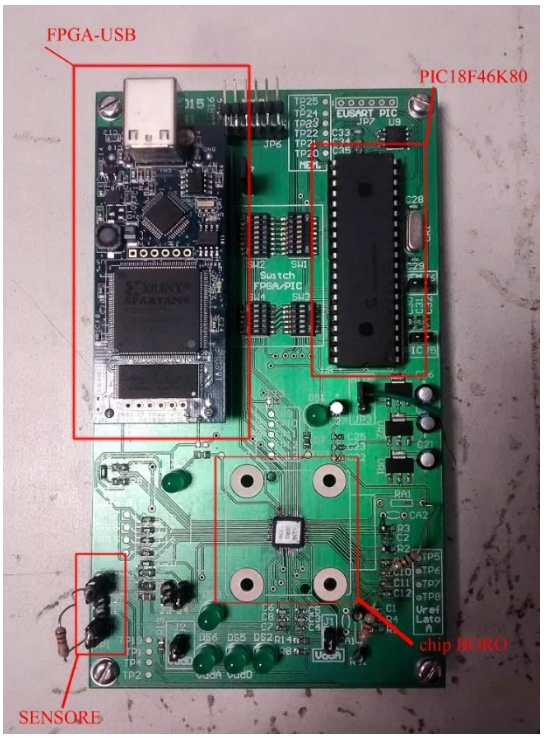


Figura 31 - Circuito finale

Bibliografia

[1] L. Martini, “Realizzazione di un sistema di misura impedenziometrico miniaturizzato ad elevata risoluzione”, Tesi di laurea, II Facoltà di Ingegneria, Cesena, 2013.

[2] M. Crescentini, M. Bennati and M. Tartagni, “A high resolution interface for Kelvin impedance sensing”, IEEE J. Solid-State Circuits, Vol.49, No.10, October 2014.

[3] Tadiran Batteries, “LTC Batteries SL-889”.

[4] IST Innovative Sensor Technology, “LFS 117 Conductivity Sensor”.

[5] Microchip Technology Inc., “PIC18F66K80 FAMILY”, 2011.

[6] M. Crescentini, M. Bennati and M. Tartagni, “Post-Chip (Boro)”.

[7] B. Baker, “How delta-sigma ADCs work, Part 1”, Analog Appication Journal, Texas Instruments Incorporated, 3Q, 2011.

[8] Spansion, “S25FL132K and S25FL164K”, September 2013.

[9] ISSI Integrated Silicon Solution Inc., “IS42S16400F IS45S16400F”, December 2011.

Indice delle Figure