POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea Specialistica in Ingegneria dell'Automazione

Sviluppo di un sistema di controllo per un modellino di treno con sistema di comunicazione wireless

---

Relatore: Prof. Francesco Castelli Dezza

Elaborato di Laurea Specialistica di:

Marco Baroni,
matr. 750553

Andrea Molinero,
matr. 746974

Anno Accademico 2011 – 2012

Indice

Indice delle figure

Indice delle tabelle

Indice degli allegati

Abstract

Scopo del progetto *TRNO1A* è controllare, tramite una comunicazione wireless, un trenino elettrico da modellismo (scala *HO*) imponendo una legge di velocità e comandando alcuni accessori come le luci e il suono del locomotore. Più nello specifico si tratta quindi di programmare un microcontrollore *PIC32* della *Microchip*, montato su una scheda creata ad hoc per tale progetto, in maniera tale che, a fronte di comandi impartiti dall'utente, possa regolare la velocità, l'illuminazione e i comandi sonori del trenino.

La scheda per il controllo dispone di un ponte ad H, una serie di convertitori analogico-digitali (*ADC*), dei sensori di corrente e tensione e un trasmettitore/ricevitore *MRF24J40MA*.

Per quanto riguarda la logica di controllo si tratta di un classico esempio di controllo in cascata, con l'anello più esterno che controlla la velocità lineare del mezzo (e quindi quella angolare del motore, opportunamente scalata) e un anello più interno che si occupa del controllo di corrente, e quindi, trattandosi di un motore DC, del controllo di coppia.

I comandi al *PIC32* vengono forniti dall'utente tramite una connessione wireless gestita tramite protocollo MiWi; in particolare saranno presenti due trasmettitori/ricevitori *MRF24J40MA*, uno, come già introdotto, saldato alla scheda posta sul trenino e collegata al motore DC, e l'altro saldato a una schedina gemella collegata tramite *USB* ad un generico *PC*. Tale schedina è fornita unicamente di Microcontrollore, trasmettitore e porta *USB*. La comunicazione fra *PC* e scheda trasmittente avviene in via seriale tramite *UART*.

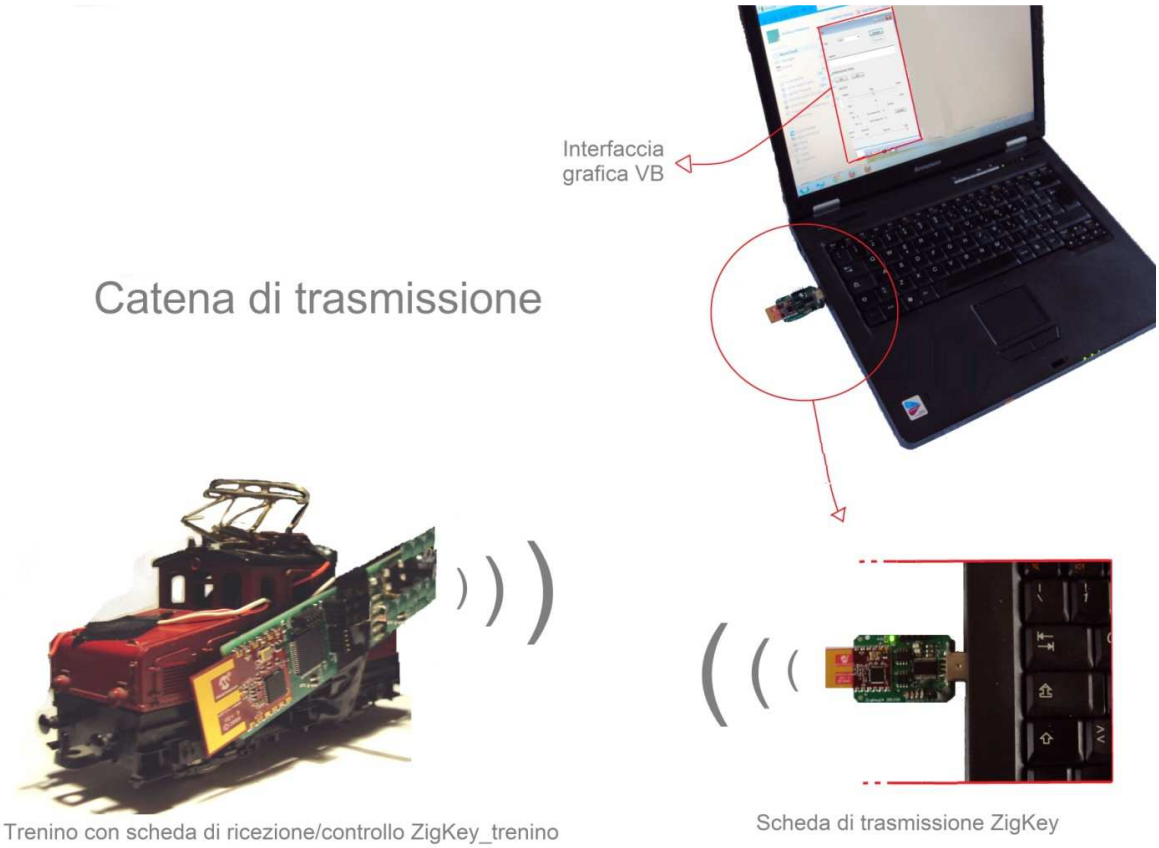


Figura 1 - Rappresentazione visiva dello scopo del progetto

Per finire viene sviluppata in ambiente *Windows* un'interfaccia utente grafica, tramite utilizzo del linguaggio *Visual Basic*, sfruttando quindi le potenzialità di un linguaggio a eventi. Lo scopo è quello di offrire al generico utente la possibilità di agire, in maniera immediata, su pulsanti e manopole per inviare comandi diretti al trenino.
Quanto è stato appena descritto è sintetizzato dalla Figura 1.

Nel corso dell'elaborato verranno discussi in primo luogo - nel Primo Capitolo - i basamenti teorici su cui si basa il nostro progetto e la sua realizzazione. Verranno ovvero mostrati gli aspetti più importanti che riguardano la natura dei dispositivi su cui andremo a operare e le soluzioni che andremo a scegliere, attraverso un'attenta ricerca nella letteratura esistente su tali argomenti.
A seguire verrà trattato - nel Secondo Capitolo - l'aspetto pratico del progetto, ovvero come siamo andati a operare per giungere alla realizzazione degli obiettivi preposti, partendo dalle simulazioni effettuate, passando per la descrizione delle azioni intraprese, per finire con la descrizione dei risultati ottenuti.

Capitolo primo 1

1.1 Introduzione

Come anticipato in questo primo capitolo ci si occuperà degli aspetti teorici alla base del progetto, attraverso un'attenta ricerca nella letteratura esistente.
Da un punto di vista degli ambiti la prima parte del capitolo sarà di ambito elettrotecnico-controllistico, nella parte centrale ci si occuperà dell'ambito elettronico per finire con aspetti di stampo più informatico.
Più nel dettaglio si comincerà quindi con una descrizione del motore DC, ovvero il motore installato nel trenino, nei suoi aspetti generali. Successivamente ci si concentrerà sulle problematiche relative all'alimentazione, presentando poi nello specifico la teoria del tipo di alimentatore utilizzato. Fatto questo verrà descritto in maniera molto approfondita tutta la tematica del controllo in velocità del motore, approfondendo in particolare le tecniche di controllo *sensorless* esistenti.
Conclusa così la parte elettrotecnico-controllistica ci si occuperà dell'aspetto elettronico, ovvero la descrizione delle schede digitali utilizzate per trasmissione e controllo. In particolare si spiegherà cosa sia e come funzioni un microcontrollore digitale e successivamente si analizzeranno tutti i componenti presenti sulle schede.
Fatto questo ci si addentrerà nella parte di stampo più informatico, conclusiva del capitolo, analizzando i protocolli di comunicazione utilizzati per la trasmissione dei dati e finendo con dei cenni sul linguaggio di programmazione *Visual Basic*.

1.2 Motore DC

1.2.1Modello ideale

Il motore a corrente continua (CC in italiano, DC in inglese) è un motore formato da un'armatura esterna fissa, o statore, su cui è posizionato un avvolgimento induttore, e un'armatura interna mobile, o rotore, su cui è disposto un avvolgimento di indotto.
Per introdurre il principio di funzionamento si fa riferimento ad una macchina esemplificativa, rappresentata in Figura 2. L'avvolgimento statorico è alimentato con una corrente continua I_e che genera un campo magnetico di asse R_e fisso nello spazio. L'avvolgimento rotorico viceversa sfrutta una particolare costruzione nota come avvolgimento di *Pacinotti*, sempre visibile in figura, tale per cui, alimentandolo con corrente continua I_i, il campo magnetico generato assume asse magnetico

R_i, ortogonale a R_e e fisso nello spazio, nonostante il movimento del rotore su cui tale avvolgimento è posto.

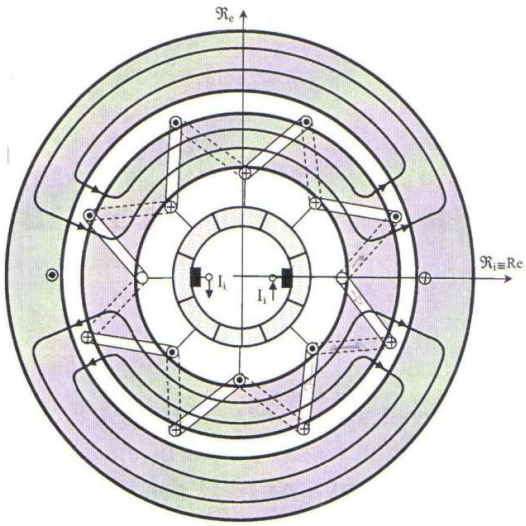


Figura 2 - motore a corrente continua ideale

Infatti si noti la presenza delle spazzole e del collettore a lamelle: le spazzole sono fisse nello spazio, mentre il collettore, suddiviso in varie lamelle, ruota sincronamente con il rotore. La corrente I_i entrante dalle spazzole si suddivide in maniera uguale fra i due lati dell'avvolgimento, che si sviluppa con la geometria mostrata in figura, determinando l'asse magnetico R_i. Quando il rotore si muove le spazzole restano fisse, mentre le lamelle, scorrendo a contatto di esse, si spostano, facendo avvenire una commutazione; a seguito di tale commutazione la corrente continuerà a entrare nell'avvolgimento negli stessi due punti geometrici di prima, e l'asse magnetico R_i rimarrà fisso nonostante il movimento del rotore. Si parla di avvolgimento pseudo-stazionario.
L'effetto dell'interazione fra i due campi magnetici fissi e ortogonali è il manifestarsi di una coppia, di modulo massimo, che pone in movimento il rotore a cui è collegato l'albero motore.
Infatti ogni conduttore di indotto di lunghezza l è sottoposto a una forza pari a:

$$F = B(\theta)\, l\, I_i$$

in cui $B(\theta)$ è il valore dell' induzione magnetica di eccitazione nella posizione θ occupata dal conduttore. Considerando la somma di tutti questi apporti si ottiene che la coppia complessiva agente sul rotore è data da:

$$C = 2UF_m\, r = 2U\left[\frac{2}{\pi} f_{em} B_m(I_e)\right] l \frac{I_i}{2} r = K(I_e) I_i$$

dove U rappresenta il numero di conduttori posti sull'avvolgimento; F_m la forza media agente sui conduttori; r il raggio dell'indotto; B_m l'induzione di eccitazione media. Il parametro $K(I_e)$ assume il nome di costante di macchina, ed è propria di ciascuna macchina a corrente continua.

A causa della presenza di questa coppia il rotore ruota e, essendo la distribuzione della corrente sull'indotto indipendente dalla rotazione, tale coppia risulta costante. Il rotore raggiungerà allora una velocità angolare costante *Ω* quando *C* equivarrà la coppia resistente del carico.
A causa della rotazione dell'indotto in ogni conduttore del suo avvolgimento si manifesta una *fem* mozionale (dovuta cioè al fatto che il conduttore si muova all'interno di un campo magnetico fisso) pari a:

$$e = blu = bl\Omega r$$

in cui *u* rappresenta il modulo della velocità lineare del conduttore. Tale *fem* si oppone alla circolazione della corrente.

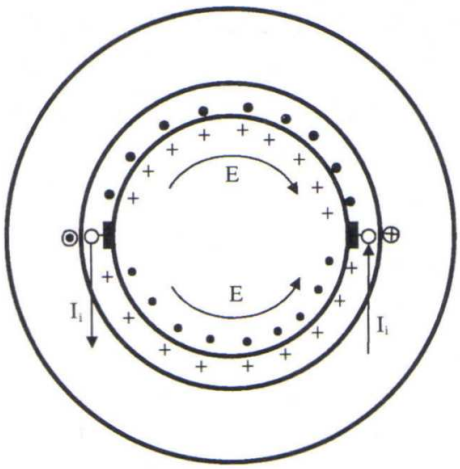


Figura 3 - comportamento della f.e.m.

Dalla figura soprastante si nota come nei due semiavvolgimenti che fan capo alle spazzole si raccoglie una *fem E* pari alla somma delle varie *fem* agenti sui singoli conduttori del polo, e il cui valore può essere ricavato dalla seguente formula:

$$E = Ue_m = U\left[\frac{2}{\pi}B_m(I_e)\right]l\Omega r = K(I_e)\Omega$$

dove *U* è il numero di conduttori sotto a un polo, ovvero il numero di conduttori di un semiavvolgimento e e_m la *fem* media manifestantesi nei singoli conduttori.

Tenendo conto delle resistenze statoriche e rotoriche, e del fatto che essendo gli assi magnetici ortogonali esiste solo la *fem mozionale*, il modello del motore a corrente continua a regime è il seguente:

$$\begin{cases} V_i = E + R_i I_i \\ V_e = R_e I_e \\ E = K(I_e)\Omega \\ C = K(I_e)I_i \end{cases}$$

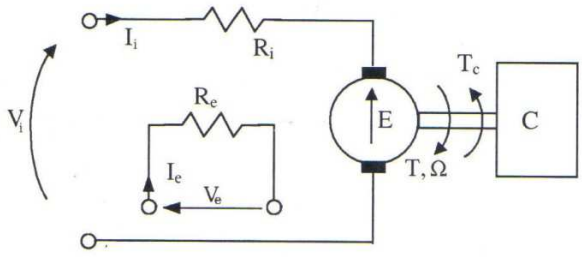


Figura 4 - modello del motore a corrente continua a regime

Come si può notare gli unici parametri di tale modello sono le due resistenze, i cui valori per un dato *range* di temperatura sono ricavati tramite misure voltamperometriche, e la costante di macchina $K(I_e)$, i cui valori si ricavano dalla caratteristica di magnetizzazione fra E e I_e, ricordando che $K = E(Ie)/\Omega$.
L'andamento tipico è il seguente, non lineare a causa delle saturazioni del materiale magnetico non ideale.

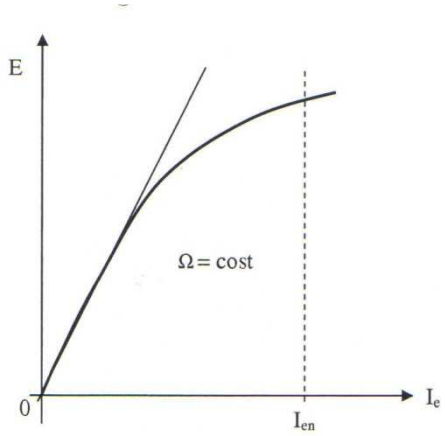


Figura 5 - caratteristica di magnetizzazione

Considerando il motore non a regime ma viceversa in fase di transitorio, ovvero con le correnti di indotto e di induttore che variano nel tempo, il modello che si ottiene è il seguente, in cui compaiono le induttanze rotoriche e statoriche, dovute appunto al fatto che le correnti non sono costanti.

$$\begin{cases} v_i = e + R_i i_i + L_i \rho i_i \\ v_e = R_e I_e + L_e \rho i_e \\ e = K(i_e)\Omega \\ C = K(i_e) i_i \end{cases}$$

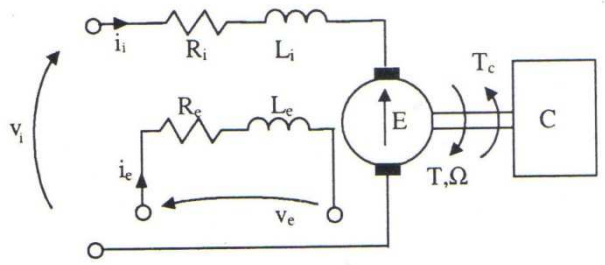


Figura 6 - modello del motore a corrente continua dinamico

dove *p* rappresenta il segno di derivata nel tempo.
Il bilancio energetico della macchina, a regime, è il seguente:

$$P_e = V_e I_e = R_e I_e^2$$
$$P_i = V_i I_i = R_i I_i^2 + E I_i$$
$$P_m = E I_i = C\Omega$$

$$P_e + P_i + P_m = \left(R_e I_e^2 + R_i I_i^2\right) + P_m$$

Potenza dissipata — Potenza meccanica

Se non si fosse a regime alle prime due equazioni andrebbero aggiunti anche i termini $(L_i \rho i_i) i_i$ - alla prima - e $(L_e \rho i_e) i_e$ - alla seconda - i quali rappresentano la variazione dell'energia accumulata nel campo magnetico statorico e rotorico.

1.2.2 Campo di operatività

Il campo di operatività a tensione e velocità variabili viene fornito per conoscere le prestazioni di una macchina a CC al variare della velocità, ed assume il seguente aspetto:

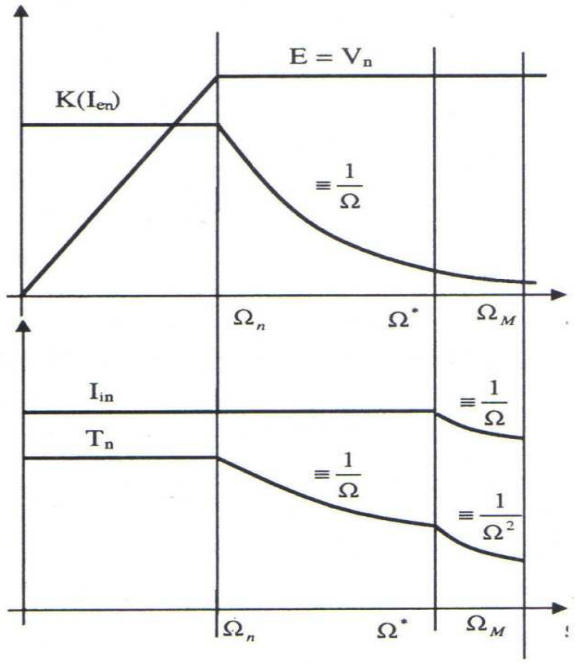


Figura 7 - campo di operatività a tensione e velocità variabili

I valori di riferimento di questo grafico sono i valori limite o i valori nominali delle grandezze che descrivono il comportamento della macchina. I valori limite sono dovuti ai problemi di surriscaldamento della macchina, che obbligano a porre delle limitazione sulle correnti, in funzione dei sistemi di raffreddamento e delle modalità operative (continuative o intermittenti).
La tensione $V_{i\,max}$ (valore limite) è definita come la massima tensione continuativa applicabile alle spazzole; la corrente $I_{i\,max}$ (valore limite) è la massima corrente continuativa che può essere fornita all'indotto in funzione dell'isolamento e del sistema di raffreddamento della macchina; la corrente I_{en} (valore nominale) è il valore della corrente di eccitazione a cui il costruttore garantisce che venga ottenuto il livello di saturazione del ferro più conveniente.
Le grandezze di cui si è fornito il valore limite possono variare da zero al valore limite stesso, e va ricordato che, per queste grandezze, ci si riferisce sempre ai loro *valori efficaci*[1]; quindi, nel caso di

[1] **Valore efficace**: il valore efficace di una data grandezza elettrica di periodo T, tensione o corrente che sia, equivale a quel valore che in regime di corrente continua svilupperebbe, per effetto *Joule*, gli stessi effetti termici. Il valore efficace si ottiene dalla seguente formula:

$$x_{eff} = \sqrt{\frac{1}{T}\int_0^T x^2\, dt}$$

funzionamento intermittente, il modulo di I_i, per esempio, può superare $I_{i\ max}$, basta che I_{ieff} rimanga sempre inferiore a tale limite.
Nel caso della corrente nominale d'eccitazione, di cui si fornisce viceversa il valore nominale, non le si permette di variare da zero a I_{en} ma la si mantiene costante su tale valore in quanto andare oltre significa saturare il ferro, mentre andare al di sotto significa sotto-sfruttare la macchina.
Come si può notare nel grafico si possono distinguere due zone, la zona a coppia costante, con $0 \leq \Omega \leq \Omega_n$, e la zona a tensione costante, o di deflussaggio, con $\Omega_n < \Omega \leq \Omega_{max}$.

- *Zona a coppia costante*: in questa zona, compresa tra la velocità nulla e la velocità nominale, è possibile fornire una corrente di eccitazione costante e pari al valore I_{en} mentre la tensione V_i cresce linearmente con la velocità Ω; la corrente rotorica I_i è pari alla massima corrente continuativa ammissibile I_{in}, e la coppia C, essendo data da $C = K(i_e)i_i$, rimane costante al suo valore nominale; raggiunta la velocità nominale Ω_n, trascurando le cadute resistive, si avrà che $V_i \approx E = K(I_{en})\Omega_n = V_{in}$, ovvero la tensione raggiunge il suo valore nominale. Di qui in poi, aumentando la velocità, si entra nella zona di deflussaggio.
- *Zona a tensione costante (o deflussaggio)*: una volta raggiunta la velocità nominale la tensione V_i rimane fissa al suo valore V_{in}, quindi per aumentare la velocità ulteriormente è necessario deflussare la macchina, cosa che si ottiene abbassando la corrente di eccitazione in modo che $K(i_e)$ sia proporzionale a $1/\Omega$. Come diretto effetto si ha che anche la coppia calerà proporzionalmente a $1/\Omega$. Raggiunta la velocità Ω^* anche la corrente I_{in} viene ridotta proporzionalmente a $1/\Omega$ per favorire la commutazione nel collettore, e di conseguenza corrente di eccitazione e coppia caleranno proporzionalmente a $1/\Omega^2$. Così si proseguirà fino alla velocità massima ammissibile Ω_M.

1.2.3 Macchina reale

Nella macchina ideale i conduttori sono puntiformi, ma nella realtà essi hanno una dimensione ben definita, e necessitano quindi di uno spazio per ospitarli, di essere sorretti e isolati. Per fare questo i vari conduttori, isolati fra loro, vengono alloggiati in cave ricavate nel ferro delle due armature, fasciate da materiale isolante.
Lo statore, esternamente, assume una forma quadrata, per facilitarne l'ammaraggio. Le cave ricavate sullo statore, di numero pari ai poli della macchina, individuano poi, come si nota nella figura sottostante, i corpi polari che isolano e sostengono le bobine.
L'indotto è semplificato, e non presenta la struttura ad *Anello di Pacinotti* ideale: le bobine, diametrali, sono allocate in cave distribuite uniformemente sulla superficie esterna del rotore, i cui terminali sono saldati a due lamelle successive del collettore.
Pur essendo molto diversa dal caso ideale, dal punto di vista magnetico e operativo tale macchina si comporta esattamente come nel caso teorico; differenze esisteranno poi ovviamente a causa della non idealità dei componenti, e di effetti parassiti o di disturbo nella commutazione e fra i campi magnetici; tali effetti, soprattutto sui motori più piccoli, per i quali minore è la cura realizzativa e sono assenti specifici *datasheet*, rende lo studio del comportamento e delle

grandezze di riferimento molto difficile, e molto distante dall'idealità, ma ciononostante il principio di funzionamento rimane immutato rispetto alla teoria di *Pacinotti*.

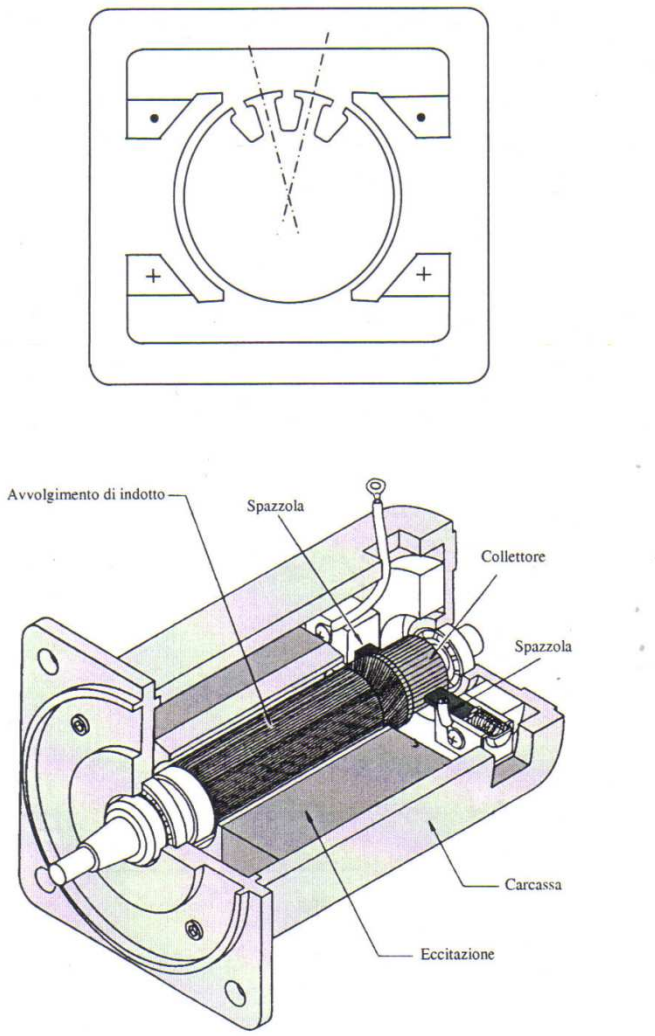


Figura 8 - motore reale

1.3 Alimentazione

1.3.1 Descrizione generale

L'alimentatore è il dispositivo in grado di fornire al motore il valore di tensione voluto, valore che viene calcolato dal regolatore per inseguire il riferimento di velocità.

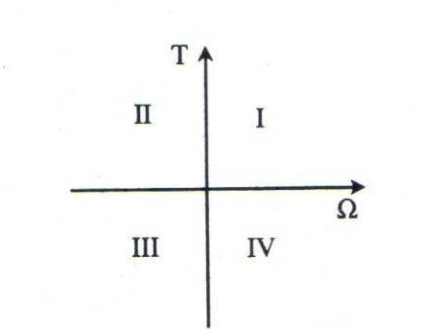


Figura 9 – andamento del segnale portante, del segnale di controllo

Dietro questa semplice affermazione si cela però una realtà più complessa. Infatti, dato un piano *C*-*Ω*, ogni azionamento è costruito per operare in un diverso numero di quadranti: tali quadranti sono visibili nella figura a lato, e rappresentano i valori ammissibili della coppia *C* al variare della velocità angolare *Ω* (o viceversa). Esistono infatti azionamenti che possono operare solo nel primo quadrante (ovvero a coppia e velocità positive, vedesi le pompe idrauliche), azionamenti che operano su due quadranti (per esempio nel primo e nel quarto, macchine per la trazione elettrica) e azionamenti che operano su tutti e quattro i quadranti (azionamenti reversibili, tipicamente si tratta di macchine utensili). Data quindi la caratteristica di un dato azionamento di operare in certi quadranti, e ricordando che per azionamento si intende il blocco *controllore-alimentatore-motore*, è l'alimentatore che assume il compito di rendere possibile tale funzionamento. Infatti, prendendo il caso della trazione elettrica, a noi più congeniale, appare evidente come l'alimentatore dovrà essere in grado di fornire energia verso il motore nella fase di trazione (e quindi a tensione e corrente positive, ricordando che alla tensione corrisponde la velocità angolare e alla corrente la coppia) e di assorbirne nella fase di frenatura a recupero (tensione positiva e corrente negativa).
Esistono quindi differenti circuiti di alimentazione che permettono di operare su un numero differente di quadranti.

1.3.2 Interruttori

Prima di entrare nel merito nei diversi convertitori DC/DC è opportuno fare una premessa sui dispositivi elettronici che ne sono alla base. Ciò che infatti permette a questi convertitori di ottenere frazioni o multipli della tensione in ingresso è l'utilizzo di interruttori abbinato a una particolare topologia circuitale. Per interruttore (o valvola) si intende un dispositivo a semi-conduttore che è in grado di condurre o non condurre a seconda di come venga comandato. Il comando può essere attivo, ovvero tramite un segnale elettrico che agisce direttamente sulla

valvola, o passivo, ovvero la commutazione avviene in base allo stato assunto dal circuito di cui fa parte. Nel caso attivo il comando della valvola viene gestito dal circuito di pilotaggio, ma è tendenza degli ultimi anni cercare il più possibile di far si che tale circuito sia già incluso nel *case* dell'alimentatore, in modo tale che dall'esterno ci sia solo un *pin* di collegamento, a cui può connettersi un microcontrollore, per comandare l'intero blocco. Ovviamente tale operazione è facilitata allorquando le potenze in gioco sono ridotte.
Nella tabella sottostante si possono vedere vari tipi di interruttori in funzione della tensione e della corrente che possono gestire, e della frequenza di commutazione.

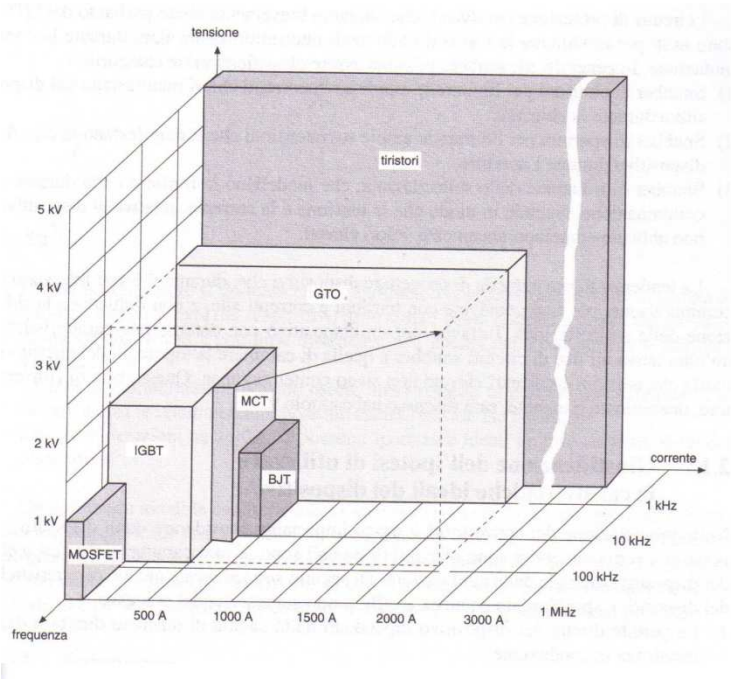


Figura 10 – rappresentazione di varie tipologie di interruttori a semi-conduttori in funzione della corrente, tensione e frequenza a cui possono operare

1.3.2.1 MOSFET

Il *MOSFET*, come si può notare, si contraddistingue fra tutti per l'elevatissima frequenza di commutazione, fino a *1 MHz*, massima fra le varie tipologie di interruttori disponibili sul mercato, al prezzo di una tensione massima di *1 kV* e una corrente di qualche decina di *Ampère*.
Tale interruttore, il cui nome sta per *Metal-Oxide-Semiconductor Field Effect Transistor*, è un dispositivo pilotato in tensione il cui simbolo circuitale e la cui caratteristica *v-i* sono mostrate in Figura 11.
Come si nota dalla figura esistono *MOSFET* a *canale P* e a *canale N*, ma il principio di funzionamento è lo stesso per entrambe le tipologie, solamente che le considerazioni valide per uno sono il duale di quelle valide per l'altro.

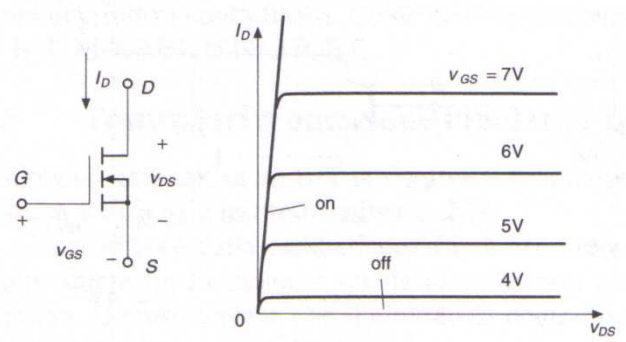


Figura 11 – simbolo e caratteristica ideale di un MOSFET

Un *MOSFET* a *canale P* è formato da uno strato di silicio drogato *N* su cui sono poste due zone di silicio drogato *P*[2], che prenderanno il nome di *source (S)* e *drain (D)*. Sulla superficie che separa *source* e *drain* viene creato un leggero strato isolante tramite ossidazione, ricoperto da uno strato conduttore che va a determinare l'elettrodo di comando, detto *gate (G)*. Se fra *G-S* non viene applicata nessuna tensione superiore ad una tensione detta di soglia l'interruttore risulta aperto, e non ci sarà passaggio di corrente nel canale *P-N-P*. Appena viene superata tale tensione di soglia si verifica l'inversione delle caratteristiche di conduzione nel silicio *N*, la rottura delle barriere *P-N* e quindi la circolazione di corrente nel verso *D-S*: l'interruttore è chiuso. Sarà poi sufficiente portare di nuovo a zero la tensione *G-S* per impedire nuovamente il passaggio di corrente.
I vantaggi di tale tecnologia sono un ritardo di spegnimento praticamente nullo, e quindi elevatissime frequenze di commutazione, unito all'isolamento dell'elettrodo di comando, che quindi non viene a contatto con le correnti transitanti nel canale. Per contro la capacità di blocco inverso è praticamente nulla, e quindi ogni *MOSFET* deve sempre essere posto in antiparallelo a un diodo di protezione, su cui possa scaricarsi un eventuale corrente inversa che potrebbe viceversa danneggiare l'interruttore.

1.3.3 Convertitori DC/DC e PWM

I circuiti di conversione *DC/DC* più utilizzati sono quelli mostrati nella seguente Figura 12. A sinistra in alto si trova il *chopper riduttore*: tale circuito, che utilizza un'unica valvola, è pensato per trasferire l'energia dalla sorgente al carico, senza possibilità di compiere l'operazione inversa. La tensione viene modulata aprendo/chiudendo la valvola con tempistiche diverse. Con tale convertitore il carico, ovvero il motore DC, funziona nel primo quadrante, e quindi non può essere effettuata frenatura a motore.

[2] Per silicio drogato *N* si intende del silicio a cui, al posto dei normali atomi tetravalenti, sono stati aggiunti degli atomi pentavalenti con lo scopo di aumentare gli elettroni liberi di muoversi nel reticolo cristallino; nel silicio drogato *P* invece gli atomi normali sono stati sostituiti con atomi tetravalenti, in modo tale da aumentare le lacune. Unendo due blocchi di silicio drogato si ottiene una giunzione P-N, alla base di ogni interruttore a semi-conduttore.

A destra in alto si trova il *chopper elevatore*: si tratta del circuito duale di quello precedente, in questo caso il flusso di energia va dal carico alla sorgente, e l'azionamento lavora nel quarto quadrante. In questo caso è possibile la frenatura con recupero di energia.
A sinistra in basso si trova il *convertitore a due quadranti*: è frutto dell'unione dei due precedenti circuiti, e serve per azionare il carico su due quadranti (a tal proposito si noti che le valvole son raddoppiate).
Per ultimo, a destra in basso, il *convertitore a quattro quadranti*, detto anche *full-bridge* o *ponte ad H*: con tale circuito è possibile azionare il carico in tutti i quadranti, e quindi permettergli di svolgere tutte le modalità operative possibili (dispone ovviamente di quattro differenti valvole).

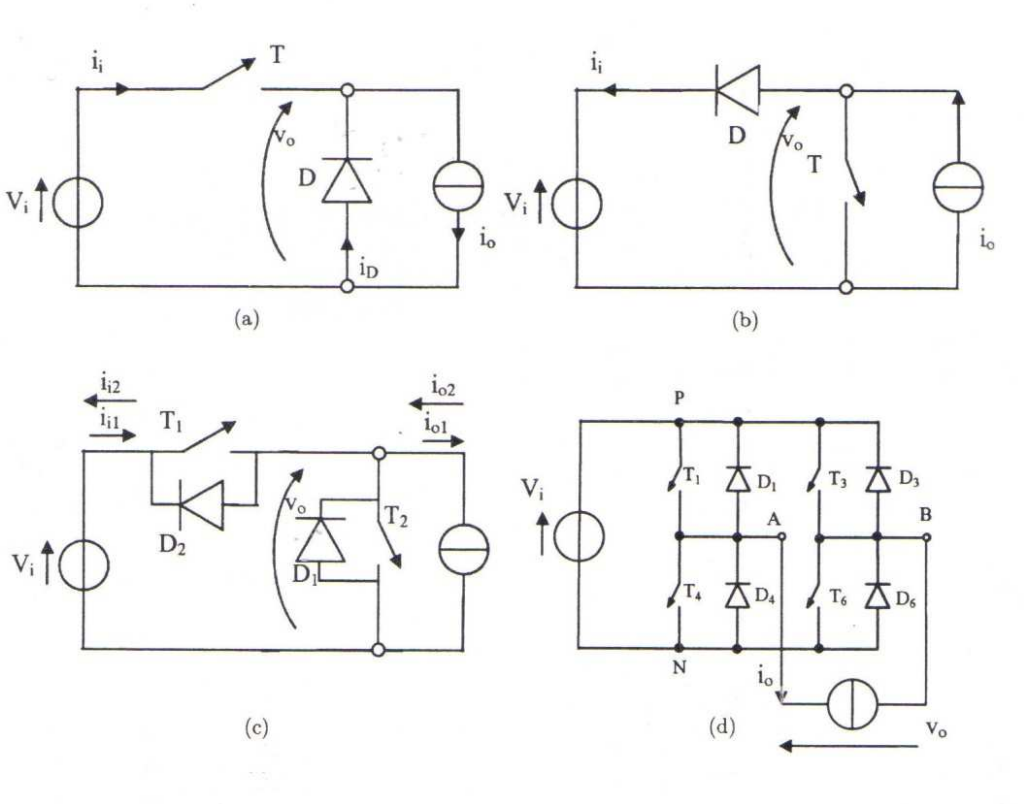


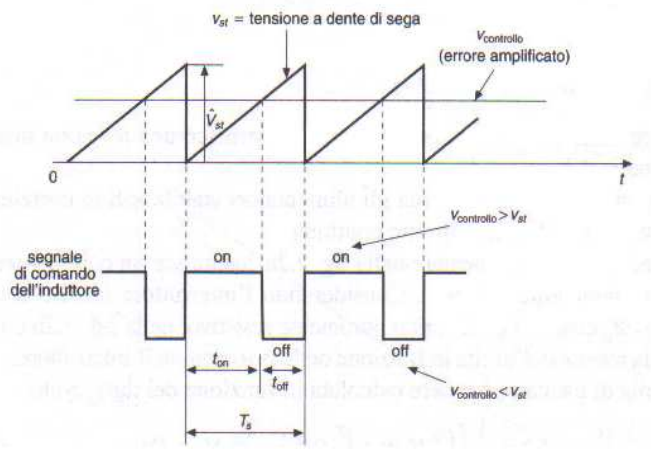
Figura 12 – rappresentazione circuitale di quattro diversi tipi di convertitori DC/DC

Quando si parla di modulare la tensione in ingresso si parla di *PWM*, *Pulse-Width Modulation*, ovvero modulazione di larghezza di impulso. Il concetto è molto semplice: ipotizziamo di avere una tensione costante in ingresso, un interruttore, e di misurare la tensione in uscita (questo esempio coincide con il funzionamento del più semplice dei convertitori DC/DC, il *chopper riduttore*); si imposti una frequenza di commutazione molto elevata, indicata con fw; se per tutto il periodo di commutazione $T=1/fw$ l'interruttore viene tenuto chiuso, in uscita avrò la stessa tensione dell'ingresso, mentre se è mantenuto sempre aperto avrò tensione nulla; se però per una certa frazione del periodo, indicata con t_s, chiudo la valvola, e per la restante parte, pari a T-t_s, la apro, in uscita avrò una tensione media pari a $V \cdot \delta$, dove $\delta=ts/T$ prende il nome di *duty cicle*. In altre parole modificando la durata t_s dell'apertura della valvola sono in grado di modulare l'ampiezza della tensione media in uscita. Il segnale di comando per la valvola viene generato confrontando una

tensione costante, v_{ctrl}, detta modulante, con una tensione v_{st} a forma di dente di sega, detta portante, secondo la logica mostrata di seguito.

$se\ v_{ctrl} > v_{st} \rightarrow \ interruttore\ chiuso$
$se\ v_{ctrl} < v_{st} \rightarrow \ interruttore\ aperto$

Utilizzando poi più valvole e circuiterie più complesse si potrà operare su più quadranti come introdotto precedentemente.
In particolare per l'alimentazione del nostro motore *CC* è stato scelto il convertitore a quattro quadranti. Nel prossimo paragrafo si procederà a descriverlo in maniera più approfondita.



controllo e della portante e
ando generato

1.3.3.1 Full Bridge (o Ponte ad H)

Lo schema classico del convertitore a quattro quadranti, o *full-bridge*, è quello visibile in figura. L'ingresso è una tensione costante V_d; l'uscita è la tensione v_o, pari a v_{AN}-v_{BN}, la cui componente media, V_o, è modificabile in ampiezza e segno (V_o è ovviamente costante). Come si nota dalla figura, la tensione V_o è la tensione di alimentazione del motore *CC*, rappresentato dal blocco *R-L-E*. La corrente i_o è la corrente di uscita del convertitore ed è modificabile in ampiezza e segno, in maniera indipendente dalla tensione V_o.

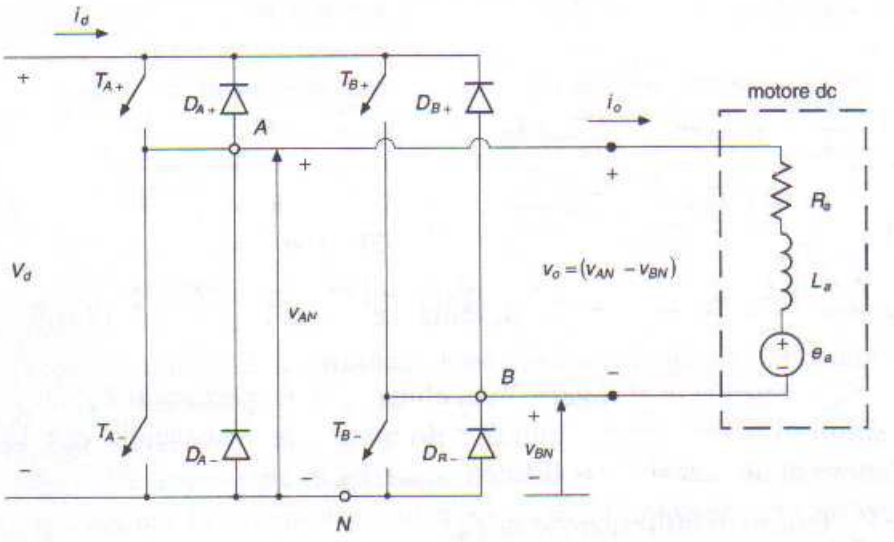


Figura 14 – Ponte-H

Il convertitore è composto da due gambe, indicate con *A* e *B*, su ognuna delle quali sono presenti due interruttori e i relativi diodi disposti in antiparallelo. Non avviene mai che, all'interno della stessa gamba, entrambi gli interruttori siano aperti nello stesso istante, ma viceversa questi agiscono aprendosi e chiudendosi in maniera complementare[3]. A seguito di ciò si deduce che la corrente i_o è priva di discontinuità, e quindi V_o dipende unicamente dallo stato degli interruttori.
Se si volesse controllare la tensione di un singola gamba, e non soltanto la tensione uscente V_o, data dalla sottrazione fra le tensioni V_{an} e V_{bn} dei singoli rami, si dovrebbero aprire entrambi gli interruttori della gamba simultaneamente, contraddicendo quanto scritto poco sopra: l'effetto, molto indesiderabile, sarebbe quello di rendere V_o dipendente dalla direzione di i_o.
Nell'esempio usato precedentemente per spiegare il concetto di *PWM*, i comandi per la valvola venivano generati confrontando una modulante con una portante a dente di sega. Viceversa, in questo caso, si userà una portante formata da un'onda triangolare simmetrica.
Due sono le strategie di *PWM* utilizzabili per lo scopo prefissato: la commutazione *PWM* con tensione bipolare e la commutazione *PWM* con tensione unipolare.

- *PWM con tensione bipolare*: in questa configurazione gli interruttori sono usati a coppie, (T_{A+} T_{B-}) e (T_{A-} T_{B+}), e all'interno di ogni coppia commutano in maniera sincrona, come fossero un'unica valvola. Le coppie di interruttori appartenenenti alla stessa gamba invece commutano in maniera complementare. I comandi alle coppie sono ottenuti confrontando una modulante, v_{mod}, con un'onda triangolare simmetrica, v_{tri}, con ampiezza di picco pari a $\hat{V}_{tri}$. Si ha che:

  $$se\ v_{mod} > v_{tri} \rightarrow\ \ T_{A+}T_{B-}\ \ sono\ chiusi \rightarrow\ \ v_{AN} = V_d,\ v_{BN} = 0 \rightarrow\ V_o =\ V_d$$
  $$se\ v_{mod} < v_{tri} \rightarrow\ \ T_{A-}T_{B+}\ \ sono\ chiusi \rightarrow\ \ v_{AN} = 0,\ v_{BN} = V_d \rightarrow\ V_o =\ -V_d$$

  Tutto ciò può essere rappresentato dalla Figura 15 in cui sono riportati gli andamenti delle grandezze appena considerate.
  Analizzando analiticamente tali andamenti si può dimostrare che la tensione media in uscita Vo, sarà legata alla modulante dal seguente rapporto:

  $$V_o = \frac{V_d}{\hat{V}_{tri}} v_{mod} = k v_{mod}$$

  Essendo *k* costante la tensione media in uscita dipenderà unicamente dal valore (e dal segno) della tensione modulante, o di controllo.
  Il nome bipolare abbinato a questa tecnica è dovuto al fatto che la tensione V_o varia istantaneamente da $+V_d$ a $-V_d$.

[3] In realtà per un breve momento, detto *tempo morto*, entrambi gli interruttori di una gamba sono aperti. Infatti, tenendo conto che gli interruttori nella realtà non sono ideali, e non possono aprire e chiudersi in un istante, se non venisse previsto tale tempo morto a ogni commutazione si potrebbe verificare un cortocircuito, causa la contemporanea chiusura di entrambi gli interruttori, che dannegerebbe tutto il convertitore. Per i MOSFET i tempi morti risultano essere di pochi microsecondi.

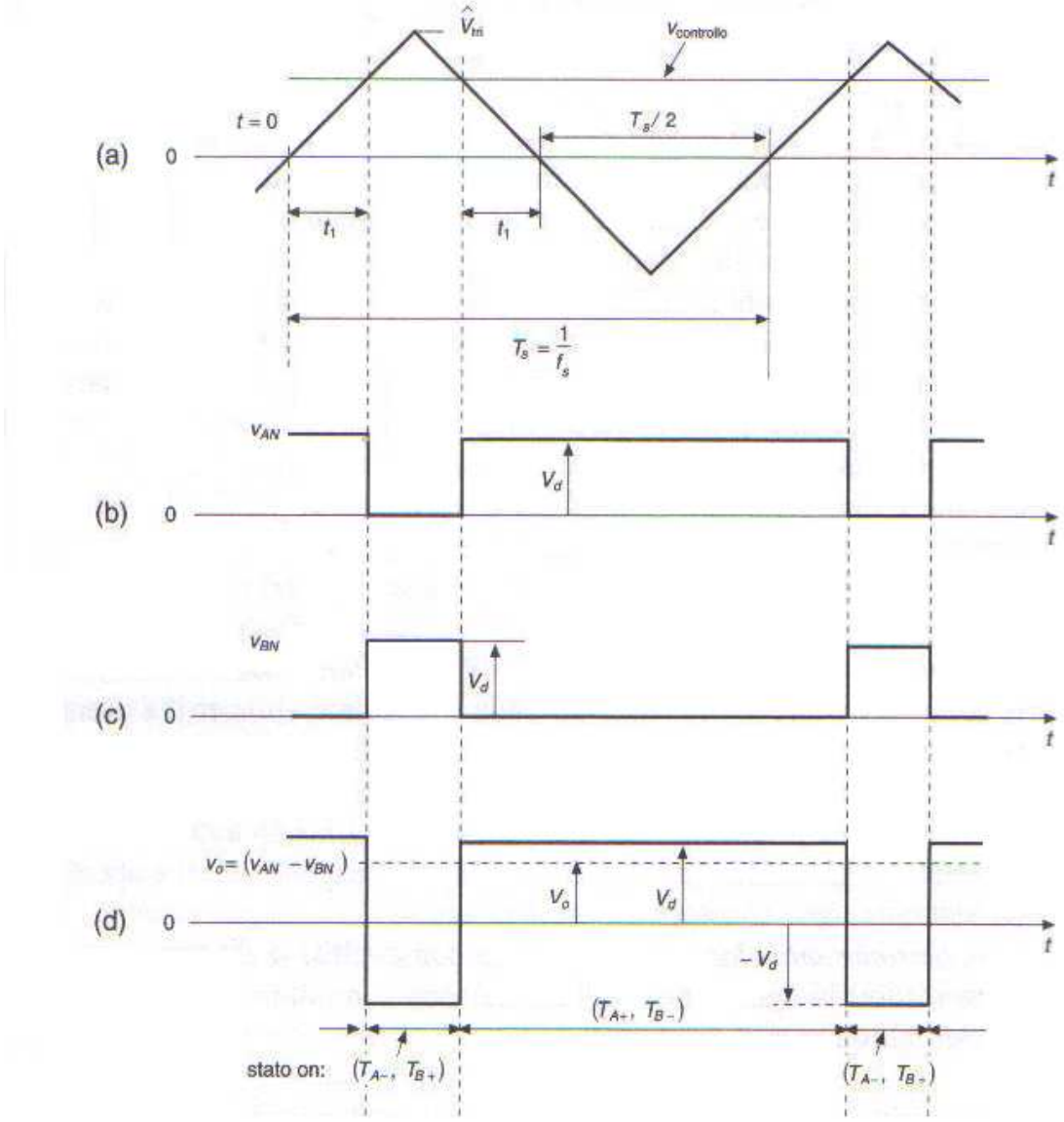


Figura 15 – Andamento del segnale di controllo e del segnale portante triangolare (a), della tensione Van (b), Vbn (c) e della tensione in uscita Vo (d) nel caso bipolare

- *PWM con tensione unipolare*: le valvole di ogni gamba sono comandate in maniera indipendente da quelle dell'altra gamba, fermo restando il vincolo, per le valvole di una stessa gamba, di commutare in maniera complementare. In questo caso l'onda triangolare simmetrica viene confrontata con le modulanti v_{mod} e $-v_{mod}$. La legge di comando delle valvole è la seguente:

$se\ v_{mod} > v_{tri} \quad \rightarrow \quad T_{A+}$ è $chiuso$
$se - v_{mod} > v_{tri} \rightarrow \quad T_{B+}$ è $chiuso$

La Figura 16 nella pagina successiva riporta gli andamenti di v_{an}, v_{bn}, e V_o.

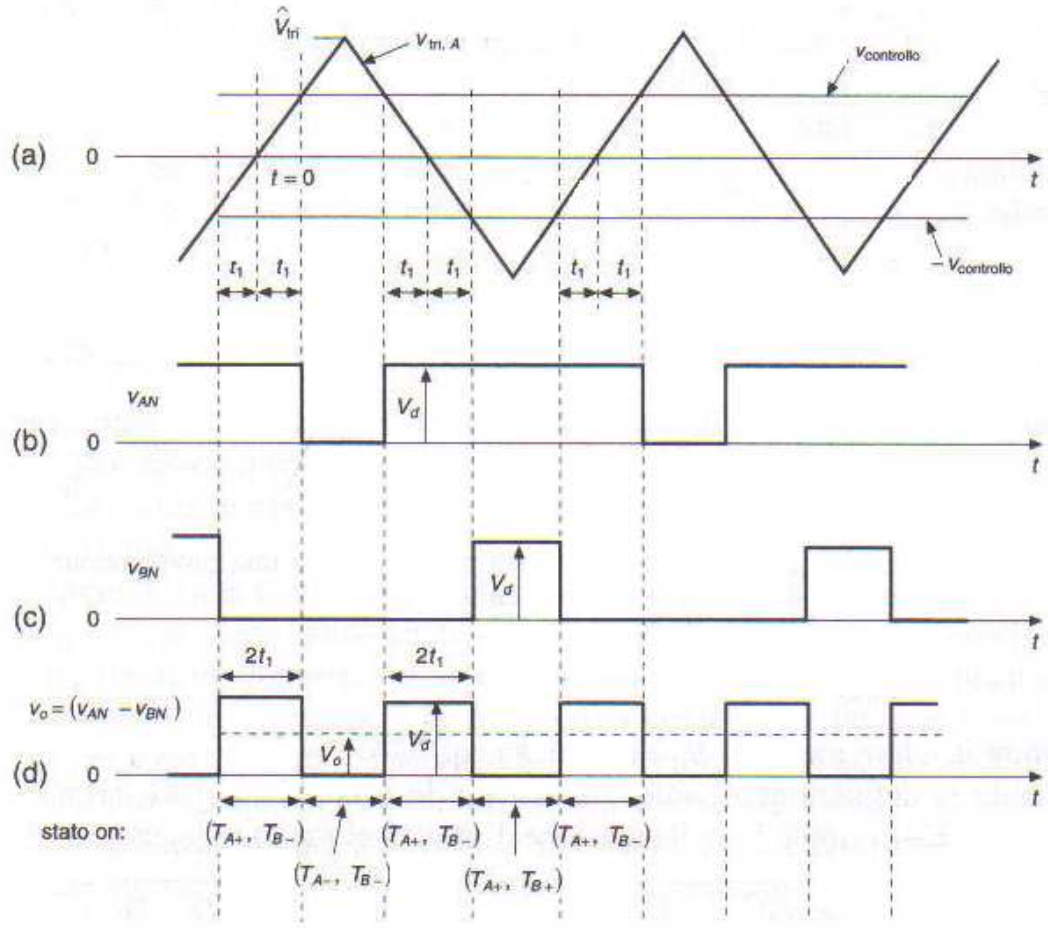


Figura 16 - Andamento del segnale di controllo e del segnale portante triangolare (a), della tensione Van (b), Vbn (c) e della tensione in uscita Vo (d) nel caso unipolare

Anche in questo caso analizzando analiticamente tali andamenti si trova che la tensione media in uscita è pari a:

$$V_o = \frac{V_d}{\hat{V}_{tri}} v_{mod} = k v_{mod}$$

A differenza del caso precedente però la tensione V_o non varia più istantaneamente da $+V_d$ a $-V_d$ ma bensì da $+V_d$ a 0, con minore stress per le valvole; un altro vantaggio di tale tecnica è che, a parità di risultato, fornisce forme d'onda migliori e una migliore risposta in frequenza, in quanto la frequenza *effettiva* di commutazione della forma d'onda della tensione d'uscita viene raddoppiata.

1.4 Controllo di velocità

1.4.1 Schema di controllo

Il controllo di velocità di un motore a corrente continua è un tipico esempio di controllo in cascata in cui vengono creati due anelli di controllo uno dentro l'altro. Il vantaggio di tale schema, raffigurato in una veste generale nell'immagine sottostante, è la possibilità, sotto precise ipotesi sulle bande passanti degli anelli di controllo, di sintetizzare il regolatore dell'anello esterno $R_2(s)$ facendo riferimento solo a $G_2(s)$, nell'ipotesi che $R_1(s)$, progettato in riferimento al solo processo interno $G_1(s)$, garantisca che $v^0 \approx v$.

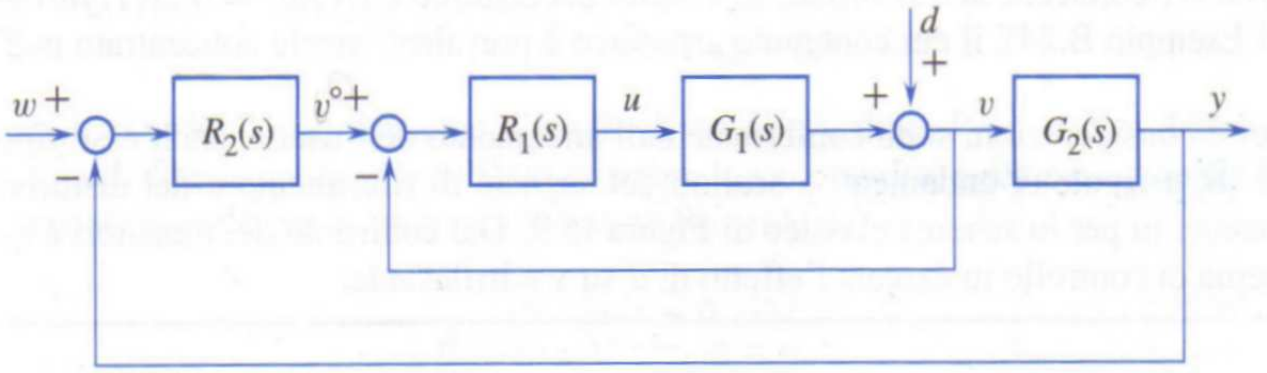


Figura 17 – schema a blocchi di un controllo in cascata

Vista la dinamica veloce legata ai transitori elettrici, si può ottenere per l'anello di controllo della corrente una banda passante molto ampia, dell'ordine delle migliaia di *rad/s*. Una volta chiuso tale anello di controllo, questo potrà ritenersi praticamente istantaneo ai fini del progetto del controllore esterno, che potrà quindi evitare di considerarlo, ottenendo così per la sua sintesi i vantaggi appena presentati.

L'anello più esterno, nel nostro caso specifico, è quello a cui viene fornito il riferimento di velocità, mentre quello interno è l'anello di controllo della della coppia. Questa configurazione è dovuta al fatto che sia che si voglia controllare il motore in velocità, sia che lo si voglia controllare in posizione, si agisce sempre sulla coppia per ottenere il profilo di velocità (o posizione) desiderato. Nello specifico, siccome i sensori di coppia sono molto onerosi e non vengono (quasi) mai utilizzati, si attua un controllo di corrente sfruttando il fatto che $C = K(i_e)i_i$.

La scelta della corrente su cui agire – corrente di indotto o corrente di eccitazione – cade sulla prima in quanto la corrente di eccitazione ha dinamiche troppo lente per permettere un controllo funzionale. Nel nostro caso, poi, la macchina è a magneti permanenti, quindi il problema non si pone.

Lo schema di controllo risultante da tali considerazione viene mostrato in Figura 18.

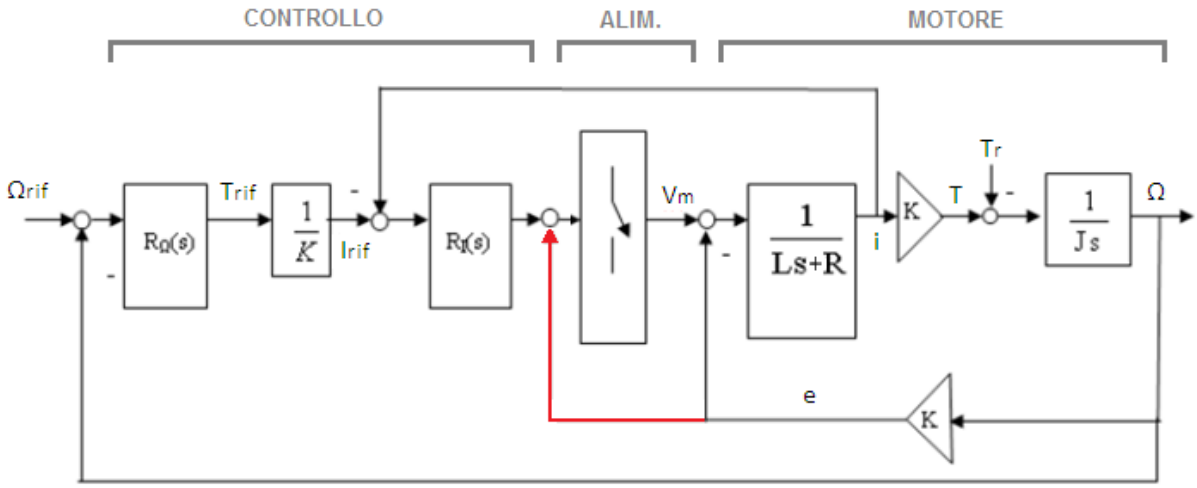


Figura 18 – schema a blocchi del controllo in velocità di un motore DC

Questo schema a blocchi è diviso in tre parti: la parte del controllo, in cui si può notare la presenza dei due anelli di regolazione precedentemente descritti, la parte dell'alimentatore e la parte rappresentante il modello della macchina a corrente continua.

- Il modello dell'alimentatore non viene specificato nel dettaglio; infatti dipende dal dispositivo che si utilizza e dall'interazione che ha col blocco motore-controllo, e può essere ottenuto solo tramite test sperimentali. Da tali test, oltretutto, si può dedurre che la dinamica dell'alimentatore sia così in alta frequenza da poter essere ignorata; se poi il guadagno risulta unitario, allora il contributo dell'alimentatore diviene irrilevante all'interno dello schema.
- Il modello della macchina a corrente continua si ottiene partendo dalle seguenti, e note, equazioni:

$$\begin{cases} C - C_r = J\Omega_m \\ C = K(i_e)\Omega_m \\ v_i = R_i i_i + L_i \rho i_i + E \end{cases}$$

Da cui, spostandosi nel dominio della trasformata di Fourier:

$$\frac{v_i - E}{R_i + sL_i} = i_i \quad \rightarrow \quad \frac{u_i}{R_i + sL_i} = i_i$$

- Per quanto riguarda la parte del controllo le tematiche più importanti sono l'utilizzo o meno della compensazione della fem, la possibilità di implementare un controllo sensorless (privo ovvero di un sensore che misuri direttamente la velocità del motore) e la sintesi dei due regolatori.

1.4.2 Compensazione della *fem*

Nello schema soprastante è stato indicato con colore rosso il contributo di *fem* che permette di ottenere un controllo a compensazione di *fem*. Il vantaggio di tale controllo è di permettere la partenza 'al volo' della macchina. Infatti, in sua assenza, se il regolatore si attivasse all'accensione del sistema quando la macchina è già in movimento, si avrebbe $E \gg 0$ e quindi $v_i - E < 0$: la corrente imposta dal regolatore risulterebbe negativa e come effetto il motore (che per esempio poteva trovarsi in una condizione di discesa) frenerebbe di colpo.
Se elimino la compensazione della fem il controllo funziona ugualmente, si deve ovviamente evitare la partenza 'al volo', non più possibile, e trattare *E* come un disturbo addittivo, sperando che non nasconda in sé *'strane'* dinamiche.
Dal punto di vista della sintesi dei regolatori i due casi precedentemente descritti possono essere così condensati:

- Con compensazione della *fem*: sintetizzo i regolatori ignorando *E*;
- Senza compensazione di *fem*: sintetizzo i regolatori in modo tale da rigettare *E*, quindi con banda passante elevata.

1.4.3 Controllo sensorless

Nello schema di controllo presentato appare evidente come sia basilare conoscere la velocità del motore, per potere così generare, tramite confronto con la velocità richiesta, l'errore da fornire al regolatore. L'altra misura necessaria è quella della corrente, utilizzata nell'anello di controllo interno per regolare la coppia.
Per misurare la corrente di indotto si inserisce una resistenza di *shunt* sulla linea di alimentazione e se ne ricava il valore misurando la tensione ai capi di tale resistenza, che è nota. Tale sensore è molto economico e resistente, per cui il suo utilizzo non desta problemi.
Per la misura della velocità del motore viceversa si possono usare encoder ottici, sensori a effetto Hall, dinamo tachimetriche[4]. L'utilizzo di questi sensori presenta però vari problemi, specialmente per motori di piccola potenza: costi maggiori di produzione, maggiore complessità circuitale e la possibilità, statisticamente sempre presente, che il sensore, come ogni dispositivo elettrico, si guasti e debba essere cambiato. Per piccoli azionamenti a basso costo tutto ciò rappresenta un uso di risorse sovradimensionato.

[4] **Encoder ottici**: utilizzano un fascio luminoso che colpisce un fotodiodo attraverso delle fenditure ricavate sul corpo del rotore. A seconda della frequenza di attivazione del fotodiodo si ricava la velocità angolare.
Sensori a effetto Hall: è un trasduttore che varia la sua tensione in uscita in risposta a un campo magnetico variabile.
Dinamo tachimetriche: sono formate da una dinamo accoppiata meccanicamente all'albero rotante di cui si vuole misurare la velocità di rotazione. La dinamo fornisce in uscita una tensione proporzionale alla velocità angolare del rotore.

Si possono quindi utilizzare tecniche sensorless in cui la velocità del motore è stimata tramite diversi artifici ma senza utilizzare sensori di velocità, ottenendo così un grosso risparmio, oltre alla sicurezza dettata dal concetto *'ciò che non c'è non può guastarsi'.*
Le tecniche di controllo sensorless che verranno presentate fanno riferimento sempre allo schema di controllo introdotto all'inizio di questo capitolo (a parte nel caso delle reti neurali che rappresentano un genere di controllo a sé stante); di conseguenza la regolazione risulta sempre essere una combinazione di *P* e *PI* all'interno di un controllo a cascata, mentre ciò che cambia di volta in volta è come viene ottenuta la misura di velocità. Analizzando la questione da questo punto di vista le tecniche sensorless si dividono in quelle basate sul modello del motore a corrente continua e quelle basate sulla componente di *ripple* della corrente di rotore.

1.4.3.1 Controllo sensorless con stima della velocità basata sul modello del motore CC

Tali metodi di controllo sfruttano, per la stima della velocità, i legami che il modello del motore determina fra i vari parametri della macchina. E' possibile distinguere fra i seguenti casi:

- Calcolo della velocità a partire dalla misura della *fem*;
- Calcolo della velocità a partire dalla stima della *fem*;
- Stima della velocità tramite l'utilizzo del filtro di Kalman;
- Controllo e stima della velocità tramite l'utilizzo di reti neurali.

1.4.3.1.1 Calcolo della velocità a partire dalla misura della fem

Per il calcolo della velocità Ω viene utilizzata la *fem*, in quanto tramite la relazione $E=K(I_e)\Omega$ è possibile, nota la caratteristica di $K(I_e)$, ricavare il valore di Ω a partire dal valore della forza elettromotrice. La *fem* viene misurata con l'uso di un sensore. In questo caso quindi non si ha una riduzione del numero dei sensori totali, che rimane fisso a due, ma viceversa una sostituzione del sensore di velocità, come visto problematico, con un sensore di tensione utilizzato per ricavare il valore di *E*. Il concetto di sensorless rimane valido, in senso lato, in quanto comunque viene eliminata la misura diretta di Ω.
Per misurare la *fem*, avendo a disposizione il *ponte ad H* per alimentare il motore, è possibile inserire dei partitori di tensione collegati alle estremità *a* e *b* del motore (si veda Figura 19) e ogni tot periodi di tempo aprire tutte le valvole in maniera tale da misurare sui partitori le tensioni e^+ ed e^-. La differenza fra tali valori rappresenta la tensione sul motore (rappresentato dal blocco *R-L-E*), ma essendo i_i momentaneamente a zero (tutte le valvole sono aperte) si ha che $V_{motore} = e^+ - e^- = E$.

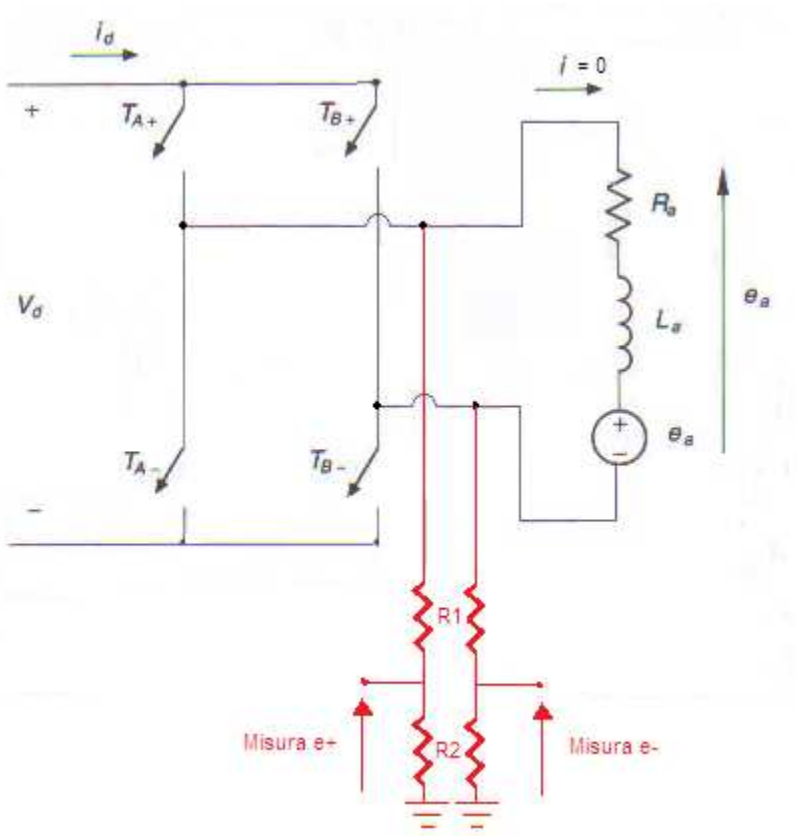


Figura 19 – Ponte-H con valvole aperte e partitori di tensione per la misura della fem

I sensori utilizzati - i partitori di tensione - sono sensori molto economici e robusti, e quindi ben si prestano per l'azionamento da noi considerato. I problemi di tale tecnica sono invece legati all'apertura periodica di tutte e quattro le valvole: in primo luogo questo implica una configurazione non standard dell'alimentatore, in secondo luogo, cosa molto più grave, tale operazione, su un azionamento così piccolo, può portare a perdite di potenza dell motore, il quale a causa dei *vuoti* introdotti potrebbe assumere un comportamento a *singhiozzo*. Oltrettutto la *fem* così ricavata potrebbe risultare, con grande probabilità, molto *sporca*, nel senso di essere affetta da molto rumore, e come tale quasi inservibile per una stima della velocità.

1.4.3.1.2 Calcolo della velocità a partire dalla stima della fem

In questo caso il calcolo della velocità angolare del motore viene effettuato sfruttando, come nel caso precedente, la relazione che lega la *fem* alla velocità Ω. Questa volta però, a differenza di prima, non si dispone di nessuna misura diretta della *fem*, ma viceversa si ottiene una sua stima direttamente dalle equazioni del modello del motore DC. Si ha infatti, a regime:

$$V_i = R_i I_i + E \;\rightarrow\; E_{stimata} = V_i - R_i I_{i\,misurata}$$

Questa stima può essere ottenuta istantaneamente in quanto V_i è la tensione che viene fornita dall'alimentatore, e quindi è nota, mentre I_i viene misurata dal sensore di corrente inserito nel sistema.

Appare ovvio che la stima vada effettuata in condizioni di regime, altrimenti occorrerebbe considerare anche il termine $L_i \rho i_i$ e quindi per ricavare *E* si dovrebbe integrare il precedente termine con tutte le complicanze computazionali e imprecisioni del caso.
Questa limitazione può costituire un problema; infatti, per disporre dei valori di regime di corrente e tensione, è necessario o disporre di una legge di controllo tale per cui le misure di *I* e *V* avvengano solo quando ogni transitorio è terminato oppure introdurre un filtro passa-basso per '*tagliare*' le componenti in alta frequenza delle misure di queste due grandezze, dovute appunto ai transitori in atto. Questo secondo metodo, per quanto permetta di effettuare misure '*continue*' senza occuparsi dell'istante in cui vengono effettuate, presenta un ovvio problema: la frequenza di taglio del filtro passa-basso impatta sulla risposta in frequenza della stima della velocità, abbassandola e degradandone pesantemente le prestazioni.

1.4.3.1.3 Stima della velocità tramite l'utilizzo del filtro di Kalman

In questo caso alla stima della velocità viene affiancata la stima della temperatura dell'armatura, in maniera tale che le variazioni della resistenza di armatura dovute alle variazioni di temperatura risultino stimabili e permettano una più accurata stima della velocità.
Per far questo è prima di tutto necessario inserire un sensore di temperatura, assente nei casi precedenti (il concetto di sensorless viene mantenuto dal fatto che in ogni caso non vengono utilizzati sensori di velocità).
Le misure della tensione e della temperatura di indotto sono utilizzate da un osservatore ad anello chiuso che include un modello del motore. Tale osservatore produce le stime di velocità, corrente e temperatura dell'indotto. La stima della temperatura viene utilizzata per raffrontarla alla misura della stessa, e generare quindi un errore della stima che viene utilizzato per correggere tutte e tre le quantità stimate usando il filtro di Kalman esteso, *EKF*. Lo schema logico è illustrato nella seguente figura:

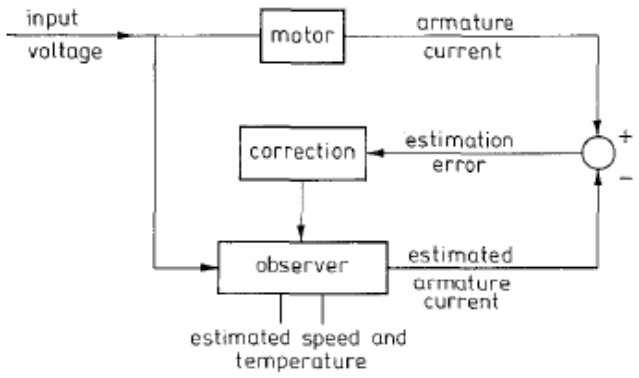


Figura 20 – osservatore per la stima della temperatura e della velocità

L'uso del filtro di *Kalman* esteso richiede un modello dinamico completo del motore, che mostri ovvero sia le dinamiche elettriche che quelle meccaniche e termiche. Si utilizza il seguente

modello, leggermente diverso da quello precedentemente introdotto, in quanto vengono considerati anche gli aspetti termici e la dipendenza della resistenza d'indotto da questi ultimi.

$$v_i = R_{i0}(1 + \alpha\theta)i_i + L_i\frac{di_i}{dt} + K(i_e)\Omega$$
$$C = K(i_e)i_i = b\Omega + J\frac{d\Omega}{dt} + C_c$$
$$R_{i0}(1 + \alpha\theta)i_i^2 + K_{fe}\Omega^2 = K_0(1 + K_t\Omega)\theta + H\frac{d\theta}{dt}$$

La prima equazione è praticamente identica al caso ideale, se non per il termine $R_{i0}(1 + \alpha\theta)i_i$, in cui R_{i0} è la resistenza dell'armatura a temperatura ambiente, ovvero circa 23°, θ la temperatura dell'avvolgimento, α il coefficiente di temperatura della resistenza, ovvero il coefficiente che lega un dato livello di temperatura a un dato incremento (o decremento) del valore della resistenza.
Nella seconda equazione viene aggiunto il termine relativo all'attrito viscoso $b\Omega$, con b costante di attrito viscoso.
La terza equazione, non presentata precedentemente, descrive la macchina dal punto di vista termico. Il significato dei suoi componenti è riassunto dal seguente schema:

$R_{i0}(1 + \alpha\theta)i_i^2 + K_{fe}\Omega^2 = K_0(1 + K_t\Omega)\theta + H\frac{d\theta}{dt}$

- $H\frac{d\theta}{dt}$ → Rappresenta il tasso di crescita della temperatura dell'armatura, che dipende dalla capacità termica del materiale.
- $K_0(1 + K_t\Omega)\theta$ → Rappresenta il flusso di potenza termica dalla superfice dell'armatura, ed è proporzionale alla differenza di temperatura tra l'armatura e l'aria dell'ambiente in cui si trova.
- $K_{fe}\Omega^2$ → Rappresenta le perdite nel ferro funzione della velocità Ω.
- $R_{i0}(1 + \alpha\theta)i_i^2$ → Rappresenta la potenza dissipata a causa dell'*effetto Joule*; il valore della resistenza dell'armatura su cui viene dissipata tale potenza dipende dalla temperatura θ per un fattore α.

Linearizzando e discretizzando il sistema sopra introdotto si ottiene il seguente sistema lineare a tempo discreto in funzione delle tre variabili di stato i_i , ω e θ:

$$i_i(k+1) = \alpha_{00}i_i(k) + \alpha_{01}\omega(k) + \alpha_{02}\theta(k) + \alpha_{03}v_i(k)$$
$$\omega(k+1) = \alpha_{10}i_i(k) + \alpha_{11}\omega(k) + \alpha_{12}\theta(k) + \alpha_{13}C_c(k)$$
$$\theta(k+1) = \alpha_{20}i_i(k) + \alpha_{21}\omega(k) + \alpha_{22}\theta(k)$$

i cui coefficienti per sinteticità sono stati rappresentati con la notazione compatta α_{mn} .

Per applicare il *filtro di Kalman esteso* è necessario considerare anche i rumori che 'sporcano' le misure e le variabili di processo, che vengono qui modellizzati come rumori bianchi a media nulla; si ottiene quindi in forma compatta il seguente modello:

$$x(k+1) = f\big(x(k), u(k)\big) + w(k)$$
$$y(k) = C(k)x(k) + v(k)$$

Dove il vettore *x(k)* <3x1> è il vettore degli stati, il vettore *u(k)* <2x1> è il vettore degli ingressi (ovvero v_i e C_c) e *w(k)* e *v(k)* sono rispettivamente vettori di dimensioni <3x1> e <1x1> rappresentanti il rumore di processo e di misura. L'uscita *y(k)* è una combinazione lineare dello stato *x(k)*.
L'osservabilità del sistema è garantita per ogni stato di equilibrio del sistema tale per cui la corrente di indotto sia diversa da zero.
L'algoritmo dell'*EKF* che permette di ottenere le stime desiderate è il seguente:

$$\hat{x}(k+1) = f\big(\hat{x}(k), u(k)\big)$$
$$P(k+1) = F(k)P(k)F^T(k) + Q \qquad \rightarrow \textit{Fase di predizione}$$

$$K(k+1) = P(K+1)C^T[CP(k+1)C^T + R]^{-1}$$
$$P(k+1/k) = P(k+1) - K(k+1)CP(k+1) \qquad \rightarrow \textit{Fase di correzione}$$
$$\hat{x}(k+1/k) = \hat{x}(k+1) + K(k+1)[i_i(k+1) - \widehat{i_i}(k+1)]$$

Laddove *F(k)* è una matrice così composta:

$$F(k) = \left.\frac{\partial f(x(k), u(k))}{\partial x}\right|_{x_n \ con\ n=1,2,3} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \\ \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_2} & \frac{\partial f_3}{\partial x_3} \end{bmatrix} = \begin{bmatrix} \alpha_{00} & \alpha_{01} & \alpha_{02} \\ \alpha_{10} & \alpha_{11} & \alpha_{12} \\ \alpha_{20} & \alpha_{21} & \alpha_{22} \end{bmatrix}$$

Tale tecnica permette un'ottima stima della velocità pur variando le condizioni operative (e quindi i parametri) della macchina e tale risultato è ottenuto senza nessun pre-filtraggio dei segnali in ingresso, evitando così il propogarsi di ritardi dovuti al filtraggio. Per contro tale tecnica necessita, dato il peso computazionale, di utilizzare dei *DSP* (*Digital Signal Processing*), i quali non hanno costi accessibili per piccoli azionamenti. Ovviamente, essendo l'elettronica un campo in cui i prezzi dei componenti diminuiscono con grandissima velocità tale affermazione non può essere considerata definitiva.
Nelle figure della pagina seguente è mostrata la struttura del filtro – Figura 21 - e una sua possibile implementazione pratica – Figura 22.

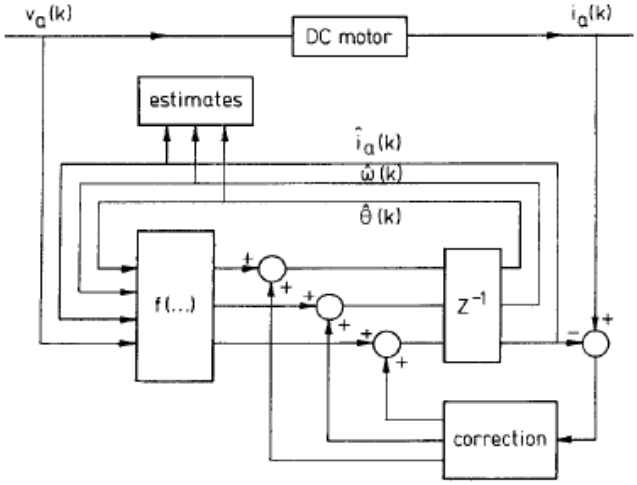


Figura 21 – struttura del filtro di Kalman esteso (EKF)

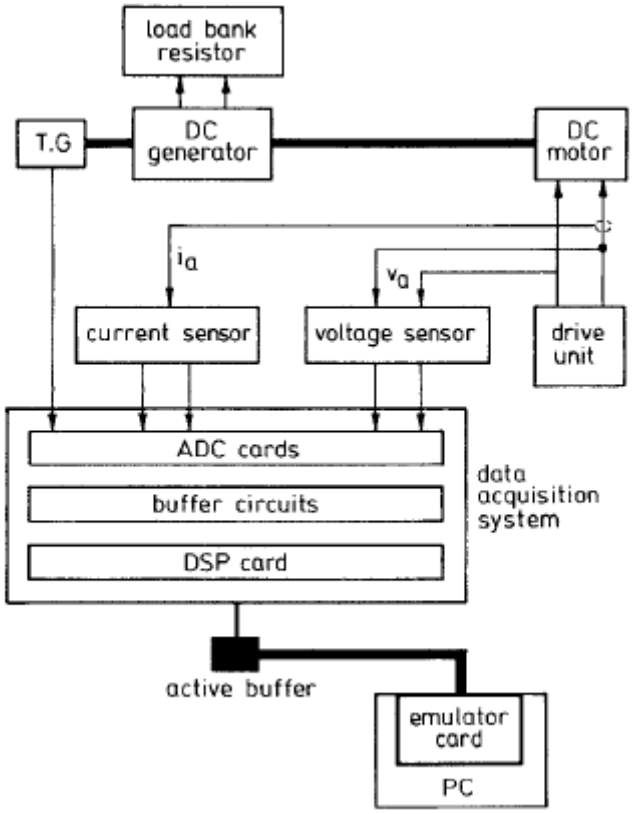


Figura 22 – possibile implementazione pratica dell'EKF

1.4.3.1.4 Controllo e stima della velocità tramite l'utilizzo di reti neurali

Le *reti neurali artificiali* (*ANN* in inglese) sono strutture non-lineari che vengono utilizzate per simulare relazioni complesse tra ingressi e uscite.

Il concetto di base della rete neurale è quello di poter essere istruita a simulare una particolare funzione, ovvero una relazione *I/O* non-lineare, sfruttando il fatto che una funzione *f(x)* generica

può essere definita come una composizione di altre funzioni *G(x)*, che possono a loro volta essere ulteriormente definite come composizione di altre funzioni. Tutto ciò può essere convenientemente rappresentato come una struttura di reti, con le frecce raffiguranti le dipendenze tra variabili.
Il loro nome dipende dalla similitudine col funzionamento dei neuroni umani: una *ANN* è formata infatti da una serie di nodi – i neuroni – collegati fra di loro in modo tale da formare una rete che abbia una serie di segnali in ingresso e un segnale di uscita; i segnali d'attivazione che giungono ad un neurone vengono trasmessi ai neuroni successivi (raggruppati in strati) attraverso archi il cui peso attenua o accentua il segnale.
L'aspetto generale di tale rete è il seguente:

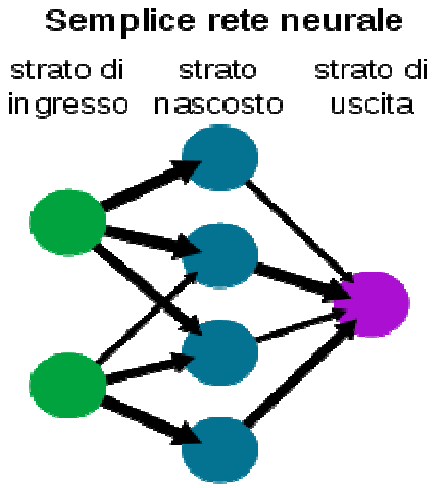


Figura 23 – aspetto generale di una rete neurale

Come si può notare la rete riceve segnali esterni su uno strato di nodi detto strato di ingresso (verde), ciascuno dei quali è collegato con numerosi nodi interni, organizzati in più livelli (strato nascosto, blu). Ogni nodo elabora i segnali ricevuti e trasmette il risultato ai nodi successivi, fino allo strato di uscita (viola), da cui viene generato il segnale in uscita.
Si parla di rete *feedforward* se il suo grafico è aciclico diretto.
La funzionalità che le rende molto interessanti dal punto di vista modellistico e controllistico è la possibilità di apprendimento, introdotta poco fa come '*capacità* (della rete) *di esser istruita*'. Il significato è il seguente: dato un compito specifico da risolvere, apprendimento significa impiegare un set di osservazioni al fine di trovare la configurazione della rete (pesi, …) che risolve il problema in modo ottimale. Ciò comporta la definizione di una funzione di costo tale che, per la soluzione ottimale, nessuna soluzione ha un costo inferiore al costo della soluzione ottimale.
In particolare si parla di *apprendimento supervisionato* (*supervised learning*) qualora si disponga di un insieme di dati per l'addestramento (*training set*) comprendente esempi tipici d'ingressi con le relative uscite. La rete è addestrata mediante un opportuno algoritmo (tipicamente e in questo particolare caso si utilizza il *backpropagation alghorythm*) il quale usa tali dati allo scopo di modificare i pesi ed altri parametri della rete stessa, in modo tale da minimizzare l'errore di previsione relativo all'insieme d'addestramento. Se l'addestramento ha successo, la rete impara a

riconoscere la relazione incognita che lega le variabili d'ingresso a quelle d'uscita, ed è quindi in grado di fare previsioni anche laddove l'uscita non è nota a priori.
Per la stima e il controllo in velocità del motore DC si utilizza il seguente modello:

$$\Omega = \frac{1}{K(i_e)}\left[v_i - R_i i_i - L_i \frac{di_i}{dt}\right]$$

$$C = K(i_e) i_i = b\Omega + J\frac{d\Omega}{dt} + C_c$$

mentre la rete neurale viene progettata a due livelli, in quanto questo garantisce un più semplice apprendimento e maggiore velocità di convergenza. Le due neuro-architetture sono chiamate *ANN1* e *ANN2*, laddove *ANN1* ha il compito di stimare la velocità mentre *ANN2* quello di controllare la tensione, ovvero fornire il segnale α di controllo del convertitore. Le due neuro-architetture sono rappresentate qui di seguito:

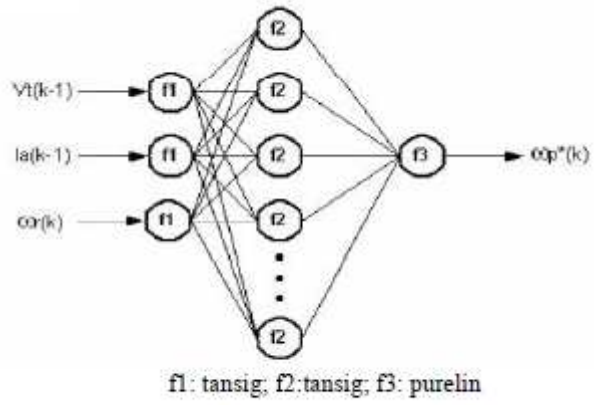


Figura 24 – neuro architettura ANN1

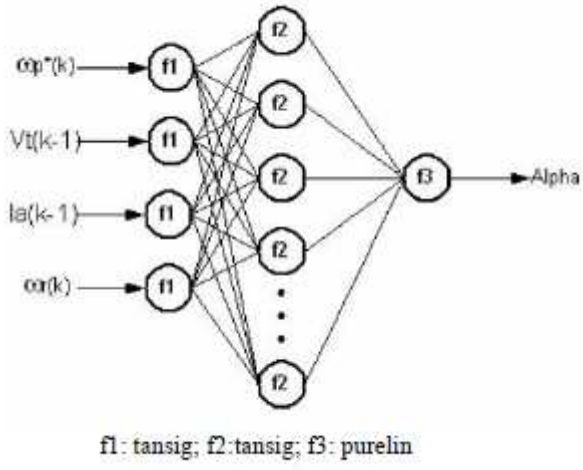


Figura 25 – neuro architettura ANN2

In una *ANN* solo il numero di ingressi e uscite è fisso, ma è regola generale limitare a un massimo di due gli strati nascosti, così come effettivamente avviene in questo caso. Purtroppo il numero di neuroni nascosti e il numero di parametri necessari all'apprendimento non dispone di una qualche regola empirica per essere stimato, e molto spesso viene scelto in base all'esperienza del programmatore.
Osservando le figure si nota che *ANN1* ha come ingressi (tutti i segnali sono discretizzati) la velocità di riferimento $\Omega_r(k)$, la tensione sul rotore $v_i(k-1)$ e la corrente di indotto $i_i(k-1)$. L'uscita è invece la stima della velocità $\Omega^*(k)$. *ANN2* invece ha $\Omega_r(k)$, $v_i(k-1)$, $i_i(k-1)$ e $\Omega^*(k)$ come ingressi e il segnale di controllo α per il convertitore come uscita.
Tramite l'apprendimento (svolto tramite l'utilizzo di *Matlab*) è possibile determinare il numero ottimo di neuroni, ovvero il minimo numero tale per cui l'errore di stima sia accettabile, e i giusti pesi dei vari archi.
La stima e il controllo di velocità ottenuti in tale modo presentano il grande vantaggio di non dover tener conto dei parametri della macchina, che non è necessario calcolare. Per contro la fase di apprendimento può essere molto lunga e complessa, con tempi neanche confrontabili col tempo di *tuning* di un normale controllore industriale, infinitamente inferiore. L'intera rete neurale ha poi il difetto di presentarsi come una *black box*, ovvero un blocco non modificabile e su cui non si può intervenire in corsa, rendendo così il sistema non adattabile ad eventuali modifiche.

In conclusione, i primi due metodi di questo gruppo hanno come vantaggio la semplicità realizzativa e il ridottissimo numero di sensori necessari, ma per contro hanno lo svantaggio di dipendere dalle grandezze del sistema - resistenze, correnti, tensioni – che a loro volta dipendono dalle condizioni operative in cui si trova a operare il motore. Tali fattori introducono un'incertezza che si propaga alla misura ottenuta della velocità.
Per ovviare a questo problema si possono usare metodi che prevedano la stima dinamica di tali parametri o la loro emulazione tramite apprendimento, ovvero gli ultimi due metodi presentati, ma, a fronte di una maggiore precisione, si paga un prezzo in termini di elevato aumento del peso computazionale e della complessità del sistema, soprattutto in relazione alla semplicità del tipo di controllo richiesto.

1.4.3.2 Controllo sensorless con stima della velocità basata sull'analisi della componente di ripple della corrente

I seguenti metodi, come introdotto, si basano sullo studio della componente di ripple della corrente rotorica e sull'andamento della *fem*. Il concetto alla base è molto semplice. Quando il motore è in movimento ogni spira dell'indotto, periodicamente, viene messa in cortocircuito da una spazzola. La situazione è quella raffigurata nella Figura 26, posta nella pagine seguente.

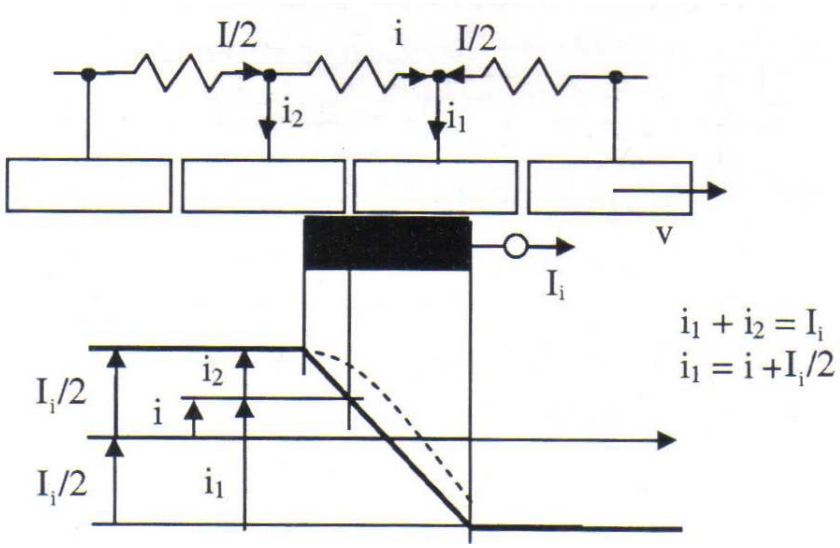


Figura 26 - comportamento delle correnti in fase di commutazione lamelle/spazzola

Ogni spira ha per estremi due lamelle attigue, e nel momento in cui viene cortocircuitata, la corrente che transita in detta spira viene forzata a passare da $\pm i_i/2$ a $\mp i_i/2$. Se non vi fossero fenomeni magnetici, ma tutta la dinamica fosse governata dalle resistenze di contatto delle spazzole, la corrente i_i si suddividerebbe tra le due lamelle a cui è connessa la spira in cortocircuito solo in funzione della superficie di contatto. Si avrebbero quindi per le correnti i_1, i_2 e i gli andamenti indicati nella figura soprastante, e i varierebbe linearmente fino ad annullarsi quando la spazzola abbandona la lamella uscente.
In realtà però esistono dei fenomeni magnetici che interferiscono con tale descrizione: in primo luogo la commutazione avviene in un punto in cui il campo magnetico generato da tutte le altre spire d'indotto è massimo, e quindi nella spira si manifesta una *fem* che tende a mantenere i, ovvero si oppone alla sua diminuzione; in secondo luogo anche la spira cortocircuitata ha una sua autoinduttanza, e quindi la corrente variante nel tempo che l'attraversa determina una *fem* trasformatorica che ha il medesimo effetto della *fem* indotta dalle altre spire.
Tali fenomeni determinano l'andamento non lineare della corrente i (linea tratteggiata in figura), che assume un valore non nullo quando la spazzola abbandona una lamella. In tale istante la corrente residua *'salterà'* sulla spazzola causando un arco elettrico di scarica.
Per quanto riguarda corrente di indotto e *fem*, tale fenomeno assume le seguenti manifestazioni.
La corrente i_i avrà due componenti: la componente continua, che è responsabile di fornire energia al motore, e una componente di *ripple*. Tale componente alternata è dovuta ai fenomeni introdotti precedentemente, e in particolare a una rettificazione non-ideale nel sistema spazzole-lamelle e alla presenza della *fem* indotta nella spira cortocircuitata. Tale corrente di *ripple* non ha un andamento sinusoidale, ma assume l'andamento mostrato in maniera dettagliata nella Figura 27.
In particolare si nota il seguente andamento periodico: una salita di durata *RWi*, una caduta a picco e una discesa moderata di durata *FWi*.

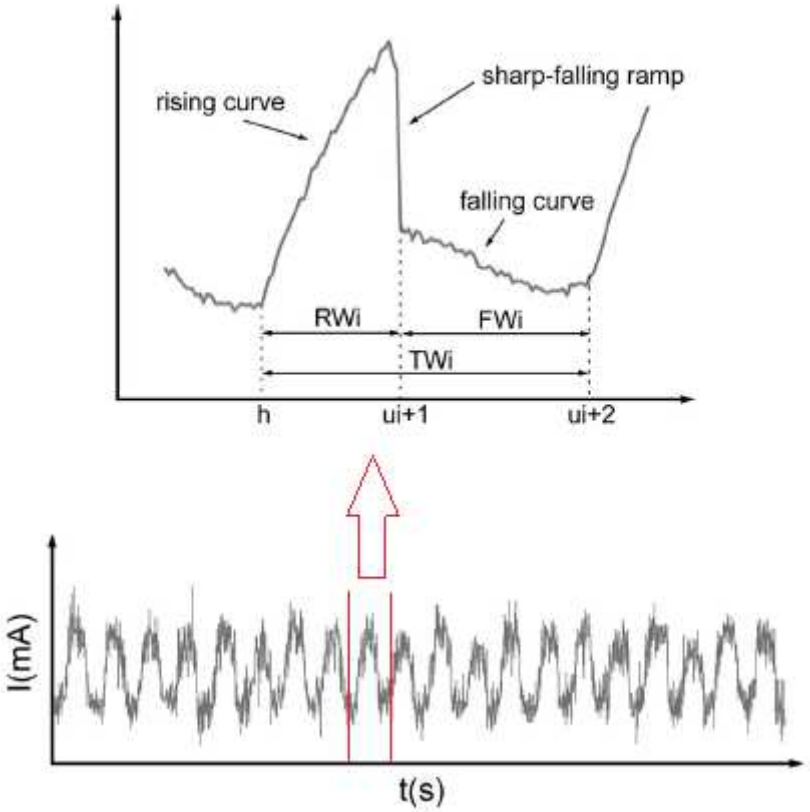


Figura 27 - andamento della corrente di ripple

Molto spesso a causa del rumore o di una banda di campionamento non adeguata tali fasi non risultano così chiaramente distinguibili, ma comunque l'andamento tende, nella sua idealità, a questa forma. Il picco della corrente di *ripple* si ha quando la spira viene cortocircuitata, ovvero quando la spazzola entra in contatto con la lamella successiva.

Per quanto riguarda la *fem*, avrà l'andamento di Figura 29, in cui si possono individuare dei picchi istantanei in corrispondenza degli istanti in cui il valore della corrente di *ripple* è maggiore e minore: ciò significa che si hanno tali picchi quando le lamelle vengono cortocircuitate.

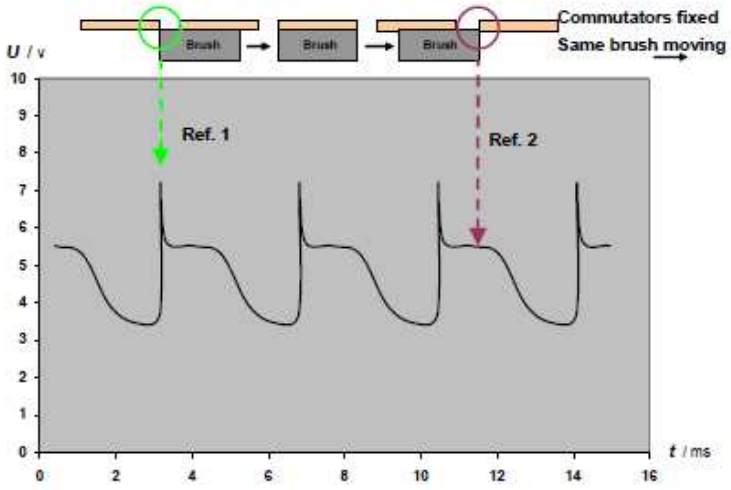


Figura 28 - andamento della f.e.m.

Di conseguenza è possibile risalire alla velocità dell'asse motore contando il numero dei picchi, nell'unità di tempo, della componente di *ripple* della corrente i_i, aiutandosi nel caso con lo studio dell'andamento della *fem*.

1.4.3.2.1 Analisi dei picchi della corrente di indotto tramite interpolazione con onda quadra

All'atto pratico uno dei metodi esistenti in teoria per analizzare i picchi della corrente di indotto è il seguente. Prima di tutto è necessario filtrare la componente di *ripple* della corrente, che ha un andamento irregolare molto affetto dal rumore e da altri disturbi. Si ottiene quindi un segnale semi-sinusoidale che tende ad assomigliare al segnale ideale così come descritto precedentemente. Da questo segnale si genera un'onda quadra avente la medesima frequenza del segnale quasi-sinusoidale, così come appare evidente nella Figura 29 (immagine sinistra).
Una volta effettuata tale operazione l'onda quadra viene fornita in ingresso al microcontrollore, che provvede a generare degli impulsi ad alta frequenza interni a ogni ciclo (o mezzo ciclo) dell'onda quadra. La frequenza di tali impulsi può essere impostata dall'esterno o automaticamente dal microcontrollore. Il treno di impulsi generato assume la forma mostrata in Figura 29 (immagine destra).

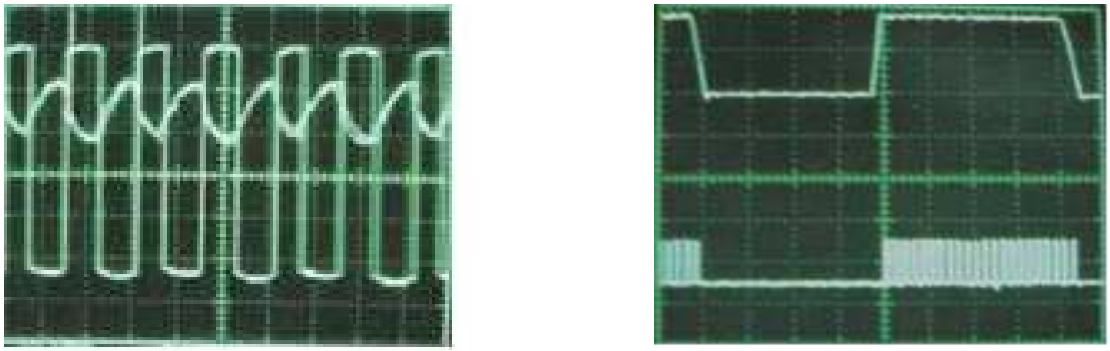


Figura 29 – generazione dell'onda quadra (a sinistra) e degli impulsi interni ad essa (a destra)

Il numero di impulsi viene contato dal microcontrollore che genera il corrispettivo numero digitale; questo numero, tramite un *DAC* (*Digital to Analog Converter*), viene portato all'esterno del microcontrollore per settare il *duty cicle* dell'alimentatore e mantenere quindi la velocità il più costante possibile rispetto al suo riferimento. Si noti che l'utilizzo di un *DAC* non è fondamentale: una volta ottenuto il numero digitale rappresentante la velocità angolare è possibile gestire tutto in ambito digitale, *duty cicle* compreso.

Per quanto riguarda il metodo appena presentato ha il grandissimo vantaggio di dipendere da un evento, la presenza dei picchi nella *fem* e nella corrente rotorica, che è direttamente proporzionale alla velocità e non è influenzato dalle condizioni operative. La limitazione all'uso di tali metodi risiede nella difficoltà, e imprecisione, nell'individuare e '*contare*' tali picchi nella pratica. Oltre a ciò si manifestano problemi pure alle basse velocità, laddove la *fem* è molto bassa.

1.4.4 Sintesi controllori PID (PID tuning)

Un controllore *PID* (*Proportional-Integral-Derivative controller*) è un meccanismo di controllo retroattivo largamente diffuso nei sistemi di controllo industriale. Questo controllore riceve in ingresso un errore dato dalla differenza tra un riferimento desiderato e la misura effettiva della variabile di processo di cui è stato fornito il riferimento, e lavora nell'ottica di ridurre al minimo tale errore *aggiustando* la variabile di controllo del processo. Il progetto di un controllore *PID* coinvolge tre distinti parametri costanti - *Kp, Ki* e *Kd* -, i quali rappresentano:

- Kp – termine proporzionale;
- Ki – termine integrale;
- Kd – termine derivativo.

La somma pesata di queste tre azioni fornisce la variabile di controllo usata per *aggiustare* il processo. Tale varibiale di controllo assume la seguente forma:

$$u(t) = K_P e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt}$$

dove *e(t)* rappresenta l'errore.
I valori dei vari parametri non possono essere dati a caso. Infatti un termine proporzionale troppo elevato può portare il sistema all'instabilità; di contro, uno troppo ridotto fornisce poca sensibilità al controllore (ad un grande errore corrisponde un azione di controllo ridotta, il che influisce anche sulla velocità di risposta) e lo rende più suscettibile ai disturbi del sistema.
Il termine integrale invece è utile per accelerare il raggiungimento del riferimento desiderato da parte del processo ed eliminare l'errore residuo a regime che si avrebbe con solo il termine proporzionale. A causa della sua caratteristica di accumulare gli errori passati può però provocare un *overshoot* rispetto al riferimento.
Il termine derivativo, infine, consente di ridurre l'*overshoot* e migliorare la stabilità del sistema controllore-processo; tuttavia ha lo svantaggio di rallentare la risposta del controllore ai cambi di riferimento e di essere molto sensibile ai rumori di misura, tanto da portare in certi casi all'instabilità del processo: per questi motivi e la conseguente difficoltà nel progettarlo viene spesso omesso (*Kd=0*).

Vengono quindi presentati tre metodi per il calcolo dei parametri *Kp, Ti* e *Td* del *PID*, ovvero della costante proporzionale, del tempo d'integrazione e di derivazione, ricordando che:

$$T_i = \frac{K_P}{K_i} \text{ e } T_d = \frac{K_d}{K_P}.$$

Questi tre metodi sono:

1. il metodo *Good Gain*, semplice metodo sperimentale che può essere usato senza alcuna conoscenza circa il processo che deve essere controllato. Se invece si dispone di un modello del processo, questo metodo può venire usato sul simulatore invece che sul processo fisico;
2. Il metodo *Skogestad*, metodo *model-based* che presuppone la disponibilità di un modello matematico del processo (funzione di trasferimento).
3. Il metodo *Ziegler-Nichols*, in anello aperto o in anello chiuso, estrapolato empiricamente attraverso numerosi esperimenti su anelli di controllo comprendenti *PID*.

Prima di vedere nel dettaglio questi metodi, ricordiamo quali sono gli scopi della sintesi di un controllore *PID*. Se possibile si vorrebbe ottenere per il sistema sotto controllo sia una robusta stabilità sia delle risposte veloci a fronte di variazioni del riferimento, ma spesso nei sistemi reali questi due desideri sono in lotta tra loro.
Di conseguenza l'obiettivo è quello di garantire una stabilità accettabile e delle velocità di risposta non troppo lente.

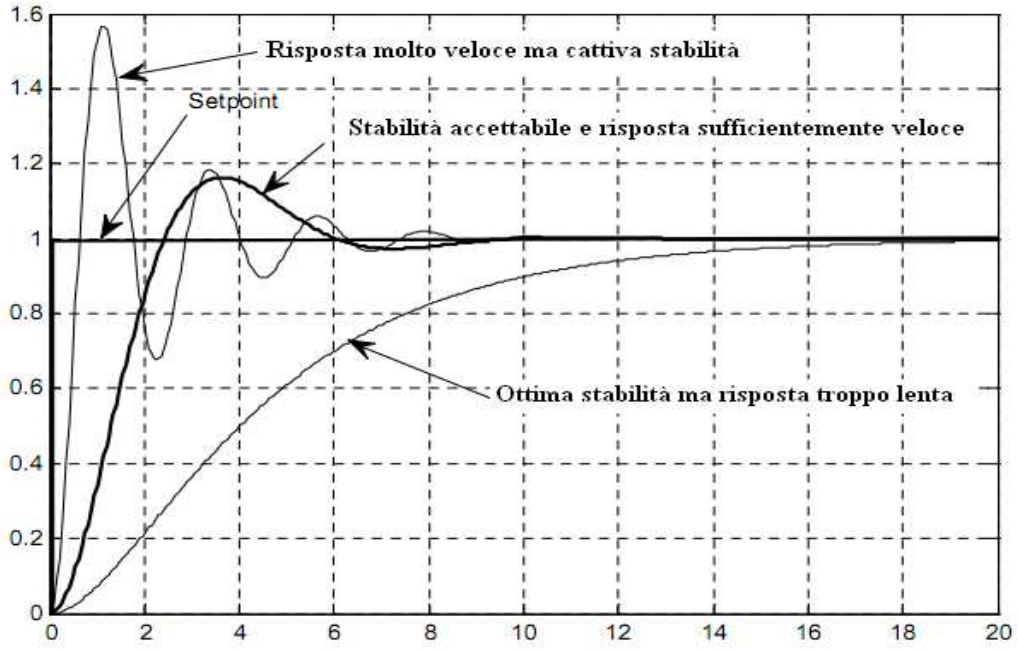


Figura 30 – diversi comportamenti della risposta allo scalino e loro analisi in funzione della stabilità e della velocità di risposta

1.4.4.1 Metodo Good Gain

Questo metodo è basato su esperimenti su sistemi reali o simulati. Nella procedura che andiamo ora a descrivere consideriamo un *PI*, ma aggiungeremo anche un'osservazione circa il termine derivativo. La procedura consta dei passi descritti nella successiva pagina.

1. Portiamo il processo il più vicino possibile alla condizione operativa nominale modificando il segnale di controllo nominale u^0, con il controllore in modalità manuale;

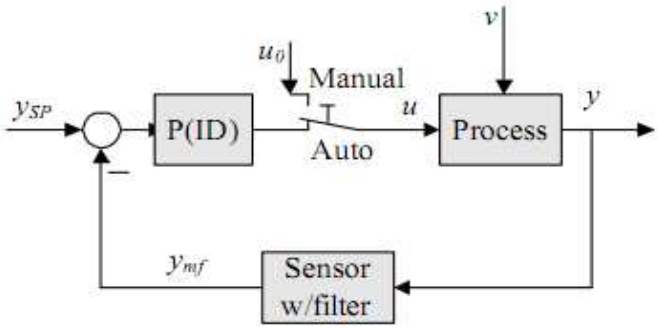


Figura 31 – controllore in modalità manuale per l'applicazione del metodo Good Gain

2. Poniamo *Kp = 0, Ti = ∞ e Td = 0* (il controllore è ora posto in modalità automatica);
3. Quindi *eccitiamo* il sistema con un gradino in ingresso e osserviamo il segnale misurato: aumentiamo *Kp* finchè l'anello di controllo non possiede una stabilità soddisfacente. E' importante che durante l'esperimento il segnale di controllo non venga portato ai limiti di saturazione o rischieremmo di trovare un valore di *Kp* che in normali condizioni di funzionamento non ci garantisca una buona stabilità. Di conseguenza il gradino in ingresso dovrà essere abbastanza piccolo, ovviamente non così piccolo rendere la risposta indistinguibile dal rumore;
4. Poniamo ora il tempo integrale *Ti* pari al valore *1.5*Tou*, dove *Tou* è l'intervallo di tempo tra la sovraelongazione e la sottoelongazione della risposta al gradino;

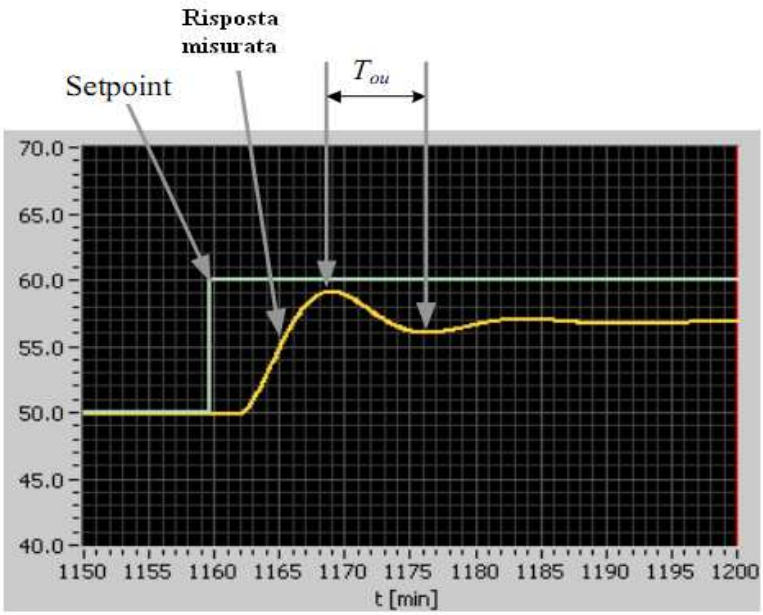


Figura 32 - risposta al gradino e misura Tou

5. A causa dell'introduzione del termine integrale, l'anello avrà probabilmente una stabilità ridotta rispetto al funzionamento col solo termine proporzionale. Per compensare ciò, *Kp* può essere un po' ridotto, ad esempio ponendo *Kp = 0.8·Kp*;
6. Se si volesse includere il termine derivativo, si può provare a porre *Td = Ti/4*, (corrispondente alla relazione *Ti-Td* usata da *Ziegler e Nichols*).
7. Infine controlliamo la stabilità del sistema di controllo applicando nuovamente un gradino. Se la risposta del sistema è soddisfacente la taratura del *PI(D)* è finita. Se il sistema desse viceversa segni di leggera instabilità, si può provare a ridurre il guadagno del controllore, possibilmente aumentando contemporaneamente il tempo integrale.

1.4.4.2 Metodo Skogestad

In questo metodo *model-based* i parametri del controllore sono espressi come funzioni dei parametri del modello del processo. Si ipotizzi che il sistema di controllo sia rappresentato da uno schema a blocchi come il seguente:

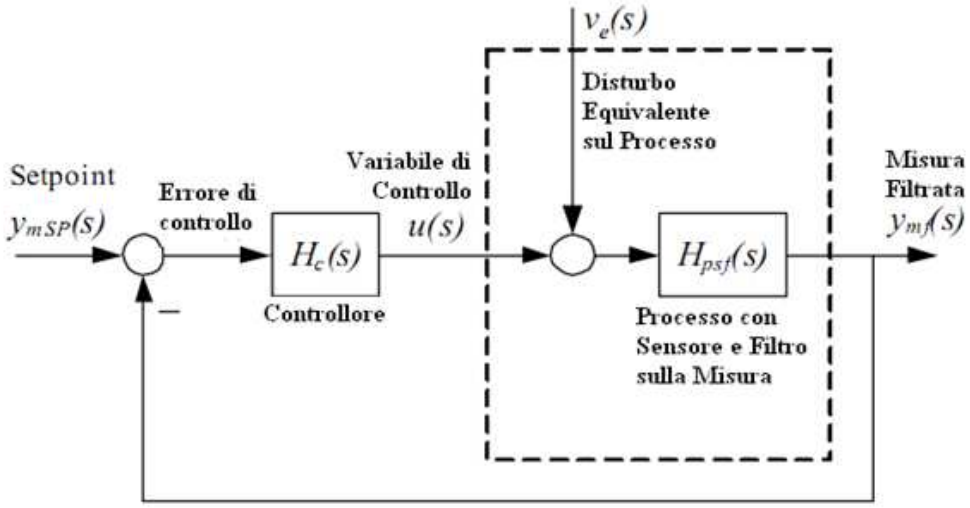


Figura 33 - schema a blocchi del sistema di controllo utilizzato

- $H_{pfs}(s)$ è la funzione di trasferimento del processo, del sensore e del filtro passa-basso sulla misura. Rappresenta quindi tutte le dinamiche percepite dal controllore e può essere ricavata da un semplice esperimento di risposta del processo a un gradino.
- $V_e(s)$ è un disturbo agente sul processo: questo viene ignorato durante la sintesi (andrebbe reinserito se si volesse effettuare un test con simulatore per vedere come il sistema di controllo compensa un disturbo di processo). Si noti che questo disturbo influenza dinamicamente il processo, nello stesso punto della variabile di controllo. Ad esempio, in un motore la variabile di controllo è la coppia mentre la coppia resistente di carico può essere considerata come il disturbo.

La funzione di trasferimento *T(s)* del sistema di controllo dal *setpoint* alla misura filtrata sarà quindi data da:

$$T(s) = \frac{y_{mf}(s)}{y_{m_{SP}}(s)} = \frac{H_c(s)H_{psf}(s)}{1 + H_c(s)H_{psf}(s)}$$

Si ipotizzi il comportamento del sistema pari a quello di un sistema rappresentato da una funzione di trasferimento del primo ordine affetta da un ritardo. Ovvero:

$$T(s) = \frac{1}{sT_C + 1} e^{-\tau s}$$

in cui T_c è la costante di tempo del sistema di controllo che l'utente deve specificare, mentre τ è il ritardo del processo. A questo punto si fornisce in ingresso un gradino e si misurano le due grandezze appena descritte, come mostrato in figura.

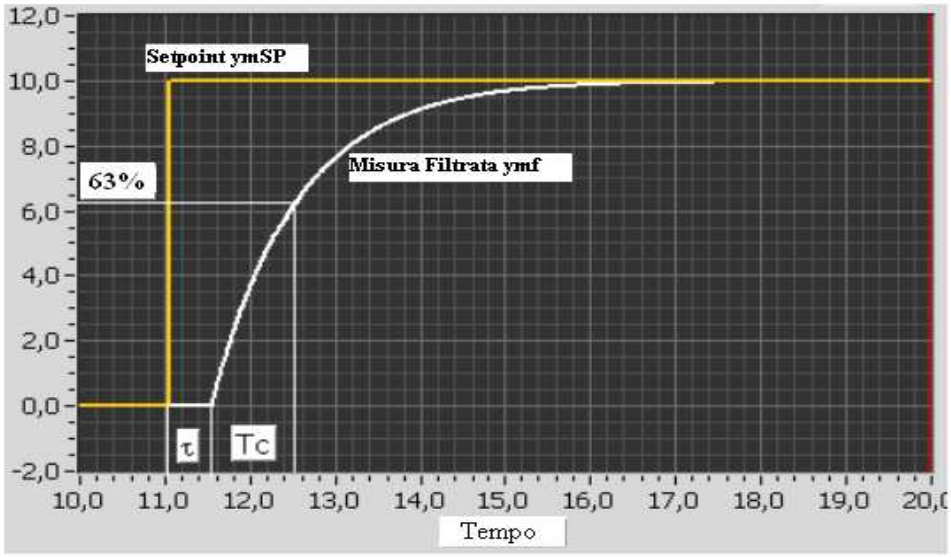


Figura 34 - risposta al gradino e misura di Tc e τ

Tenendo conto che per quanto detto:

$$T(s) = \frac{H_c(s)H_{psf}(s)}{1 + H_c(s)H_{psf}(s)} = \frac{1}{sT_C + 1} e^{-\tau s}$$

si ha che l'unico fattore sconosciuto è la funzione di trasferimento del controllore. Approssimando in modo opportuno il ritardo, il controllore diventa un *PID* o un *PI*.
Nella tabella seguente sono riportate le formule di sintesi di *Skogestad* per alcuni tipi di processo.

Tipo di processo	$H_{psf}(s)$	K_p	T_i	T_d
Integratore + ritardo	$\frac{K}{s}e^{-\tau s}$	$\frac{1}{K(T_C+\tau)}$	$c\,(T_C+\tau)$	0
Costante di Tempo + ritardo	$\frac{K}{Ts+1}e^{-\tau s}$	$\frac{T}{K(T_C+\tau)}$	$\min\left[T,\, c\,(T_C+\tau)\right]$	0
Integratore + Cost.Tempo + ritardo	$\frac{K}{(Ts+1)s}e^{-\tau s}$	$\frac{1}{K(T_C+\tau)}$	$c\,(T_C+\tau)$	T
Due Costanti di Tempo + ritardo	$\frac{K}{(T_1s+1)(T_2s+1)}e^{-\tau s}$	$\frac{T_1}{K(T_C+\tau)}$	$\min\left[T_1,\, c\,(T_C+\tau)\right]$	T_2
Doppio Integratore + ritardo	$\frac{K}{s^2}e^{-\tau s}$	$\frac{1}{4K(T_C+\tau)^2}$	$4\,(T_C+\tau)$	$4\,(T_C+\tau)$

Tabella 1 - formule di sintesi di Skogestad

T_1 è la più grande costante di tempo e T_2 la più piccola. Il fattore c è pari a 4 ma può essere ridotto a 2 nel caso la compensazione del disturbo risulti troppo lenta; ciò causa però una riduzione della stabilità e della robustezza rispetto a variazioni dei parametri del processo e un piccolo aumento della sovraelongazione della risposta al gradino.
Infine il metodo di *Skogestad* suggerisce di usare T_c pari a τ.
Il vantaggio del metodo di *Skogestad* sta nel non dover realizzare simulazioni per sintetizzare il controllore, in quanto i parametri vengono direttamente dal modello del processo e dal tempo di risposta del sistema.

1.4.4.3 Metodo Ziegler e Nichols

I parametri ottenuti con questo metodo danno buoni risultati nella reiezione dei disturbi sul carico. Il transitorio generalmente imposto da tali disturbi è infatti costituito da oscillazioni smorzate: con questo metodo il secondo picco di queste oscillazioni risulta attenuato di un fattore 4 rispetto al primo.
Il metodo ha due varianti: quella in anello aperto e quella in anello chiuso.

In anello aperto considero il seguente modello integrativo:

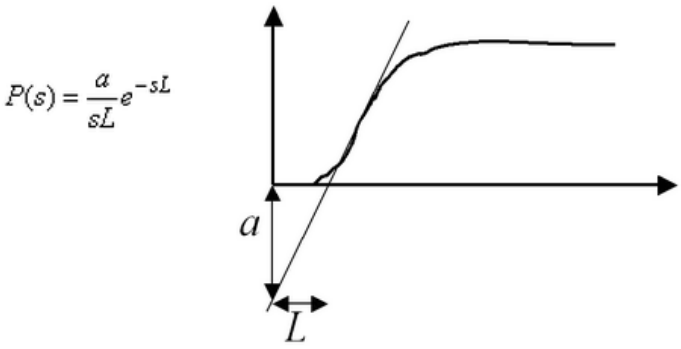


Figura 35 - modello integrativo

dove i parametri *a* e *L* vengono ricavati dalla risposta al gradino del sistema in anello aperto.
Stimati questi parametri, i parametri del controllore vengono ricavati dalla seguente tabella:

Controllore	K_p	T_i	T_d
P	$1/a$		
PI	$0.9/a$	$3L$	
PID	$1.2/a$	$2L$	$L/2$

Tabella 2 – valori di Ziegler e Nichols dei parametri col metodo in anello aperto

In anello chiuso, invece di compiere un'analisi nel tempo, realizziamo un'analisi in frequenza.
Inizialmente analizziamo la risposta del sistema controllato da un regolatore unicamente proporzionale, con valore di *Kp* vicino allo zero. Quindi, aumentiamo *Kp* fino a portare il sistema ai limiti della stabilità: un ulteriore incremento del valore *K** trovato porterebbe all'instabilità della risposta. Il sistema tarato con *K** genera una risposta oscillante di periodo *T**: partendo da questi due valori ricavo i parametri del controllore per avere una buona reiezione dei disturbi sul carico.

Controllore	K_p	T_i	T_d
P	$0.5K^*$		
PI	$0.4K^*$	$0.8T^*$	
PID	$0.6K^*$	$0.5T^*$	$0.125T^*$

Tabella 3 – valori di Ziegler e Nichols dei parametri col metodo in anello chiuso

Vediamo che in presenza di azione integrale, è necessario ridurre *Kp* per evitare troppa sovraelongazione. Se aggiungo azione derivativa, posso invece aumentare sia *Kp* che *Ki* in quanto il derivatore funge da smorzatore.
In generale, il metodo in anello chiuso fornisce risultati migliori, ma presenta l'inconveniente di dover portare il sistema ai limiti della stabilità, il che non è sempre permesso o fisicamente accettabile.

1.5 Microcontrollore

1.5.1 Descrizione generale

PIC (Programmable Interface Controller) è una famiglia di circuiti integrati a semiconduttore con funzioni di microcontrollore, prodotta da *Microchip Technology*.
Un microcontrollore è un dispositivo utilizzato generalmente in sistemi *embedded*, ovvero per applicazioni specifiche di controllo digitale. È progettato per ottenere la massima autosufficienza funzionale ed ottimizzare il rapporto prezzo-prestazioni per una specifica applicazione.
È in grado di interagire direttamente con il mondo esterno tramite un programma residente nella propria memoria interna e mediante l'uso di *pin* specializzati o configurabili dal programmatore.

Il cuore del microcontrollore è la *CPU (Central Processing Unit),* la quale recupera le istruzioni, le decodifica, recupera le risorse necessarie, esegue le istruzioni e scrive i risultati dell'esecuzione nelle opportune destinazioni. Il risultato dipende, oltre che dal dato su cui opera, dallo stato interno della *CPU* che tiene traccia delle passate operazioni. La *CPU* contiene un'unità *ALU (Arithmetic Logic Unit)*, che si occupa di eseguire le operazioni logiche e aritmetiche, un'*Unità di Controllo*, che legge dalla memoria le istruzioni e i dati, esegue l'istruzione ed eventualmente memorizza il risultato, e dei *Registri*, memorie interne rapidamente accessibili il cui stato nel complesso rappresenta lo stato della *CPU*.

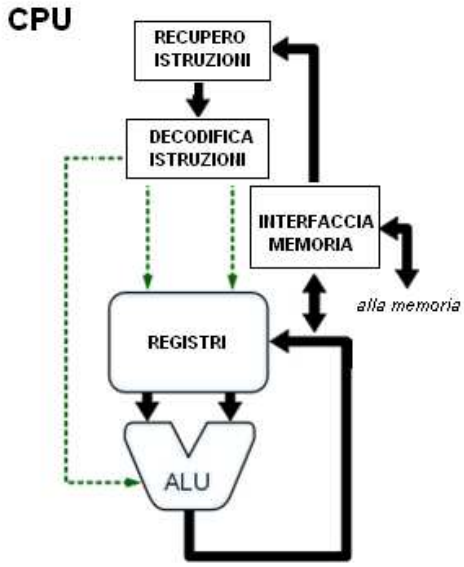


Figura 36 - CPU (Central Processing Unit)

Due particolari registri sono sempre presenti: il registro *IP (Instruction Pointer)* o *PC (Program Counter)*, che contiene l'indirizzo della cella di memoria della prossima istruzione da eseguire, e il

registro dei *flag*, un insieme di bit che segnalano stati particolari della *CPU* e danno informazioni sul risultato dell'ultima istruzione eseguita.

Per accelerare l'esecuzione e ridurre gli stati di attesa, il microcontrollore è dotato di un modulo *Prefetch* che precarica le prossime istruzioni e le risorse richieste. Per sopperire i problemi legati al salto delle istruzioni, l'attività di *Prefetch* è realizzata mediante un complesso algoritmo di predizione che consente di individuare la giusta istruzione successiva da caricare.

Lo stato della *CPU* cambia ogni volta che riceve un impulso da un segnale detto di *clock*. Esiste infatti un sistema di distribuzione che porta tale segnale a tutte le unità della *CPU*, in modo che siano sempre sincronizzate. Il tempo che impiega il *clock* a percorrere il ramo più lungo del sistema di distribuzione definisce la massima frequenza operativa della *CPU*.
PIC32 offre un sistema di quattro oscillatori interni ed esterni utilizzabili come sorgenti di *clock*, a cui può essere applicato un modulo *PLL (Phase Locked Loop)* che consente di ottenere una sincronizzazione costante nel tempo e di assorbire le eventuali variazioni nella frequenza del segnale di riferimento. Il *PLL* è anche fornito di divisori e moltiplicatori che consentono di ottenere in uscita un'oscillazione a frequenza desiderata. E' altresì a disposizione dell'utente un divisore della frequenza della sorgente di *clock* selezionata, il *Postscaler*.

Un importante modulo è l'*Interrupt Controller* che ordina, in base al livello di priorità, gli eventi di interrupt prima di presentarli alla *CPU*. Un *interrupt* è un segnale che indica il *bisogno di attenzione* da parte di una periferica che sta richiedendo un particolare servizio. All'arrivo di un *interrupt* la *CPU* deve mandare in esecuzione una particolare funzione chiamata *ISR* (*Interrupt Service Routine*). Affinchè il meccanismo delle interruzioni funzioni correttamente, prima di eseguire la *ISR* la *CPU* deve salvare il suo contesto attuale e ripristinarlo alla fine dell'*ISR*.

Una serie di segnali elettrici esterni tengono inoltre la *CPU* al corrente dello stato del resto del sistema e le permettono di agire su di esso (ad esempio il già citato *clock* o il *reset*).

Il *PIC32* implementa due spazi d'indirizzo: uno virtuale, l'altro fisico. Gli indirizzi virtuali sono usati esclusivamente dalla *CPU* per recuperare ed eseguire le istruzioni, come ad esempio l'accesso alle periferiche. Tutte le risorse *hardware* come la memoria dei programmi (*Program Memory*), la memoria dati (*Data Memory*) e le periferiche sono allocate ai loro rispettivi indirizzi fisici. Questi indirizzi sono inoltre utilizzati da periferiche come il *DMA (Direct Memory Access) Controller* che accede alla memoria senza interrompere l'esecuzione della *CPU*, aumentando così le prestazioni.

Le versioni *PIC* più usate sono quelle con memoria di tipo *FLASH* (indicata con una F all'interno del nome del componente), una memoria non volatile che preserva inalterato il suo contenuto anche in assenza della tensione di alimentazione.
Il funzionamento delle memorie *FLASH* si basa su celle costituite da un normale transistor MOS a cui viene aggiunto uno strato conduttivo sepolto tra gate e canale conduttivo che prende il nome di *floating gate*.

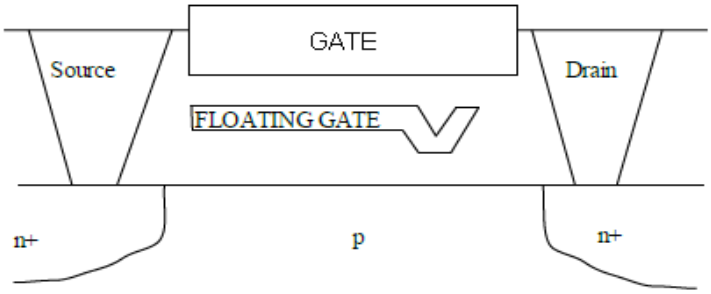


Figura 37 - transistor MOS con floating gate

In fase di programmazione (scrittura 1), alle tensioni elevate applicate corrisponde un'elevata corrente tra *source* e *drain* che innesca fenomeni fisici che portano al passaggio di alcuni elettroni nel *floating gate*, dove depositano una carica negativa che aumenta la tensione di soglia del dispositivo. Questa differenza nella tensione di soglia permette di distinguere due stati corrispondenti ai due livelli logici da memorizzare.
In fase di cancellazione (scrittura 0), si sfrutta lo spessore ridotto tra canale e *floating gate* nei pressi del *drain* per ridurre la barriera di potenziale così superabile per effetto *tunnelling.*
Esistono due tipi diversi di memorie *FLASH*, ma per entrambe la cancellazione non può essere fatta su un numero arbitrario di celle, bensì avviene per settori.
Le *FLASH* di tipo *NOR* permettono di indirizzare ciascuna cella, avente *floating gate* a massa, in modo indipendente. Per la lettura, piloto il *Control Gate* della cella che mi interessa: se la tensione di soglia non è stata alterata dalla presenza di carica sul *floating gate* la *bit line* mostrerà "0", altrimenti "1". Per la programmazione, sfrutto l'effetto degli *hot-electrons*, ovvero elettroni sufficientemente energetici da superare la barriera di potenziale del dielettrico tra canale e *floating gate*.
Le *FLASH* di tipo *NAND*, al contrario, non permettono di indirizzare la singola cella bensì la pagina che la contiene, in cui i *floating gate* sono in serie tra loro. Di conseguenza la lettura/scrittura di un singolo bit comporta la lettura/scrittura dell'intera pagina. Per la lettura, di conseguenza, si applica a ciascun transistor a giro una tensione pari a massa mentre agli altri una tensione tale da accendere il transistor qualunque sia la tensione di soglia. Così facendo, solo in assenza di carica negativa nel *floating gate* del transistor a massa, la corrente fluisce lungo tutta la pagina fino alla *bit line*. Per la programmazione viene sfruttato l'effetto *tunnelling.*
A fronte di un operazione di lettura più complessa, le *FLASH NAND* presentano due vantaggi rispetto le *NOR*: il primo è quello di ridurre i collegamenti elettrici, il secondo è quello di avere una maggiore affidabilità garantita grazie all'utilizzo dello stesso fenomeno fisico per la cancellazione e la scrittura, il che consente di ottimizzare i parametri tecnologici della cella.

La *Program Flash Memory* interna utilizzata per eseguire il codice utente viene da noi programmata con metodo *ICSP (In-Circuit Serial Programming)*, il quale sfrutta una connessione dati seriale e permette tempi di programmazione moto rapidi. In particolare è stato utilizzato il *PIC Kit 3 In-Circuit Debugger/Programmer* che permette il debug e la programmazione di tale memoria

Flash del PIC, grazie anche all'interfaccia utente di *MPLAB IDE*. Il *PIC Kit* viene connesso tramite interfaccia *USB* al *PC* e tramite apposito connettore al dispositivo.

Figura 38 - PIC Kit 3 della Microchip Technology

Il linguaggio di programmazione dei *PIC* è l'*assembly*, ma sono stati implementati alcuni compilatori per semplificarne la programmazione. Per la famiglia *PIC32*, *Microchip* mette a disposizione un compilatore *C*; i vantaggi di utilizzare un linguaggio ad alto livello sono la riutilizzabilità di codice sorgente sviluppato per altre architetture e la maggior leggibilità.

La presenza di periferiche *On-Chip* fa la differenza tra un microprocessore ed un microcontrollore e permette di partire da semplici *I/O* per arrivare a funzioni complesse.

Il nostro modello di *PIC32* lavora fino a *80 MHz* su dati a 32 bit, offre *512 KB* di *Program Memory* e ospita al suo interno numerosissime periferiche, come *UART, SPI, ADC*, utilizzabili sfruttando i relativi pin predisposti dei 64 pin totali del *package*.

1.5.2 PIC32MX340F512H

In Figura 29 viene mostrato lo schema del microcontrollore che verrà usato per il nostro progetto, con in evidenza le porte I/O.
Vediamo quindi di descrivere le periferiche che vengono utilizzate per questo progetto:

PORTE I/O: sono le periferiche più semplici. Esse consentono al *PIC* di monitorare e controllare gli altri dispositivi. Per aumentare la flessibilità, la maggior parte di queste porte sono multiplexate con diverse funzioni legate alle periferiche presenti sul *PIC.* Quando una di queste funzioni è attivata, il corrispondente pin non può essere utilizzato come generico *I/O*.

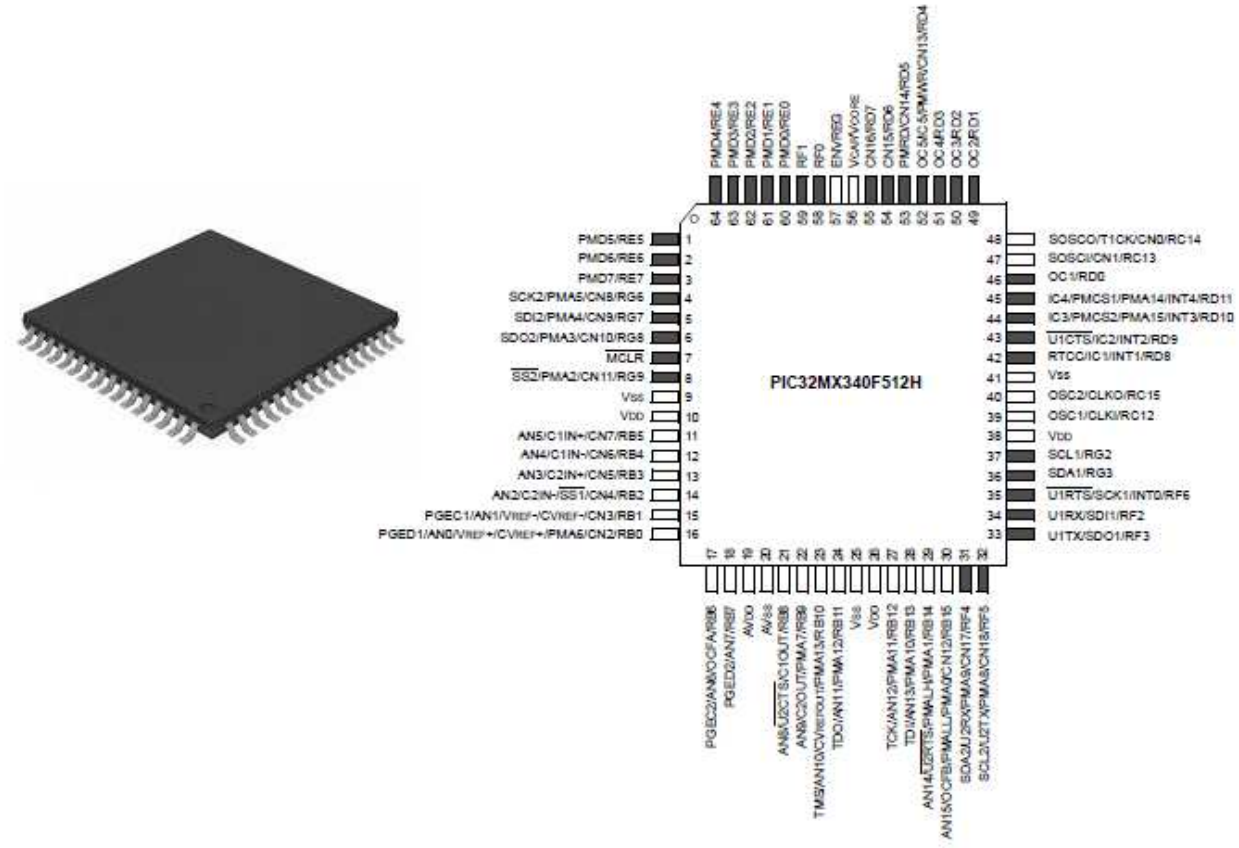


Figura 39 - PIC32MX340F512H

TIMER 2/3: sono due timer sincroni a 16-bit forniti di *Prescaler* che possono essere utilizzati come base temporale per i moduli di *Output Compare*. I due timer possono essere fusi all'occorrenza in un unico timer a 32-bit.

OUTPUT COMPARE: è usato per generare un singolo impulso o un treno di impulsi in risposta agli eventi selezionati su base temporale. Per ogni modalità di funzionamento questo modulo compara i valori salvati nei registri OCxR e OCxRS con il valore del timer selezionato. Quando questi valori si uguagliano, il modulo genera un evento basato sulla modalità di funzionamento selezionata. La modalità più comune è la modalità PWM (Pulse Width Modulation): scrivendo un opportuno valore nel registro PRy del timer selezionato viene settato il periodo di *PWM*, scrivendo nel registro OCxRS invece viene settato il *Duty Cycle* desiderato. Alla fine di ogni periodo di *PWM* il valore scritto in OCxRS viene automaticamente caricato nel registro OCxR che rappresenta il *Duty Cycle* applicato.

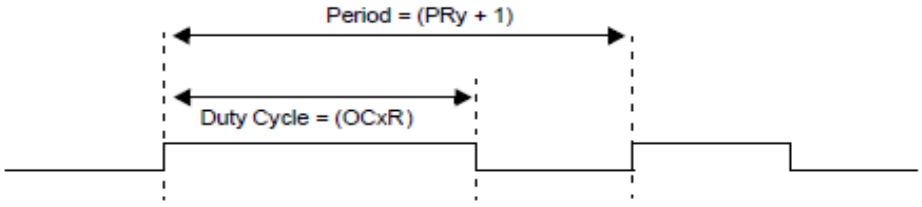


Figura 40 - modalità PWM dell'Ouput Compare

SERIAL PERIPHERAL INTERFACE (SPI): è un interfaccia seriale sincrona utile per comunicare con periferiche esterne e con altri microcontrollori. Essa dispone di *buffer FIFO* separati per ricezione(*SPIxRXB*) e trasmissione(*SPIxTXB*), dimensione configurabile dei dati e interrupt dedicato. Nella modalità di funzionamento tipica (*Master to Slave*), *SPI* controlla la generazione del clock seriale; il numero di impulsi di clock in uscita corrisponde alla dimensione dei dati trasferiti. Nella modalità standard di funzionamento dei buffer, i due registri unidirezionali di ricezione e trasmissione condividono i buffer *SPIxBUF* e *SPIxSR*. Quando un dato viene ricevuto in *SPIxSR*, viene trasferito in *SPIxRXB* e quindi in *SPIxBUF* dove attende di essere letto dal software. Quando un dato viene scritto in *SPIxBUF*, viene trasferito in *SPIxTXB* e quindi in *SPIxSR* da dove viene trasmesso in uscita. Sia la ricezione che la trasmissione sono vincolate da interrupt dedicati che le abilitano quando i buffer di *RX* e *TX* sono liberi e lo scambio dati precedente è terminato.

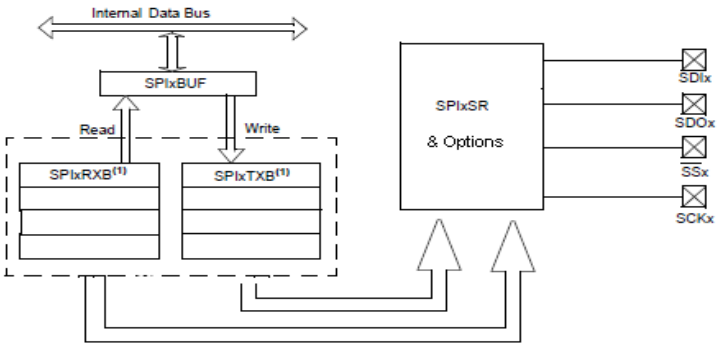


Figura 41 - struttura dell'interfaccia SPI

ADC (Analog-to-Digital Converter): è un convertitore a 10 bit con architettura *SAR (Successive Approximation Register)* che accetta in ingresso fino a 16 ingressi analogici. Tali ingressi sono collegati tramite due multiplexer a un *SHA (Sample&Hold Amplifier)* e possono essere campionati uno dopo l'altro abilitando opportunamente lo *Scan* automatico dei canali e l'acquisizione alternata dai due multiplexer. I risultati della conversione vengono immagazzinati in un buffer di 16 parole di 32 bit.

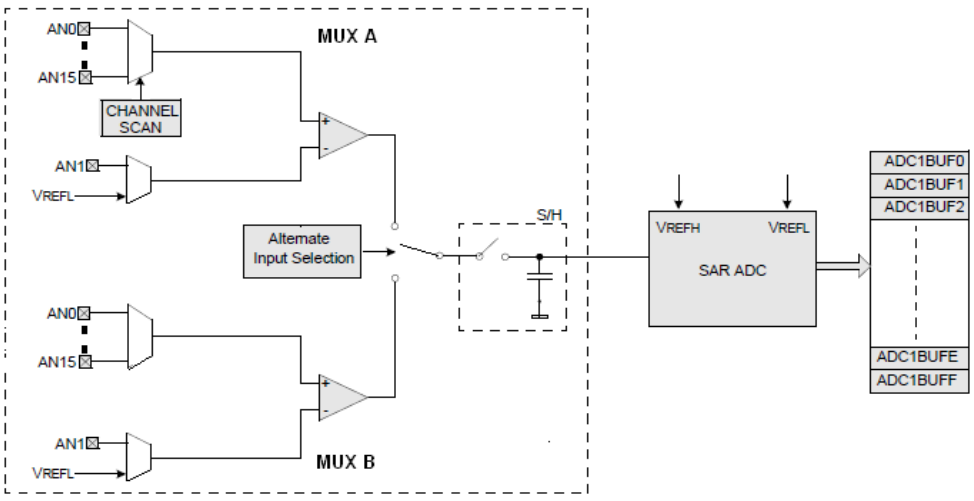


Figura 42 - struttura di acquisizione dei segnali analogici

L'architettura *SAR*, basata sul metodo delle approssimazioni successive, utilizza un unico comparatore e giunge alla conversione finale ricorrendo a più confronti successivi. Ciò permette di realizzare una struttura compatta ed economica, ma comporta velocità ridotta.

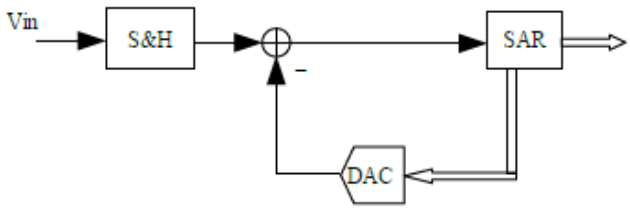


Figura 43 - architettura SAR

Il *Sample & Hold* ha il compito di mantenere il valore analogico costante per il tempo necessario al convertitore per eseguire la comparazione dello stesso con una serie di tensioni prodotte da un *DAC* interno, finchè le due misure non corrispondono entro un certo margine di errore. Per garantire ciò, si sfrutta un condensatore a bassa capacità *C* per conservare la tensione analogica ed un interruttore (solitamente un *MOSFET*) per connettere e disconnettere il condensatore dall'ingresso analogico.

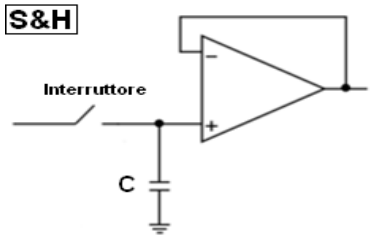


Figura 44 – Sample&Hold

Il blocco *SAR* esegue l'algoritmo iterativo grazie a cui viene trovata la codifica corrispondete all'ingresso analogico campionato dal *S&H*. In partenza si imposta il *DAC* presente sulla linea di retroazione in modo da generare una tensione pari a $\frac{\mathrm{FSR}}{2}$, dove *FSR (Full Scale Range)* è la dinamica complessiva a disposizione del convertitore. Il comparatore verifica se questo valore è maggiore o minore rispetto alla tensione in ingresso e ne comunica l'esito fornendo il *MSB (Most Significant Bit*, ovvero il *bit (N-1)*), che viene memorizzato dal blocco *SAR*. Al secondo passaggio il *DAC* viene impostato per generare in uscita una tensione pari a $\mathrm{MSB} \cdot \frac{\mathrm{FSR}}{2} + \frac{\mathrm{FSR}}{4}$, il comparatore ci fornisce il *bit (N-2)* e *SAR* ne memorizza il valore. Si procede così finchè non si ottiene il *bit 0*. Per ottenere *N bit* di risoluzione è quindi necessario ciclare *N* volte: se volessi una risoluzione elevata dovrei dunque accontentarmi di una bassa frequenza di campionamento.

UART: un ricevitore/trasmettitore asincrono universale (*Universal Asynchronous Receiver/Transmitter*) è un componente hardware che traduce i dati dalla forma parallela (del processore) alla seriale (della porta *RS232*) e viceversa. Il termine “universale” indica che il

formato dei dati e le velocità di trasmissione sono configurabili e che i metodi e i livelli dei segnali elettrici tipicamente sono gestiti da uno speciale circuito driver esterno al *UART* (*transcreiver*).
Lo *UART* è un circuito integrato usato per le comunicazioni seriali con un computer o con la porta seriale di una periferica, sfruttando il protocollo RS-232. Quando lo *UART* riesce a comunicare anche in maniera sincrona, prende il nome di *USART* (*Universal Synchronous/Asynchronous Receiver/Transmitter*). Il modulo supporta opzionalmente il controllo di flusso hardware e mette a disposizione un *encoder/decoder IrDA*, da noi non utilizzati. I buffer di ricezione e trasmissione seguono logica *FIFO* ed hanno interrupt dedicati. La dimensione dei dati trasmessi è scelta di 8 bit e il *Baud Rate Generator* è dotato di *Prescaler* a 16 bit.

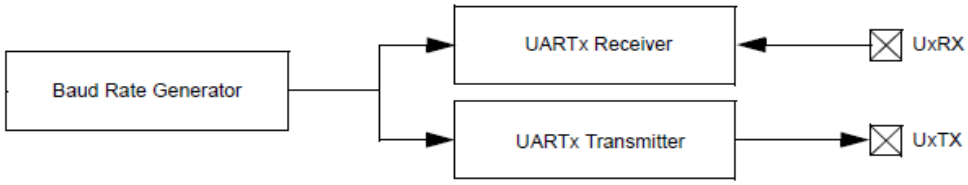


Figura 45 – modulo UART

L'azione dello *UART* si manifesta nel prendere byte di dati e trasmettere i singoli bit in modo sequenziale. A destinazione, un secondo *UART* riassembla i bit nei byte completi. Ciascun *UART* contiene un registro a scorrimento (*shift register*) che rappresenta il metodo fondamentale di conversione tra parallelo e seriale e viceversa. La trasmissione seriale di informazione digitale attraverso un singolo cavo o un altro mezzo ha un miglior rapporto qualità-prezzo rispetto alla trasmissione parallela attraverso molteplici cavi. L'*UART* solitamente non genera ne riceve direttamente i segnali esterni bensì dei dispositivi di interfaccia hanno il compito di convertire i segnali dei livelli logici dell'*UART* nei livelli dei segnali esterni e viceversa.
Tipicamente ciascun carattere spedito è formato da un bit iniziale posto al livello logico basso, un numero di bit dati configurabile (solitamente 8) e da uno o più bit finali posti al livello logico alto.
Il bit iniziale segnala al ricevitore che un nuovo carattere è in arrivo, i bit finali che il carattere è terminato. Grazie alla differenza di livello logico tra bit iniziale e bit finali è sempre chiara la demarcazione tra caratteri successivi.
Tutte le operazioni dell'hardware *UART* sono controllate da un segnale di clock operante ad una velocità multiplo della velocità dati (ad esempio, ciascun bit dati è lungo come 16 impulsi di *clock*).
Il ricevitore testa lo stato del segnale in ingresso ogni impulso di *clock*, cercando il bit di inizio. Se l'ipotetico bit di inizio dura almeno metà del tempo di un bit, esso viene considerato valido, indicando così l'inizio di un nuovo carattere. Se così non è, l'impulso spurio viene ignorato. Trovato il bit d'inizio, dopo aver aspettato l'ulteriore tempo di un bit, lo stato della linea viene nuovamente campionato e il livello risultante registrato dentro al registro a scorrimento. Dopo che il tempo corrispondente alla lunghezza configurata del carattere è trascorso, il contenuto del registro a scorrimento viene reso accessibile (in parallelo) al sistema ricevente. L'*UART* alza un bit di *flag* per indicare che un nuovo dato è a disposizione e può anche generare un interrupt che richiede al processore di trasferire il dato ricevuto. In alcuni tipi comuni di *UART* viene inserita una piccola

memoria di *buffer* a logica *FIFO (First In First Out)* tra il registro a scorrimento del ricevitore e l'interfaccia del sistema. Questo accorgimento permette al processore di avere più tempo per gestire l'interrupt in arrivo dall'*UART* e prevenire così la perdita di dati ad alte velocità.
L'operazione di trasmissione è più semplice in quanto è sotto il controllo del sistema trasmittente. Non appena il dato viene depositato nel registro a scorrimento dopo il completamento del carattere precedente, l'hardware *UART* genera un bit di inizio, fa scorrere in uscita il numero desiderato di bit dati e infine aggiunge i bit di fine. Dato che la trasmissione di un singolo carattere può impiegare un lungo tempo rispetto alle velocità della *CPU*, l'*UART* mantiene alto un bit di *flag* indicante lo stato "occupato" così che il sistema non depositi un nuovo carattere da trasmettere finchè il precedente non è stato completato (questo può essere alternativamente realizzato utilizzando un *interrupt*).
La trasmissione e la ricezione della *UART* devono essere settate con uguali velocità, lunghezza dei caratteri e numero di bit di fine.
Se la comunicazione non è di tipo *Half Duplex* (o ricezione o trasmissione) bensì di tipo *Full Duplex* (ricezione e trasmissione nello stesso momento), l'*UART* utilizza due differenti registri a scorrimento: uno per i caratteri trasmessi, l'altro per i caratteri ricevuti.
Se viene messa a disposizione anche la trasmissione sincrona (*USART*), in essa il *clock dati* viene recuperato separatamente dal flusso dei dati e non vengono usati bit di inizio e di fine. Questo aumenta l'efficienza della trasmissione dato che aumenta la percentuale di bit dati utili sui bit totali. D'altro canto, mentre nella trasmissione asincrona nessun carattere viene spedito quando il dispositivo trasmittente non ha dati da inviare, nella trasmissione sincrona deve essere sempre mantenuta la sincronizzazione tra trasmittente e ricevente e per questo anche in assenza di dati da trasferire viene ripetutamente inviato un carattere (di solito il carattere *ASCII "SYN"*).

1.6 Layout zigkeys

Per il controllo wireless del trenino abbiamo a disposizione due schede, una lato *PC,* chiamata *ZigKey*, e una lato trenino, chiamata *ZigKey_trenino*, entrambe equipaggiate con lo stesso *PIC32* e con lo stesso modulo *MRF24J40MA*.
La scheda lato trenino possiede inoltre un *Ponte-H*, un connettore a 8 pin, un trasformatore *LM2937ES-3.3* e i sensori di corrente (*shunt*) e tensione (*partitori di tensione*).

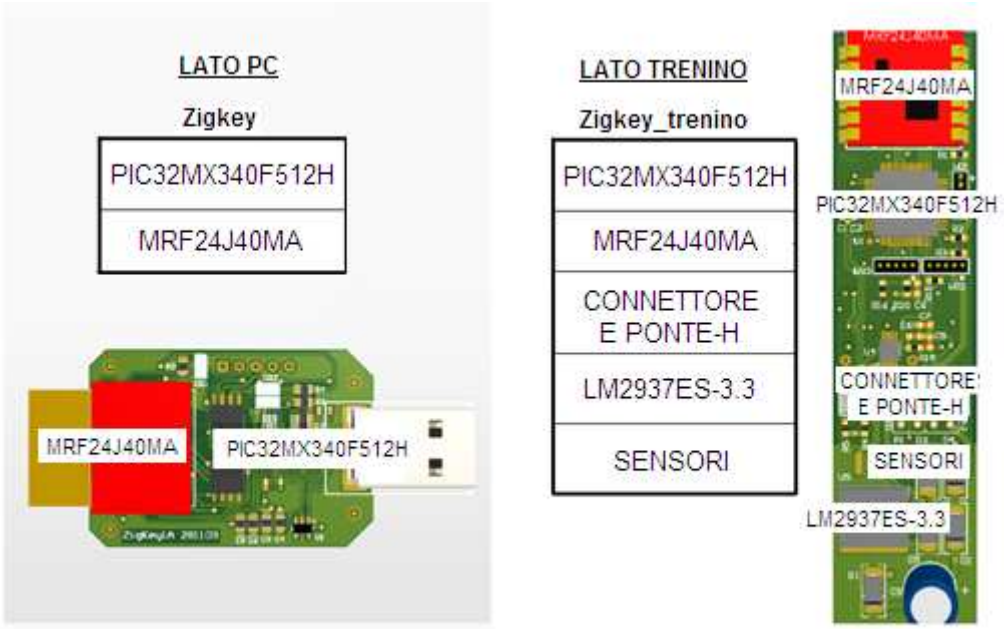


Figura 46 - layout schede

Di seguito si presenteranno nel particolare questi componenti, escludendo ovviamente il *PIC32* di cui si è già parlato in precedenza.

1.6.1 MRF24J40MA

E' un modulo ricetrasmittente prodotto da *Microchip* compatibile con la famiglia *PIC32*. Grazie alle sue piccole dimensioni (*17.8 mm x 27.9 mm*) è facilmente montabile sul circuito stampato. Esso rispetta lo standard *IEEE 802.15.4* e supporta i protocolli di rete wireless *MiWi, MiWi P2P* e *ZigBee*.
E' principalmente costituito da due parti: il modulo *MRF24J40* e l'antenna, collegate tramite un circuito intermedio di adattamento.

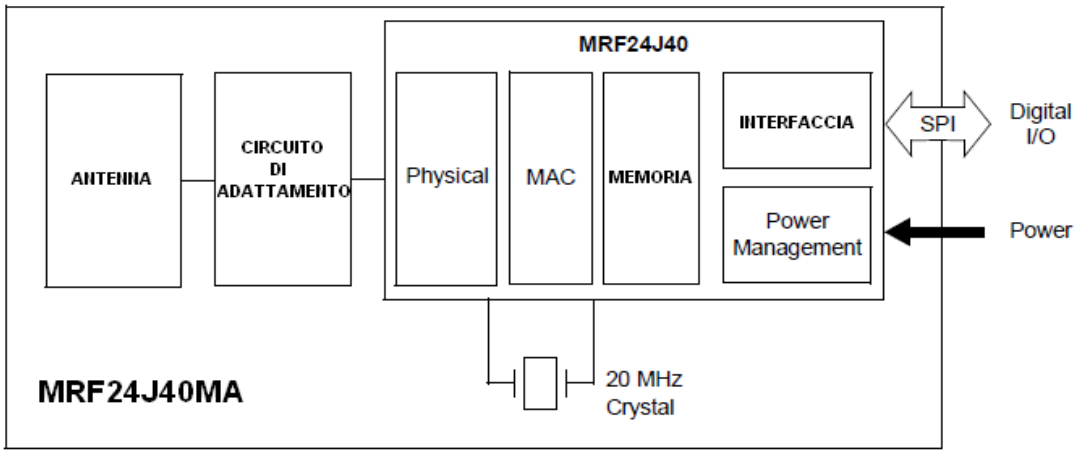


Figura 47 – trasmettitore MRF24J40MA

1.6.1.1 MRF24J40

Questo modulo è il cuore della ricetrasmittente. Il ricevitore presenta architettura *Low-IF (Low-Intermediate Frequency),* composta da un amplificatore a basso rumore *LNA (Low Noise Amplifier)*, dei mixer, dei filtri polifase di canale e degli amplificatori limitanti la banda base con un indicatore della forza del segnale ricevuto *RSSI (Receiver Signal Strength Indicator)*.
La funzione principale di un ricevitore *RF (Radio Frequency)* è selezionare il canale utile, per poi traslarlo a bassa frequenza rendendo più facile la demodulazione. La demodulazione è un'operazione che consente di estrarre, da un segnale modulato in ampiezza, l'informazione a bassa frequenza. Nell'operazione di demodulazione si realizza una conversione di frequenza che a partire dallo spettro del segnale modulato in ampiezza permette di ricostruire il segnale in banda base.
L'*LNA* deve essere in grado di amplificare segnali a potenza molto bassa senza introdurre distorsioni armoniche e rumore. La riduzione del rumore è infatti l'obiettivo principale.
I filtri di canale sono deputati alla reiezione dei disturbi e hanno il compito di eliminare il segnale alla frequenza immagine *IMF (IMage Frequency)* prima che esso arrivi ai mixer.
La frequenza immagine *IMF* è simmetrica a *RF* rispetto alla frequenza LOF *(Local Oscillator Frequency)* del segnale prodotto dall'oscillatore locale (ovvero: $IMF = 2 \cdot LOF - RF$)
e proprio per questo può causare problemi: il mixer, infatti, trasla il semiasse frequenziale di destra verso sinistra e il semiasse frequenziale di sinistra verso destra di una quantità pari a *LOF*, sovrapponendo lo spettro del segnale utile con quello della frequenza immagine.
Di conseguenza se alla frequenza immagine *IMF* sono presenti interferenze, rumore o altri segnali, si avrà una riduzione del rapporto segnale/rumore (*SNR*, ovvero *Signal Noise Ratio*), indice della qualità del segnale ricevuto. É perciò fondamentale il lavoro dei filtri polifase di canale.

Spetta quindi ai mixer il compito di convertire la frequenza *RF* in frequenza intermedia *IF*.
La frequenza intermedia *IF* è data dalla radio frequenza *RF* a cui viene sottratta la frequenza *LOF* (ovvero *IF = RF – LOF*).
A questo punto gli amplificatori a frequenza intermedia aumentano l'intensità del segnale, lo filtrano e lo mandano in ingresso al demodulatore per la ricostruzione dell'informazione originaria.

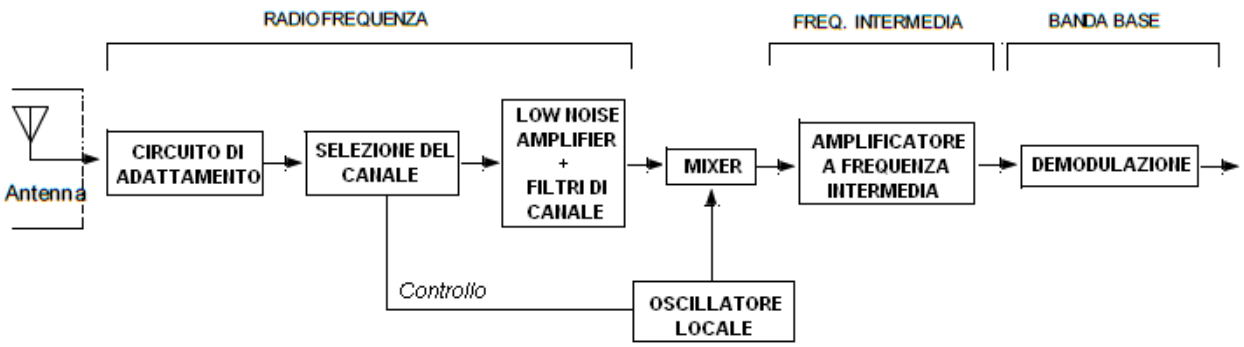


Figura 48 - ricevitore

Il trasmettitore presenta architettura a conversione diretta. Questa sfrutta il concetto di modulazione *IQ* (ovvero modulazione in fase e in quadratura) secondo cui due segnali moltiplicati rispettivamente per un coseno (modulazione in fase) e per un seno (modulazione in quadratura) alla stessa frequenza rimangono distinguibili anche se sommati tra loro. I segnali analogici I e Q in banda base vengono generati da due convertitori *DAC (Digital-to-Analog Converter)*. In uscita ai *DAC* vengono posti dei filtri passa-basso per rimuovere le distorsioni alla frequenza di campionamento prima della conversione. La conversione diretta in radio frequenza RF è fatta dal modulatore *IQ*, aiutato da un sintetizzatore di frequenza scandito da un oscillatore al cristallo esterno a *20 MHz* generante la radio frequenza di *2.4 Ghz*.
Infine il segnale in uscita *RF* viene amplificato o attenuato da un amplificatore a guadagno variabile (*VGA*) con un range di controllo del guadagno di *36 dB*. Viene messo a disposizione anche un amplificatore esterno *PA (Power Amplifier)*.

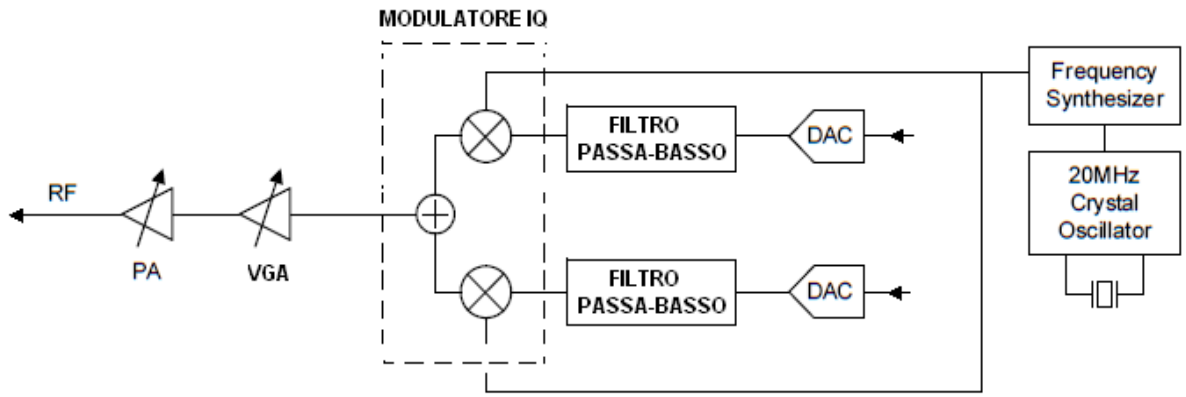


Figura 49 - trasmettitore

Un interruttore interno *TR (Transmit/Receive)* combina i circuiti di trasmissione e ricezione in due porte differenziali *I/O* chiamate *RFP* e *RFN* (*Positive* e *Negative*). Questi pin sono connessi al circuito di adattamento e di lì all'antenna.
Oltre all'oscillatore principale a *20 MHz*, che fornisce il segnale alla frequenza principale (*MAINCLK*) al trasmettitore, alla banda base e al circuito *MAC*, vengono messi a disposizione altri due oscillatori per generare la frequenza di *Sleep Mode (SLPCLK)*. Il primo, al cristallo, è esterno a *32 KHz*, il secondo interno a *100 KHz*; entrambi vengono sfruttati in *Sleep Mode* per contare il periodo di inattività e l'intervallo tra beacon se il dispositivo è *beacon-enabled*, l'intervallo di *Sleep* se il dispositivo è *beacon-disabled*.
Tutta questa parte del modulo *MRF24J40* rappresenta il livello fisico (physical layer). Tale livello è il primo dei 7 livelli ISO-OSI, uno standard per reti di calcolatori che stabilisce per l'architettura logica di rete un'architettura a strati composta da una pila di protocolli suddivisa in 7 livelli, i quali insieme espletano in maniera logico-gerarchica tutte le funzionalità della rete (tale standard sarà illustrato più dettagliatamente nella parte dedicata ai protocolli di rete). Lo schema è riportato nella seguente figura:

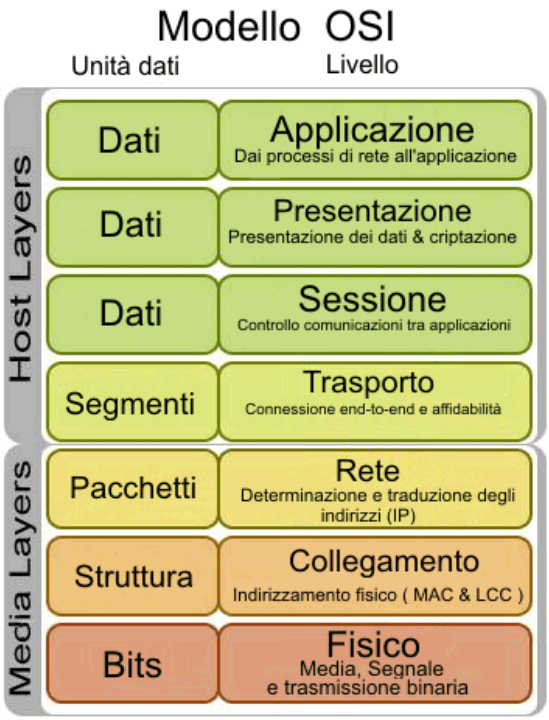


Figura 50 – standard ISO/OSI

Il livello fisico si occupa dell'attivazione e disattivazione del ricetrasmettitore, della selezione e del controllo del segnale e della ricetrasmissione del segnale attraverso il mezzo fisico.
E' sintetizzato dallo schema di Figura 51, nella pagina seguente.

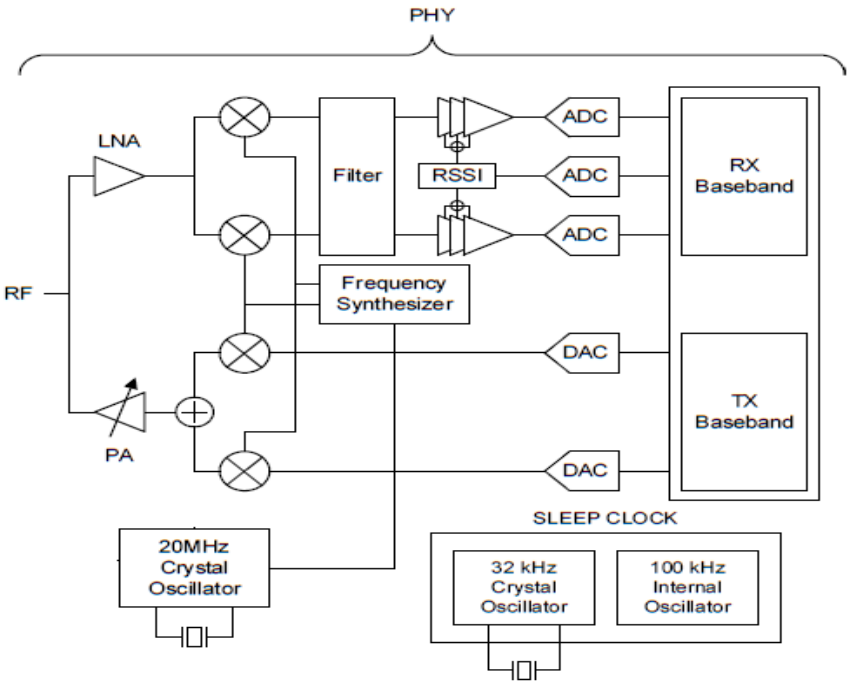


Figura 51 – livello fisico (PHY)

Oltre al livello fisico appena visto, il modulo *MRF24J40* integra anche la funzionalità *MAC*. Il MAC (*Media Access Control*) è uno dei due sottolivelli del livello di collegamento (*Data link layer*). Tale livello si occupa di formare i dati da inviare attraverso il livello fisico, incapsulando il pacchetto proveniente dallo strato superiore in un nuovo pacchetto provvisto di un nuovo *header* (intestazione) e *tail* (coda), usati anche per sequenze di controllo. Questa frammentazione dei dati in specifici pacchetti è detta *framing*.

Entrambi i livelli sono soggetti allo standard *802.15.4*^{TM} e sono così responsabili dello standard dei livelli superiori a cui forniscono servizi. Questi livelli superiori sono responsabili della configurazione della rete gestendo il reindirizzamento dei messaggi e quindi della parte applicativa della rete.

Ritornando al modulo *MRF24J40,* succede che il livello *PHY* manda al livello *MAC* un segnale *CCA (Clear Channel Assessment)*, ovvero lo informa se il mezzo trasmissivo è libero o occupato. Questa informazione può essere rilevata o tramite un semplice *Carrier Sense*, o misurando l'energia del mezzo con la funzione dedicata *Energy Detection* o la combinazione di questi due metodi.

Il livello *MAC* gestisce la connessione/disconnessione della rete, se il dispositivo è coordinatore garantisce l'accesso al mezzo fisico con la generazione di segnali di sincronizzazione (*beacon*) e svolge verifiche in ricezione e in trasmissione.

In ricezione (*RXMAC*) verifica la presenza di eventuali errori di ricezione controllando la FCS *(Frame Check Sequence)* del pacchetto e quindi verifica il frame.

In trasmissione (*TXMAC*) verifica che il formato dei pacchetti rispetti lo standard, attua il meccanismo *CSMA/CA*, genera la *FCS* e gestisce il *beacon enable*.

La memoria di *MRF24J40* è di tipo *SRAM (Static Random Access Memory),* volatile, caratterizzata da bassi tempi di lettura, bassi consumi, un elevato numero di componenti per cella (che ne limita

il numero di celle all'interno del chip sia per problemi di spazio che di dissipazione di potenza), ma libera dall'obbligo di *refresh* (a differenza della *DRAM*, ovvero la *Dynamic RAM*).

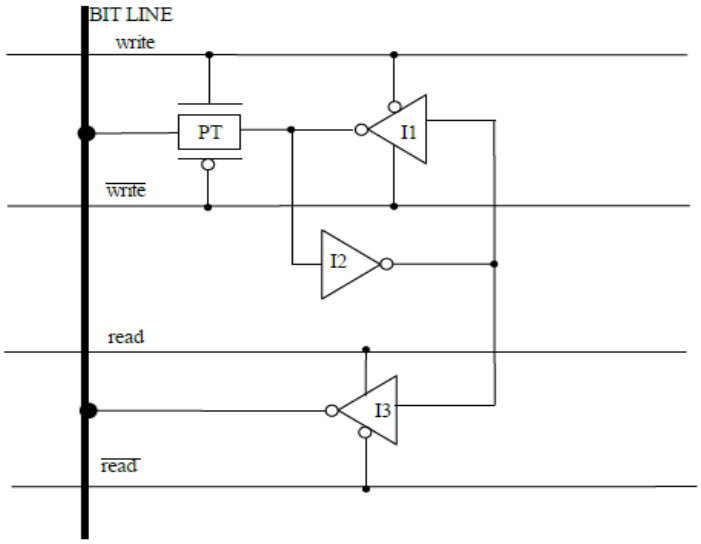


Figura 52 – cella di memoria SRAM a 12 transistors

Come possiamo vedere dallo schema della cella di memoria *SRAM* a 12 transistor, durante la scrittura del dato presente sulla *Bit Line* viene abilitato l'interruttore *PT (Pass Transistor)* e spento l'invertitore *I1*, disabilitando così la retroazione tra *I1* e *I2*. Il dato viene così letto da *I2* che setta il nuovo valore in ingresso a *I1*. Scritto il dato, la retroazione tra i due invertitori vieni riabilitata e grazie ad essa il dato viene conservato nella cella. Durante la lettura viene invece abilitato *I3* che permette l'acquisizione del dato immagazzinato.
Questa memoria *SRAM* è accessibile tramite la porta *SPI (Serial Peripheral Interface)* ed è divisa in registri di controllo e buffer dati. I registri di controllo provvedono al controllo, allo stato e all'indirizzamento per le operazioni dell'*MRF24J40*. I *buffer FIFO (First In First Out)* servono come buffer temporanei per i dati ricevuti e da trasmettere. Il buffer di ricezione è uno (*RXFIFO*), mentre i buffer di trasmissione sono 4, ma noi ne usiamo solo uno, il buffer *TX Normal FIFO*, usato per tutte le trasmissioni. I 3 rimanenti infatti vengono utilizzati in modalità *beacon-enabled*, che noi non utilizziamo.

L'interfaccia dell'*MRF24J40* presenta 13 pin: 6 *GPIO*, 4 *SPI*, un *RESET*, un *WAKE* e un *INT.*

- *Pin General Purpose Input/Output*: possono essere configurati individualmente ai fini di realizzare un controllo o un monitoraggio. Con essi, ad esempio, possono essere controllati gli amplificatori *LNA* e *PA* del trasmettitore.

- *Pin SPI:* servono all'*MRF24J40MA* per comunicare con un microcontrollore; un pin viene usato per la ricezione, uno per la trasmissione, uno per il clock utile a sincronizzare la comunicazione e uno per abilitare la comunicazione stessa.

- *Il pin RESET* consente un reset esterno dell'hardware, il pin *WAKE* consente al

microcontrollore di svegliare l'*MRF24J40MA* quando questo è in *Sleep Mode*, il pin *INT* permette di inviare un interrupt configurabile al microcontrollore.

Il circuito di gestione della potenza (*Power Management*) consiste principalmente di un regolatore di tensione che può variare la modalità di funzionamento tra *Sleep Mode* e *Normal Mode* e viceversa, con evidenti differenze di consumo. Il meccanismo di sicurezza (*Security Engine*) implementa uno standard che consente di avere comunicazioni sicure.

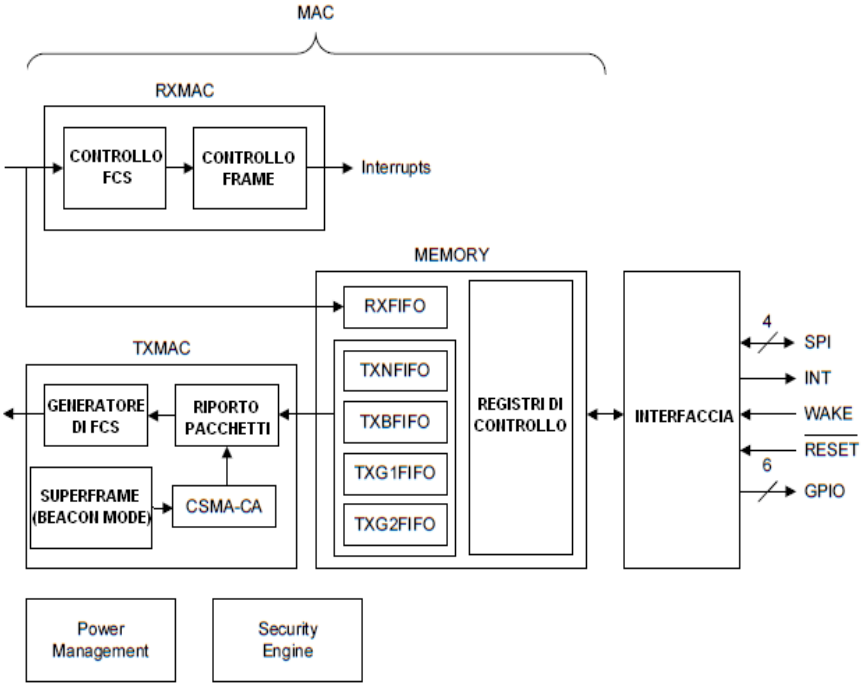


Figura 53 – livello MAC

1.6.1.2 Antenna

Il modulo *MRF24J40* è collegato tramite un circuito di adattamento ad una *PCB (Printed Circuit Board)* Antenna realizzata su circuito stampato.
Un antenna è un dispositivo elettrico che converte correnti elettriche in onde radio e viceversa.
In trasmissione, il radiotrasmettitore applica ai terminali dell'antenna una corrente elettrica oscillante a frequenza radio *RF*; questa corrente forzata attraverso l'antenna crea un campo magnetico oscillante intorno agli elementi dell'antenna, oltre al campo elettrico oscillante creato lungo gli elementi. Questi campi tempovarianti diffondono quindi l'energia della corrente sotto forma di onde radio elettromagnetiche.
In ricezione, il campo elettro-magnetico oscillante dell'onda radio in ingresso esercita forza sugli elettroni presenti negli elementi dell'antenna, causandone il continuo movimento e creando così delle correnti oscillanti nell'antenna. La piccola tensione creatasi ai terminali dell'antenna viene quindi applicata al ricevitore per essere amplificata.

Ecco un'immagine di come si presenta l'oggetto reale.

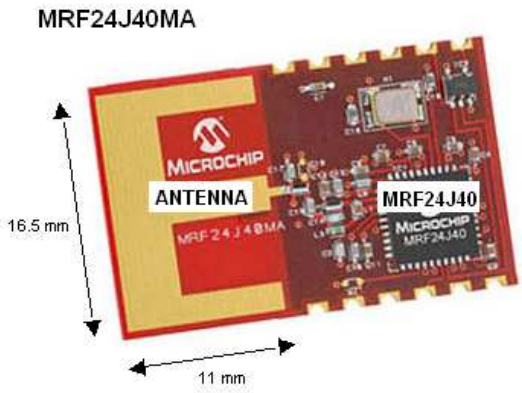


Figura 54 - ricetrasmittente MRF24J40MA

1.6.2 Ponte-H BD6222HFP-TR

Il nostro modello di Ponte-H ha le seguenti caratteristiche: 7 pin, tensione di alimentazione *Vcc* tra i *6* e i *15 V*, tensione massima in uscita di *18 V*, corrente massima in uscita di *2 A* ed è costruito per lavorare nel range di temperatura *-40°C - +85°C.*
Grazie ai due ingressi di controllo, *FIN (Forward Input)* e *RIN (Reverse Input),* la tensione di uscita può essere controllata in ampiezza e polarità, variando gli istanti di conduzione degli *switch* tramite l'aggiornamento dei *duty cycle*.
Noi imponiamo tensione di alimentazione V_{cc} e tensione di riferimento V_{ref} a *12 V* così da poter imporre in uscita una qualsiasi tensione compresa tra i *-12V* e i *+12V*.

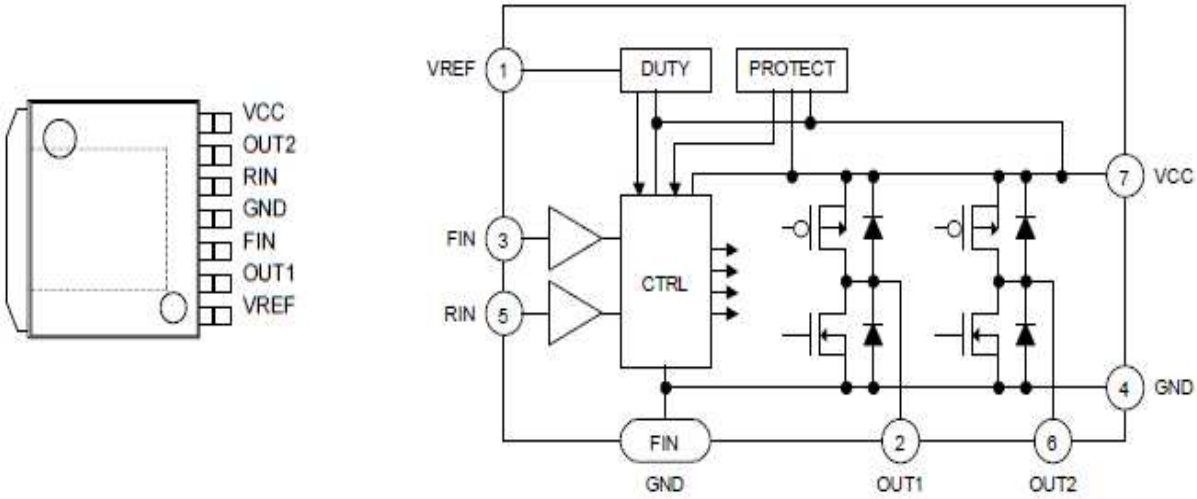


Figura 55 - package e struttura interna del ponte-H BD6222HFP-TR

Dal *DataSheet* abbiamo scelto le seguenti modalità di funzionamento:

1. *Stand-By Mode*: tutti i circuiti interni sono spenti e le uscite sono in alta impedenza. Se il motore sta girando al momento del passaggio in *Stand-By Mode*, il sistema entra in uno stato di inattività (*idle*) a causa dei diodi. Questa modalità è indipendente dalla tensione di riferimento *Vref*.
2. *Brake Mode*: è usata per fermare velocemente il motore. Se non serve uno stop immediato, conviene utilizzare al suo posto la *Stand-By Mode*. Infatti in *Brake Mode* le circuiterie interne sono operanti, il che comporta maggiori consumi rispetto alla *Stand-By Mode*.
3. *PWM Control Mode A*: l'uscita alta è fissa mentre l'uscita bassa esegue lo *switching* corrispondente al segnale di controllo in ingresso. Per un corretto funzionamento, la frequenza di *PWM* deve essere impostata tra i *20 KHz* e i *100 KHz*. Per operare in questa modalità devo connettere i pin di *Vref* e *Vcc*.

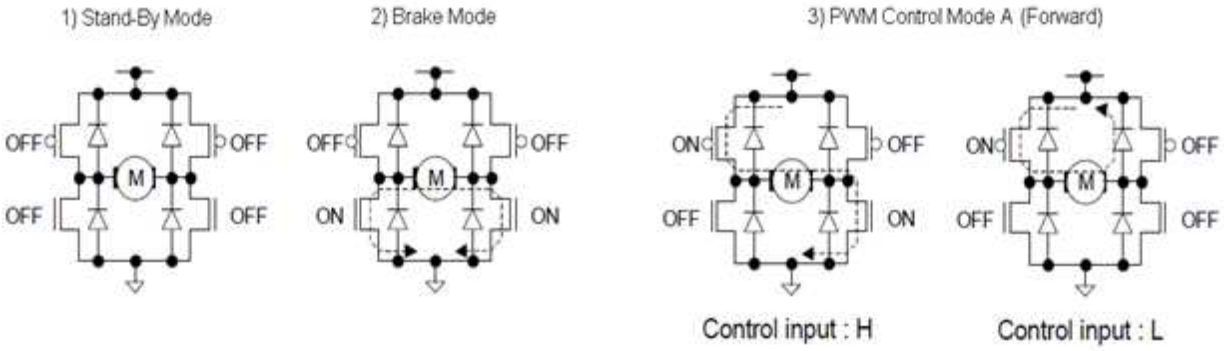


Figura 56 - modalità di funzionamento del ponte-H

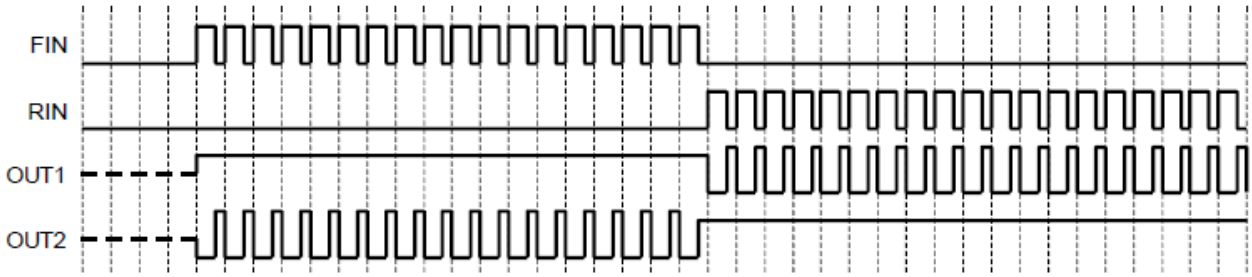


Figura 57 - esempio dell'andamento degli ingressi e delle uscite del ponte-H

1.6.3 Connettore

Il connettore è semplicemente un componente elettrico che permette di raggruppare i segnali provenienti da *PIC32* e da *PONTE-H* destinati al trenino elettrico e portarli ad esso.
La sua scelta è stata fatta nel rispetto delle pratiche raccomandate dalla *National Model Railroad Association (NMRA),* indicazioni meno importanti rispetto agli *Standards*, dato il ruolo meno critico

dei soggetti che trattano, ma comunque raccomandate al fine di promuovere la massima intercambiabilità tra le unità.
Nello specifico, è stato scelto il connettore "*Medium*" presentato nella pratica elettrica raccomandata *RP-9.1.1 "Electrical Interface & Wide Color Code for Digital Command Control"*.

Basic Interface Electromechanical Characteristics

	Small	Medium	Large
Connections (layout)	6 (1x6)	8 (2x4)	4 (none)
Part in Locomotive/car	female	female	male
Pitch	0.050"	0.100"	None
Pin Section	circular(1)	circular(1)	Circular
Pin Length	0.118"	0.155"	0.300"
Tolerance	0.001"	0.010"	0.030"
Pin Diameter	0.017"	0.022"	0.050"
Tolerance	0.002"	0.002"	0.003"
Power Rating (2)	0.50A	1.50A	4.00A
Peak Power Rating	0.75A	3.00A	6.00A
Suitable for Scales	N or larger	HO or larger	O or larger55

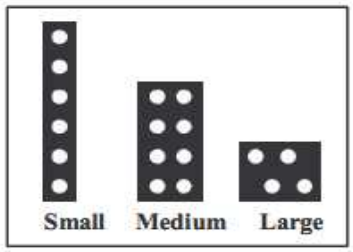


Figura 58 - caratteristiche e packages dei connettori

Gli 8 pin del connettore rappresentano rispettivamente:

PIN1: + del motore
PIN2: luci posteriori
PIN3: altra uscita (PWM)
PIN4: binario sinistro
PIN5: - del motore
PIN6: luci anteriori
PIN7: +12V
PIN8: binario destro

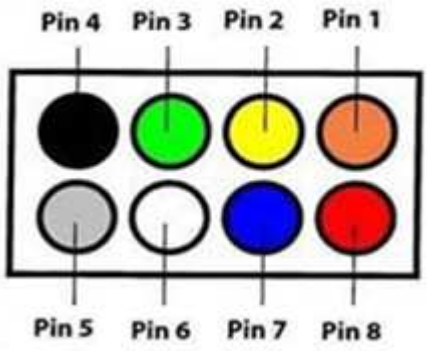


Figura 59 – standard di utilizzo dei pin del connettore medium

Il pin 3, indicato come *altra uscita*, viene utilizzato per imporre il *PWM* relativo alle luci sfumate.
Nel layout del progetto *TRNO1A* i pin sono stati numerati in modo diverso, ma i collegamenti corrispondono esattamente a quanto appena descritto.

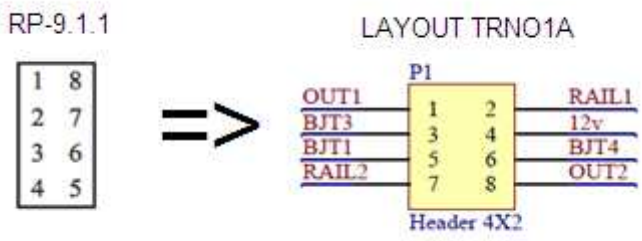


Figura 60 – differenze nella numerazione dei pin del connettore tra caso ideale e caso implementato

1.6.4 LM2937ES-3.3

Questo trasformatore ha la funzione di convertire la tensione d'alimentazione in ingresso a *12V* alla tensione più bassa pari a *3.3V* adatta per alimentare il microcontrollore (ricordiamo che solo alcuni pin del *PIC32* tollerano fino a *5V*).
Questo dispositivo richiede in uscita, il più vicino possibile, un condensatore adatto a garantire una tensione in uscita stabile: la sua *resistenza serie equivalente* (*ESR*) è un parametro critico del progetto, ma grazie alla speciale circuiteria di compensazione presente nel modulo la criticità diminuisce. La massima corrente in uscita dal dispositivo è pari a *500mA*. In ingresso inoltre può essere necessario inserire un ulteriore capacità se il regolatore è distante più di *7.5 cm* dai condensatori del filtro dell'alimentazione.
Ecco infine l'immagine del regolatore ed uno schema circuitale su cui viene fatto il calcolo della potenza dissipata dal dispositivo:

$$I_{IN} = I_L/I_G \ , \ P_D = (V_{IN} - V_{OUT})I_L + V_{IN}I_G$$

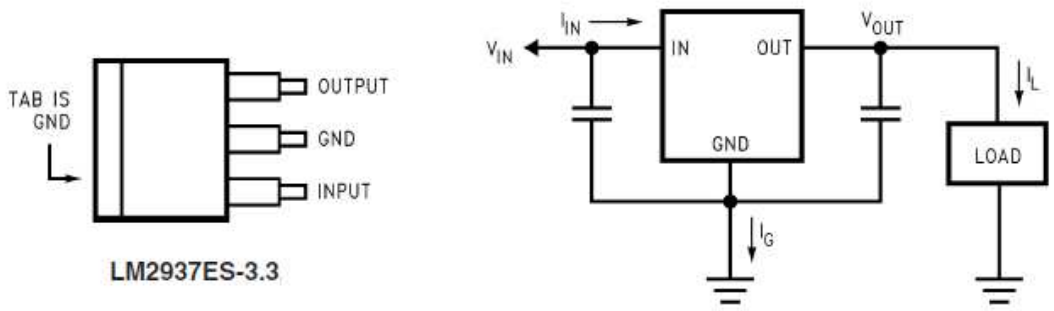


Figura 61 - package LM2937ES-3.3 e schema circuitale per il calcolo della potenza dissipata

1.6.5 Sensori

I sensori di cui dispone la scheda sono tre partitori di tensione e uno *shunt* con amplificatore per la misura.
I tre partitori di tensione vengono utilizzati per la misura della *fem* (due partitori) e per la misura della $V_{dc\ bus}$ (un partitore). Tramite la resistenza di *shunt* invece si misura la corrente entrante nel motore.
Nella pagina seguente è mostrato lo schema circuitale che mostra come sono stati collegati i tre partitori e la resistenza di *shunt*.

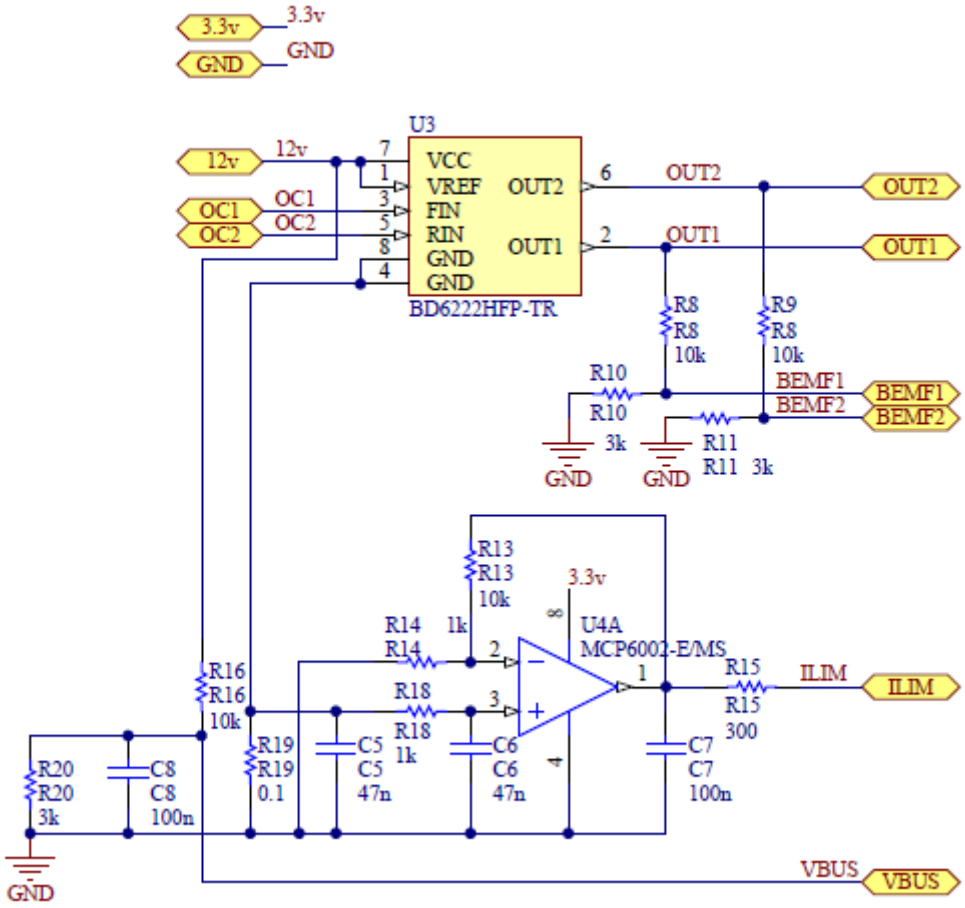


Figura 62 – schema circuitale raffigurante i tre partitori di tensione e il sensore di corrente con relativo amplificatore

1.7 Protocollo RS-232

Questo standard definisce un'interfaccia seriale a bassa velocità di trasmissione per lo scambio di dati tra dispositivi digitali. Collegando tramite un cavo fisico due apparecchiature elettroniche dotate di una porta *RS-232* è possibile realizzare una comunicazione tra di loro.
L'interfaccia *RS-232* ridotta (ovvero solo asincrona) usa un protocollo di trasmissione seriale di tipo asincrono. Seriale significa che i bit di informazione sono trasmessi uno alla volta su di un solo cavo fisico. Asincrono significa che i dati sono trasmessi, *byte per byte*, anche in maniera non consecutiva e che non viene utilizzato un segnale comune di *clock* per sincronizzare la trasmissione con la ricezione. Ovviamente per poter interpretare i dati sia il ricevitore che il trasmettitore devono essere dotati di un *clock locale*: la sincronizzazione viene fatta sulla linea dei dati in corrispondenza della prima transizione.
Nello standard *RS-232* la trasmissione è binaria, ovvero vengono utilizzati solamente due livelli, uno alto e uno basso; di conseguenza come unità di misura della velocità di trasmissione può essere usato sia il *baud rate* che il *bps*, in modo del tutto equivalente.
Infatti il *baud rate* indica il numero di transizioni al secondo che avvengono sulla linea e il *bps* quanti bit al secondo vengono trasmessi lungo la linea: potendo definire i due livelli utilizzando un solo bit, le due unità di misura in questo standard si equivalgono.
Vediamo quindi come appare un segnale *RS232* rappresentante il valore *01001011*.

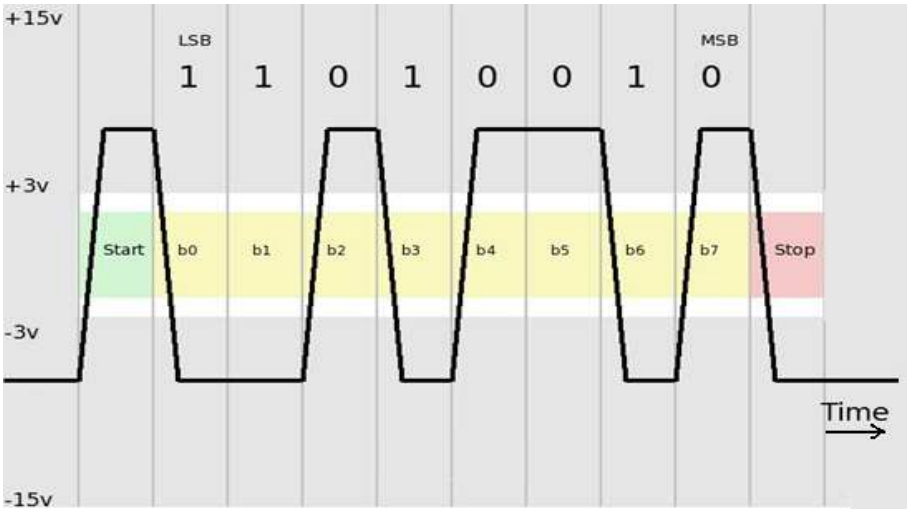


Figura 63 - esempio di segnale RS-232

Il protocollo permette di utilizzare tensioni che vanno da *±5V* a *±15V*, ma classicamente viene utilizzata una tensione pari a *±12V* (capita a volte che venga diminuita a *±6V* così da garantire minori emissioni elettromagnetiche (disturbi) e favorire maggiori velocità di trasmissione).
Da notare che questo standard prevede una codifica a logica negativa: il segnale alto *(+12V*) rappresenta lo *0 logico* e il segnale basso (*-12V*) rappresenta l'*1 logico*. L'interfaccia elettrica ha

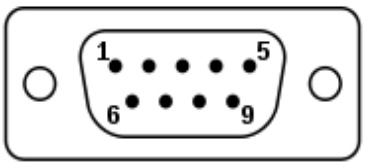
inoltre una soglia di commutazione di *±3V* per difendersi dai disturbi elettrici: per passare da uno stato all'altro non basta oltrepassare gli *0V* ma bisogna superare i *3V* di segno opposto.

Ciascun bit dura l'inverso del baud rate. Nello stato di riposo si è all'*1 logico*; la prima transizione da *1* a *0*, dovuta al bit di inizio (*start*), indica l'inizio della trasmissione. Dopo di questa seguono gli 8 bit di informazione utile, dal meno (*LSB*) al più significativo (*MSB*). Alla fine troviamo un periodo di riposo lungo almeno un bit di fine (*stop*), dopodichè una nuova trasmissione può eventualmente iniziare.
Esiste la possibilità di trasmettere 5-6-7-9 bit di informazione utile al posto dei classici 8 e di aggiungere un bit di parità per prevenire errori nella trasmissione. Il numero di bit di fine è inoltre variabile (tipicamente 1 o 2).
Nei *PC* il formato del pacchetto ricetrasmesso è indicato da una sigla del tipo *xyz*: *x* indica il numero di bit di informazione utile, *y* indica se viene utilizzato o no un bit di parità, *z* indica il numero di bit di fine. Ad esempio, la sigla 7e2 indica che vengono trasmessi 7 bit dati utili, che viene aggiunto un bit di parità pari e che sono 2 i bit di fine.
Lo standard *RS-232* prevede una velocità da 75 a *19200 baud*; tuttavia nei *PC* le interfacce *RS-232* arrivano anche a *115000 baud* senza creare particolari problemi di interconnessione.
Prima di iniziare la trasmissione sia trasmettitore che ricevitore devono ovviamente accordarsi sulla modalità di trasmissione dei dati. Essendo gli unici elementi di sincronizzazione dati dai fronti del bit di inizio e dei bit di fine ed essendo il campionamento in ricezione di norma fatto al centro di ciascun bit, risulta fondamentale garantire la corretta durata di ciascun bit, dato che l'errore massimo ammesso per un pacchetto standard *8n1* (n = senza bit di parità) è di circa il *5%* della frequenza di *clock*, corrispondente alla durata di mezzo bit.
Per questo motivo è praticamente imposto l'uso di oscillatori al quarzo che garantiscono una precisione migliore dell'1%.
Elenchiamo quindi i parametri elettrici dettati dallo standard *RS-232*:

1. la tensione di uscita ad un trasmettitore deve essere compresa in valore assoluto tra *5V* e *25V*.
2. il ricevitore deve funzionare correttamente con tensioni di ingresso in valore assoluto tra i *3V* e i *25V.*
3. l'impedenza di uscita del trasmettitore deve essere sempre maggiore di *300Ω*.
4. anche a dispositivo spento, l'impedenza d'ingresso deve essere compresa tra i 3 e i 7 *KΩ*.
5. lo *slew-rate* (ovvero la pendenza dei fronti transitori) deve essere minore di *30V/us* per evitare eccessive emissioni elettromagnetiche.

Il connettore *RS-232* utilizza due segnali in modalità "*Full Duplex*" più altri di controllo e la massa comune. La modalità "*Full Duplex*" permette di ricevere e trasmettere contemporaneamente.
Mentre prima dell'avvento dell'*IBM Personal Computer AT (Advanced Technology)*, intorno al 1984, era di uso comune il connettore DB-25 a 25 pin, poi si è optato per il connettore DE-9 a 9 pin.
Il motivo di tanti pin è legato alla nascita dell'interfaccia *RS-232*: essa infatti nacque unicamente con lo scopo di connettersi ad un Modem ed è quindi fornita di tutti i controlli utili per gestire tale

apparecchiatura. Per ricevere e trasmettere un segnale in modo asincrono bastano tuttavia tre fili: massa, ricezione e trasmissione. Gli altri segnali servono per l'*handshake* tra *PC* e periferica, ovvero per sincronizzare in hardware la comunicazione.
Mostriamo infine la piedinatura del connettore *DE-9 maschio* (lato computer).



Pin 1: Data Carrier Detect (informa il terminale se sul canale è stato rilevato il segnale di portante)
Pin 2: Ricezione Dati
Pin 3: Trasmissione Dati
Pin 4: Data Terminal Ready (informa se il terminale dati è pronto)
Pin 5: Massa Dati di riferimento
Pin 6: Data Set Ready (informa se il modem è pronto ad operare)
Pin 7: Request To Send (informa il modem dell'eventuale presenza di dati da trasmettere)
Pin 8: Clear To Send (informa il terminale se il canale dati è pronto per la trasmissione)
Pin 9: Ring Indicator (informa il terminale se è in arrivo una chiamata sul canale di comunicazione)

Figura 64 - connettore DE-9

1.8 MiWi wireless solution

1.8.1 Descrizione generale

Sempre più spesso si richiede che svariati tipi di dispositivi implementino una comunicazione di tipo wireless. I benefici sono ovvi: riduzione dei costi e semplicità di installazione. Infatti una comunicazione wireless non richiede cavi né altro hardware ad essi associato (amplificatori, filtri, …) ed elimina le difficoltà e i costi connessi all'installazione dei cavi in luoghi in cui ciò risulta molto problematico.
Il protocollo MiWi si basa fondamentalmente sullo standard *IEEE 802.15.4*^{TM}, lo standard alla base di tutte le reti *WPAN* (*Wireless Personal Area Network*). In particolare sono state aggiunte varie modifiche per poter supportare i trasmettitori *RF* prodotti dalla Microchip, e fornire un protocollo più semplice e immediato per quel particolare sottogruppo delle reti *WPAN* che sono le reti caratterizzate da un basso tasso di trasmissione di dati, distanze ridotte fra i nodi, bande attorno ai *2.4GHz* e costi ridotti.
Il protocollo *MiWi* utilizza l'interfaccia *MiMAC* per comunicare col trasmettitore *Microchip* implementato (livello fisico, o *physical layer*) e l'interfaccia *MiApp* per interagire col livello applicazione.

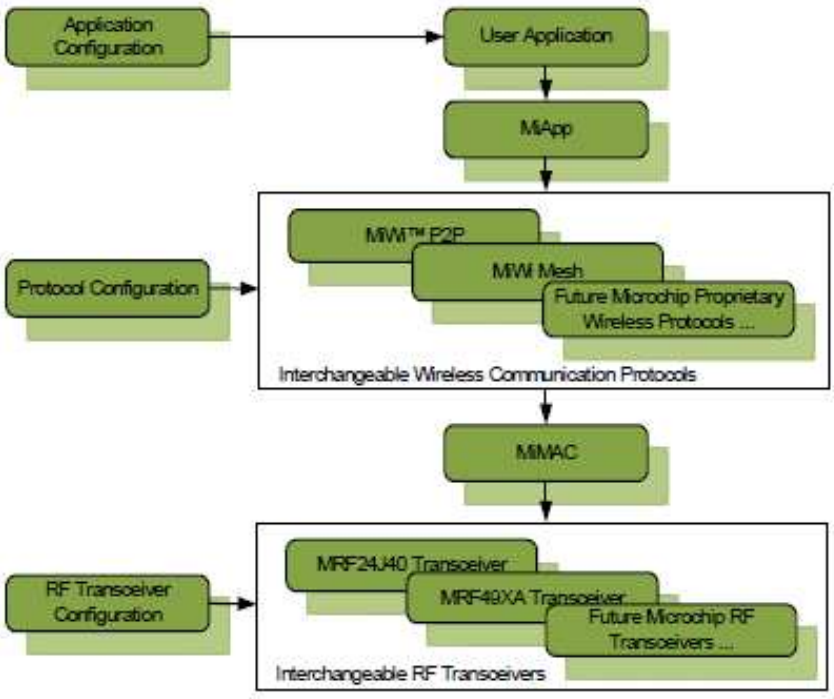


Figura 65 – MiWi Wireless Solution

Descriviamoli separatamente. La prima funzione di un protocollo di comunicazione wireless è trasmettere o ricevere dati fra due nodi. Il sotto-livello *MAC* (*Media Access Controller*), facente parte del livello collegamento (*Data Link layer*), provvede alle funzionalità di accesso al canale, indirizzamento, trasmissione/ricezione che vengono sopra al livello fisico, che gestisce i singoli dati a livello di segnali fisici. Essendo infiniti i modi di implementare il livello fisico, il *MAC* è il primo livello dove sia possibile effettuare una standardizzazione. Tramite il *MiMAC* viene data possibilità di implementare lo *stack* per svariati tipi di trasmettitori, non curandosi ovvero di quale particolare trasmettitore sia stia utilizzando.
Il *MiApp* invece definisce l'interfaccia di programmazione tra il livello dell'applicazione e il protocollo *Microchip* utilizzato, e permette una standardizzazione di più alto livello che agisce permettendo l'interscambiabilità dei vari protocolli di comunicazione wireless che la *Microchip* offre.
Il tutto è riassunto nella Figura 65.

1.8.2 MiWi protocol: nodi, rete, comunicazione, …

Il protocollo *MiWi* definisce tre tipi di elementi, in base alla loro funzione nella rete: il coordinatore *PAN*, i coordinatori e i dispositivi terminali.

- Il coordinatore *PAN* è quello che forma la rete, decidendo il canale di comunicazione e il *PANID*, ovvero l'*ID* della rete stessa. Ovviamente ve ne è uno solo, e un il suo compito all'interno della rete è di gestire e tenere in memoria gli indirizzi dei vari elementi facentene parte.
- Un coordinatore è un elemento opzionale, serve ad aumentare il range fisico della rete, può avere compiti di controllo e monitoraggio.
- I dispositivi terminali svolgono funzioni di controllo e monitoraggio.

A seconda di come vengono a collegarsi tali elementi si hanno diverse configurazioni della rete. Le principali sono:

- A stella (*star network*). E' la rete più semplice, è formata dal coordinatore *PAN* e da svariati dispositivi terminali, ognuno dei quali è connesso unicamente al *PAN coordinator*.
  La differenza fra *FFD* e *RFD End Device* sta nel diverso utilizzo dei dispositivi terminali: *FFD* sta per *Full Function Device*, ovvero dispositivo a piene funzioni, e sta a indicare un dispositivo, spesso collegato a una rete di alimentazione, tale per cui gli viene permesso di svolgere ogni genere di funzionalità implementata; *RFD* sta per *Reduced Function Device*, ovvero dispositivo a funzioni ridotte, e rappresenta il caso di quei dispositivi alimentati a batteria per i quali molte funzioni vengono disabilitate per favorire un minore consumo energetico.

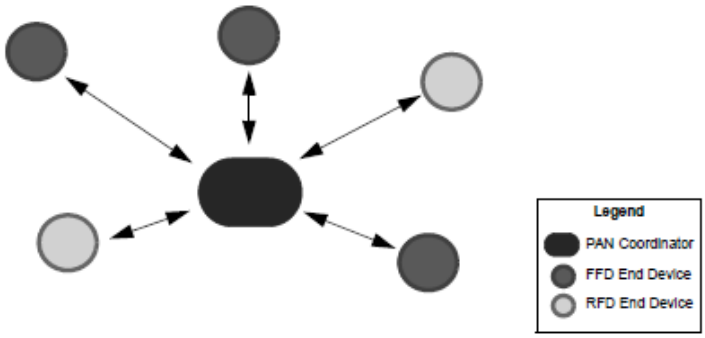


Figura 66 – rete a stella (Star network)

- Ad albero (*tree network*). In questo caso si ha l'aggiunta di altri coordinatori, che si collegano direttamente al *PAN coordinator* che assume il senso di radice dell'albero. I vari dispositivi terminali si collegano o direttamente al *PAN coordinator* o agli altri coordinatori.

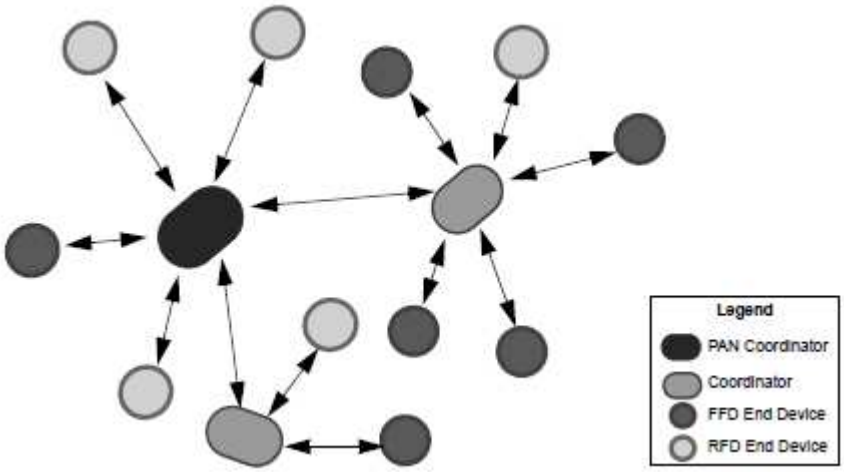


Figura 67 – rete ad albero (Tree network)

Definiti gli elementi e i tipi di connessioni possibili, il protocollo definisce quindi un aspetto fondamentale per la gestione e il funzionamento della rete: la creazione degli indirizzi. Per quanto riguarda tale aspetto il protocollo *MiWi* prevede, basandosi sullo standard *IEEE 802.15.4*^{TM}, tre diversi tipi di indirizzo.

- Indirizzo *EUI* (*Extended organizationally Unique Identifer*). Questo indirizzo di *8 Byte* è globalmente unico, i primi 3 byte son gestiti dallo standard *IEEE 802.15.4*^{TM}, gli altri *5 byte* dall'utente.
- Indirizzo *PAN* (*PANID*). E' l'indirizzo che definisce un gruppo di nodi che formano una certa rete. E' composto da *2 Byte*.

- Indirizzo corto (*Short Address*). E' l'indirizzo dei dispositivi terminali, è formato da *2 Byte* ed è fornito dal nodo padre al nodo figlio. Serve per inviare e ricevere dati all'interno della rete, ed all'interno di tale rete è ovviamente unico (ma non globalmente chiaramente).

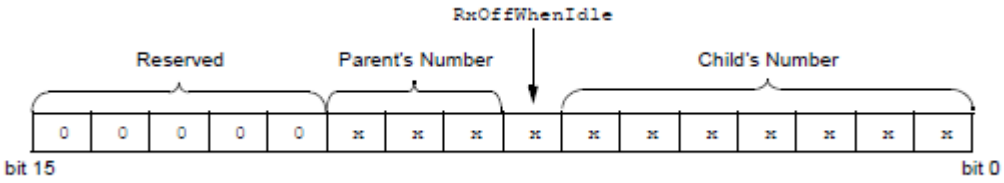


Figura 68 – short address

Come si può facilmente dedurre il numero massimo di nodi padri (ovvero coordinatori) è 8, in quanto si hanno a disposizione 3 bit. Il *RxOffWhenIdle* è un singolo bit, un *flag*, che se settato alto indica che il trasmettitore *RF* verrà spento quando il dispositivo è in *stato idle*.

Una volta che la rete è formata in termini di dispositivi, indirizzi e collegamenti fra essi è necessario definire come avvenga la trasmissione/ricezione di dati e il routing[5]. Al di sopra del livello *MAC*, dove vengono impostati i pacchetti in termini di sincronizzazione e *payload*, si trova il *MiWi Protocol Header*, che contiene tutte le informazioni necessarie per processare e indirizzare i pacchetti.

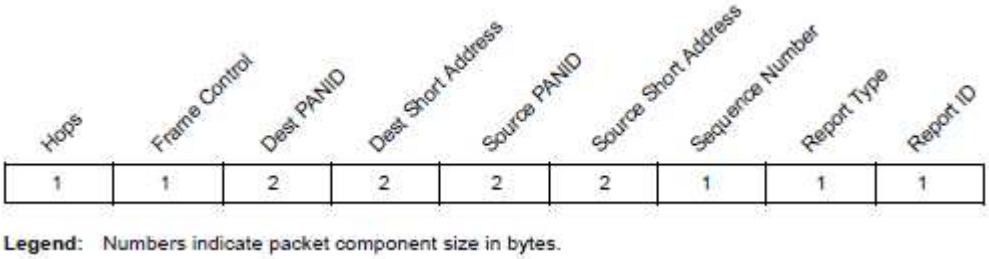


Figura 69 – MiWi protocol haeder

I campi definiti in figura, tolti gli indirizzi del mittente e del destinatario che sono di immediata comprensione, assumono i seguenti significati:

- *Hops*: il numero di *hops* massimo che un pacchetto può avere, pari al numero massimo di ritrasmissioni che può effettuare:
- *Frame Control*: gestisce il comportamento del pacchetto;
- *Sequence Number*: è un numero che viene utilizzato per tracciare il pacchetto mentre viaggia nella rete;
- *Report Type* e *Report ID*: indicano il tipo (prefissato o creato dall'utente) e il contenuto dei *Reports* (vedesi dopo nella descrizione del trasporto dei pacchetti dati).

Per gestire il *routing* il protocollo *MiWi* utilizza l'allocamento degli indirizzi dei nodi padri del nodo a cui si voglia inviare un pacchetto. Uno dei punti fondamentali di un *algoritmo di routing* infatti è determinare il prossimo *hop* di un pacchetto in uscita. Per far questo il protocollo *MiWi* usa

[5] Il **routing** è il processo di selezione dei percorsi all'interno della rete lungo i quali indirizzare il traffico dati.

l'algoritmo già definito dallo standard *IEEE 802.15.4*^{TM}: quando un dispositivo subentra nella rete, per prima cosa invia un pacchetto di riconoscimento (*beacon request packet*); tutti i coordinatori che ricevono tale pacchetto ne inoltrano immediatamente un altro ai dispositivi ad essi connessi fornendo le informazioni di cui sono venuti a conoscenza. In aggiunta a tale standard il protocollo *MiWi* prevede che *3 Byte* siano aggiunti al *payload* del messaggio *beacon*, contenenti le seguenti informazioni:

- *Protocol ID* (*1 Byte*): serve a distinguere fra una rete *MiWi* e una qualsiasi altra rete basata sullo *standard IEEE 802.15.4*^{TM};
- *Version Number* (*1 Byte*): il codice delle specifiche;
- *Local Coordinators* (*1 Byte*): indica quale coordinatore è visibile dal coordinatore che sta inviando il *beacon*. Attraverso la rete di coordinatori locali ogni coordinatore impara nuovi percorsi verso i vari dispositivi senza dover mandare singoli messaggi di richiesta. Risiede in questo il concetto di conoscenza dei coordinatori vicini.

Fatto questo l'invio dei pacchetti a un singolo dispositivo della rete (non coordinatore) diviene molto facile, e viene schematizzato dalla seguente figura.

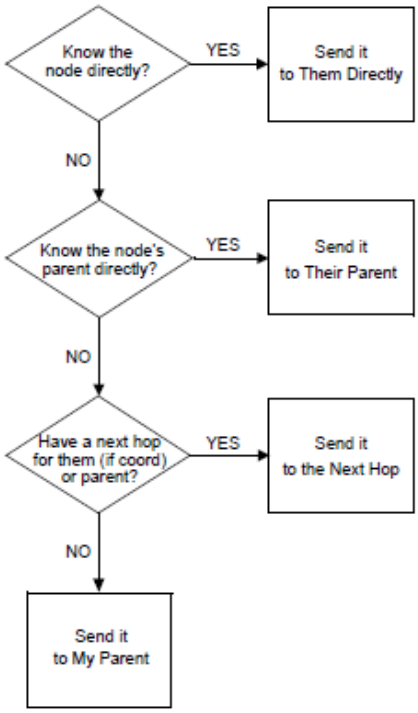


Figura 70 – schema di invio di un pacchetto a un singolo dispositivo della rete

Definito il *routing* è necessario definire come venga gestito il trasporto dei pacchetti. Questo avviene attraverso l'utilizzo di particolari pacchetti detti *Reports*, di cui il protocollo ne permette la creazione fino a un numero massimo di 256, di cui i primi 10 prefissati dallo standard.

Senza entrare nel dettaglio si fornisce come esempio visivo l'elenco di tali *reports*, nei cui nomi, previa conoscenza dei termini (che vengono spiegati nelle note a piè di pagina[6]), è insito il loro scopo.

Report Type	Report ID	Name
00h	10h	OPEN_CLUSTER_SOCKET_REQUEST
	11h	OPEN_CLUSTER_SOCKET_RESPONSE
	12h	OPEN_P2P_SOCKET_REQUEST
	13h	OPEN_P2P_SOCKET_RESPONSE
	20h	EUI_ADDRESS_SEARCH_REQUEST
	21h	EUI_ADDRESS_SEARCH_RESPONSE
	30h	ACK_REPORT_TYPE
	40h	CHANNEL_HOPPING_REQUEST
	41h	RESYNCHRONIZATION_REQUEST
	42h	RESYNCHRONIZATION_RESPONSE
01h-FFh	00h-FFh	Available for use

Tabella 4 – commands definiti dal protocollo MiWi

1.8.3 MiWi P2P protocol

Guardando la Figura 65 si nota come la *Microchip* metta a disposizione più tipi di protocolli di comunicazione *MiWi* (*P2P, Mesh, …*), ognuno dei quali è in grado di interfacciarsi al livello fisico col *MiMAC* e al livello applicativo col *MiApp*. Nel paragrafo precedente si è data una descrizione del protocollo *MiWi* in termini generali, mettendo in mostra quello che è in grado di realizzare e il come lo realizza. Ora ci concentreremo non più sulle caratteristiche generali ma sulle caratteristiche specifiche di uno dei protocolli *MiWi*, il *MiWi P2P*. Nel farlo seguiremo le linee descrittive del paragrafo precedente, mettendo però in luce per ogni punto le caratteristiche di questo specifico protocollo rispetto all'idea generale.
In generale questo protocollo modifica le specifiche del livello *MAC* così come sono definite dallo standard *IEEE 802.15.4*^{TM}, aggiungendo dei comandi che semplificano il processo di *handshaking*, la disconnessione e il *channel hopping*.
Per quanto riguarda i tipi di dispositivi il protocollo supporta tutti i dispositivi precedentemente descritti.
A livello di rete invece ci sono solo due tipologie di connessione fra i vari dispositivi: reti a stella e di tipo *peer-to-peer*. La rete a stella è già stata descritta; per quanto riguarda la rete *peer-to-peer* è presente, anche in questo caso, un coordinatore *PAN* da cui partono tutte le comunicazioni, ma a differenza della rete a stella quando un dispositivo terminale si aggiunge alla rete non deve stabilire una connessione diretta con il coordinatore *PAN*.

[6] **Cluster**: per *cluster* si intende un insieme di nodi (da 3 in su) che formano una rete.
Socket: per *socket* si intende una porta virtuale attraverso la quale i dati passano dalla rete all'applicazione, e viceversa.
Ack: l'*acknowledged* che è presente nel *MAC Header*.

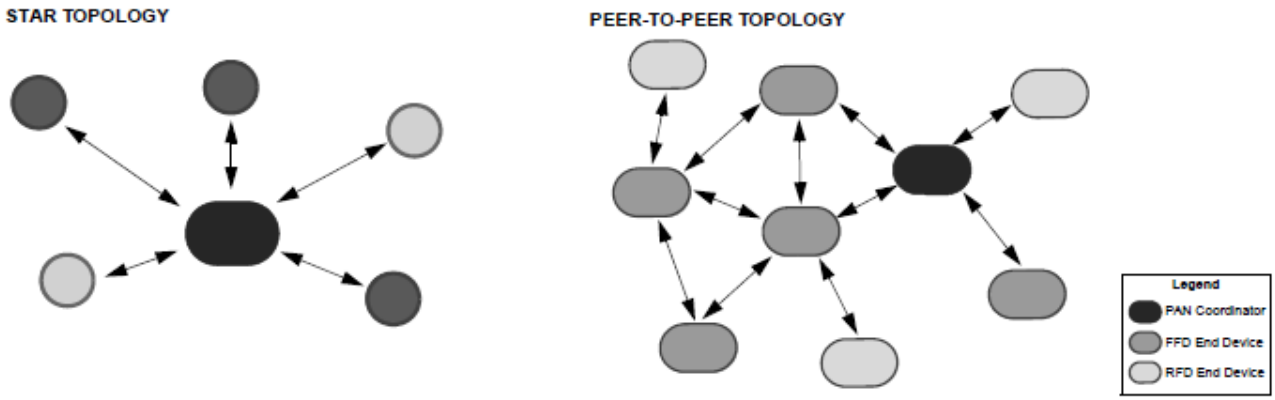


Figura 71 – rete a stella e rete Peer-to-Peer (P2P)

Per quanto riguarda gli indirizzi il *MiWi P2P* definisce e utilizza due degli indirizzi precedentemente descritti: l'*EUI*, e lo *Short Address*. In particolare lo *Short Address* viene usato solo quando si trasmette un pacchetto broadcast, viceversa si utilizza sempre l'*EUI*.
Per quanto riguarda il *PANID* è anche esso definito dal protocollo, ma con la corrente implementazione non viene utilizzato. Infatti ha senso fornire il *PANID*, sia del destinatario che della sorgente, solo se si ha una comunicazione *inter-PAN*, ovvero fra due reti aventi ognuna un proprio *PAN coordinator*. Siccome per la corrente implementazione del *MiWi P2P* tutte le comunicazioni sono *intra-PAN*, il *PANID* può essere non specificato.
Il formato del pacchetto inviato è il seguente.

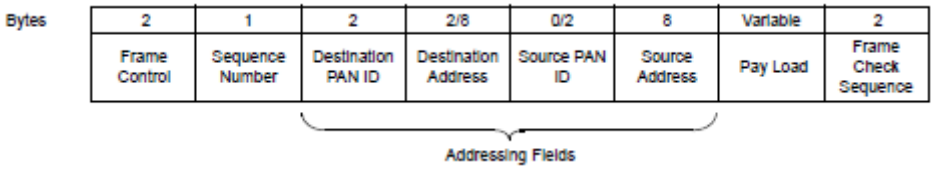


Figura 72 – formato pacchetto secondo il protocollo MiWi P2P

Per quanto detto i campi *Destination PANID* e *Source PANID* risultano in realtà settati a zero. I campi *Frame Control* e *Frame Check Sequence* sono campi di controllo che gestiscono la corretta comunicazione, mentre il *Sequence Number* è un numero random in 8 bit che ogni volta che un pacchetto viene inviato è incrementato di uno. Svolge funzione di tracciamento e riconoscibilità di un dato pacchetto.

Definiti i dispositivi, le tipologie di rete, gli indirizzi e il formato del pacchetto, ci si occupa, come nel precedente paragrafo, dei metodi di trasmissione e ricezione, parti in cui maggiore è la peculiarità di questo protocollo.

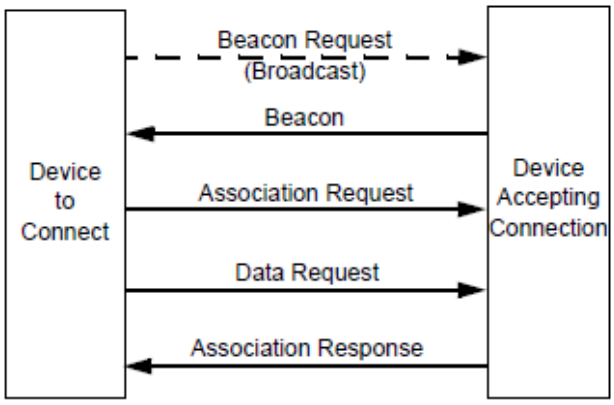
- Trasmissione. Son previsti due metodi di trasmissione, *Broadcast* e *Unicast*. Quando si trasmette in modalità *broadcast* la destinazione di un pacchetto sono tutti i dispositivi attivi nel range radio della rete. E' l'unica modalità di trasmissione del *MiWi P2P* che utilizza solamente gli *short address* come indirizzi del destinatario.

Viceversa se si utilizza la modalità *unicast* il pacchetto inviato ha un'unica destinazione e utilizza l'*EUI* come indirizzo del destinatario. Per questa modalità è richiesto l'invio di un *Acknowledgement*.

- Ricezione. Quando un dispositivo spegne la sua radio può ricevere un messaggio solo dal dispositivo a cui è connesso, previo invio da parte di quest'ultimo di un *Data Request Command* al suo *peer* di connessione. Il pacchetto, finchè non viene ricevuto dal dispositivo in *idle* viene conservato dal trasmettente nella sua memoria *RAM*.

A differenza dello standard *IEEE 802.15.4™* il protocollo *MiWi P2P* semplifica molto la parte inerente all'invio di messaggi di *beacon* e in generale alla procedura di *handshaking*. In un *beacon network* ogni dispositivo può inviare dati solo in precisi slot temporali, che vengono assegnati dal coordinatore *PAN* tramite l'invio un frame detto appunto *beacon frame*. Tutti i dispositivi si sincronizzano col *beacon frame*. In un network non beacon, come appunto il *MiWi P2P*, ogni dispositivo può inviare dati in ogni istante quando il livello di energia (rumore) è prossimo al livello predefinito. Ciò implica un aumento del consumo energetico dei dispositivi *FFD* che devono sempre mantenere il trasmettitore acceso.
Per quanto riguarda la procedura di *handshaking* vale la pena ricordare come si presenta per lo standard *IEEE 802.15.4™*, progettato per un *beacon network*.
La prima cosa che compie un dispositivo appena si aggiunge alla rete è appunto l'*handshake*. Questa procedura consta di 5 passi:

1. Il nuovo dispositivo invia una *beacon request;*
2. Tutti i dispositivi in grado di connettersi agli altri dispositivi (i dispositivi attivi e connessi ovvero) rispondono con un messaggio *beacon*;
3. Il nuovo dispositivo raccoglie tutti i *beacon* e decide quale di questi utilizzare per stabilire una procedura di *handshake*. Fatta tale scelta invia al dispositivo scelto un *association request command*;
4. Dopo un certo tempo il nuovo dispositivo invia anche un *data request command*.
5. Il dispositivo scelto risponde con un *association response*.

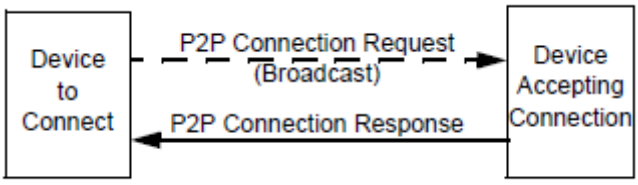


lshake in una rete beacon

L'*handshake* risulta essere la procedura di associazione a una rete e ricerca del nodo padre, operazione che viene effettuata passando in rassegna tutti i possibili nodi e scegliendone uno solo fra questi. Come si può notare la procedura di *handshaking* così concepita è particolarmente complicata. Il protocollo *MiWi P2P* è concepito per semplificare questa operazione, e rendere possibile una connessione diretta e più rapida per reti a stella o di tipo *P2P*.
Proprio per questo il *MiWi P2P* ha sviluppato una sua procedura di *handshaking*.

Tale procedura è così concepita:

1. Il nuovo dispositivo invia un *P2P connection request command*;
2. Ogni dispositivo attivo nel range radio risponde con con un *P2P connection response command* che finalizza la connessione.



e nel protocollo MiWi P2P

Si è quindi di fronte a un processo da-uno-a-tanti che porta a instaurare connessioni multiple.

Vengono ora mostrati i principali comandi (la maggior parte di essi son già stati introdotti nelle righe precedenti) che permettono di gestire l'invio/ricezione dei dati, e il relativo significato.

Command Identifier	Command Name	Description
0x81	P2P Connection Request	Request to establish a P2P connection. Usually broadcast to seek P2P connection after powering up. Alternately, unicast to seek an individual connection. Also used for active scan functionality.
0x82	P2P Connection Removal Request	Removes the P2P connection with the other end device.
0x83	Data Request	Similar to the IEEE 802.15.4™ specification's Data Request command (0x04), a request for data from the other end of a P2P connection if the local node had its radio turned off. Reserved for the previously sleeping device to request the other node to send the missed message (indirect messaging).
0x84	Channel Hopping	Request to change operating channel to a different channel. Usually used in the feature of frequency agility.
0x91	P2P Connection Response	Response to the P2P connection request. Also can be used in active scan process.
0x92	P2P Connection Removal Response	Response to the P2P connection removal request.

Tabella 5 – commands definiti dal protocollo MiWi P2P

L'*active scan* di cui si parla in alcune delle descrizioni della tabella è il processo di acquisire informazioni a proposito della propria rete (*PAN*). Tramite l'*active scan* vengono determinati:

- Il canale operativo del dispositivo;
- La forza del segnale;
- Il codice identificativo della rete *PAN*.

La durata della scansione e i canali da analizzare son decisi prima dell'inizio del processo.
L'*energy scan* invece ha la funzione di analizzare tutti i canali disponibili e trovare il canale con rumore minore.

Per completare si mostra il *flowchart* di un processo di connessione con *Active scan* e *Energy scan* abilitati, così come risulta essere nel nostro progetto.

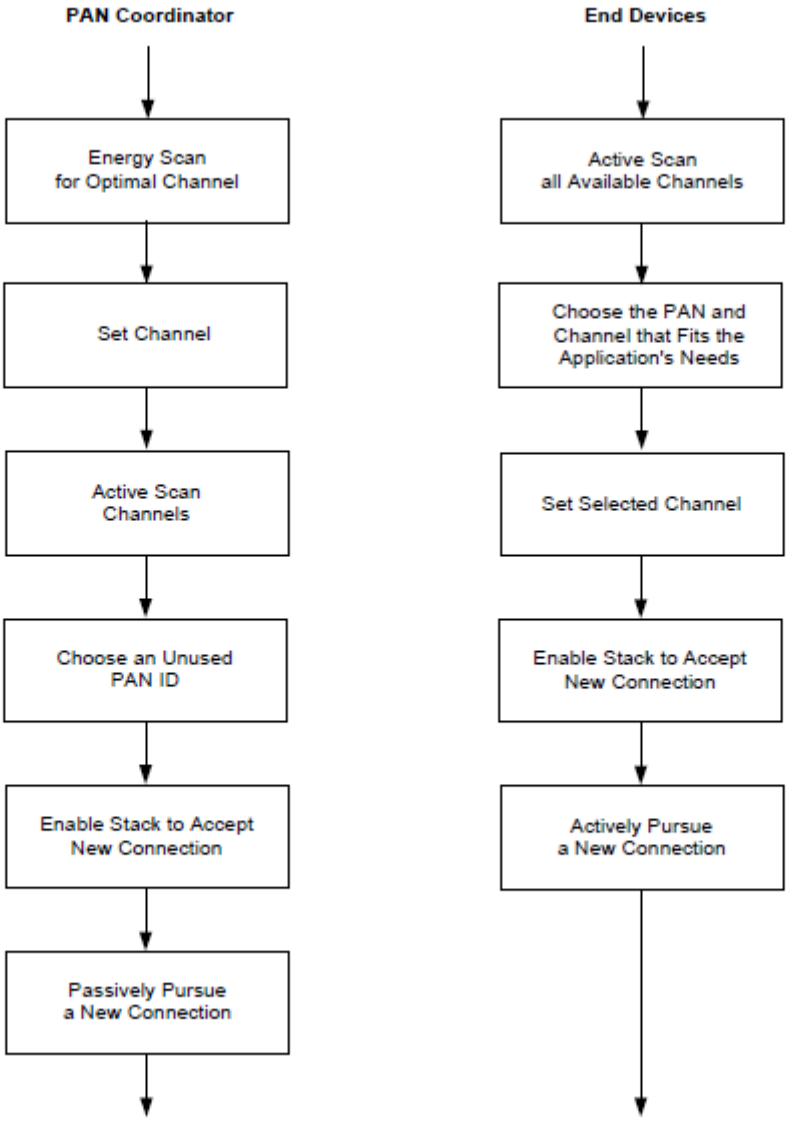


Figura 75 – flowcharts di un processo di connessione con active scan ed Energy scan attivi

1.9 Linguaggio Visual Basic. Linguaggio ad eventi

Il linguaggio Visual Basic è un linguaggio a eventi (event-driven programming language) di terza generazione, associato all'ambiente di sviluppo Visual Basic IDE fornito da Microsoft.
La programmazione a eventi è un paradigma di programmazione dell'informatica. Mentre in un programma tradizionale l'esecuzione delle istruzioni segue percorsi fissi, che si ramificano soltanto in punti ben determinati predefiniti dal programmatore, nei programmi scritti utilizzando la tecnica *a eventi* il flusso del programma è largamente determinato dal verificarsi di eventi esterni.
Invece di aspettare che un'istruzione impartisca al programma il comando di elaborare una certa informazione, il sistema è predisposto per eseguire all'infinito un *loop* di istruzioni all'interno del quale vi sono istruzioni che verificano continuamente la comparsa delle informazioni da elaborare (potrebbe trattarsi della creazione di un file in una certa cartella o della pressione di un tasto del mouse o della tastiera), e quindi lanciare l'esecuzione della parte di programma scritta appositamente per gestire l'evento in questione. La programmazione ad eventi prevede che una parte di codice venga eseguita quando avviene un evento; per riconoscere un evento si puo' eseguire un *loop infinito* (tecnica di *polling*) oppure utilizzare tecniche di interruzione (quindi senza loop); di solito eventi legati all'*hardware* operano mediante un meccanismo di interruzione (*interrupt*).
Programmare *a eventi* vuol dire, quindi, ridefinire in modo personalizzato le azioni "*di default*" con cui il sistema risponde al verificarsi di un certo evento.
Visual Basic in particolare è stato sviluppato col fine di permettere lo sviluppo rapido di applicazioni (*RAD – Rapid Application Development*) per interfacce grafiche (*GUI – Graphical User Interface*). L'interfaccia grafica è un tipo di interfaccia utente che consente all'utente di interagire con la macchina manipolando oggetti grafici convenzionali (pulsanti, manopole, …).
Questo linguaggio permette comunque di creare applicazioni anche molto complesse, non solo interfacce grafiche.
Il linguaggio *VB* deriva dal linguaggio *BASIC*, e mantiene molte differenze rispetto ai linguaggi derivati dal *C.* Infatti:

- Le dichiarazioni terminano con l'espressione '*End*', per esempio '*End if*', invece di racchiuderle nelle classiche parentesi graffe *{…}*;
- Non è possibile fare assegnamenti multipli, ovvero *A=B=C* non significa che *A, B* e *C* abbiano lo stesso valore;
- I vettori son dichiarati definendone il limite superiore e inferiore;
- I numeri interi (formato *integer*) sono automaticamente trasformati in numeri reali nelle espressioni che comprendono l'operatore di divisione '/', permettendo di ottenere il risultato esatto e non troncato;
- La costante *Boolean TRUE* ha il valore numerico di -1. Questo perché i dati *Boolean* sono salvati in interi a 16 bit. In questo genere di costrutto -1 corrisponde all'esadecimale 1s (*TRUE*), mentre 0 a 0s (*FALSE*). Tutto questo torna utile quando si effettuano operazioni sui singoli bit.

Il compilatore di *VB* è condiviso con gli altri linguaggi raggruppati nell'ambiente *Visual Studio* (*C, C++*) ma restrizione commerciali impediscono per tale linguaggio la creazione di target (*Windows model DLL*).

Programmare in *VB*, inteso come ambiente di sviluppo, e non solo come linguaggio, significa posizionare dei controlli (pulsanti, tasti, …) su di un *form* (ovvero una finestra - *window*), specificando gli attributi e le azioni di questi componenti e aggiungendone funzionalità tramite scrittura nel codice. Il posizionamento dei controlli avviene utilizzando tecniche *drag-and-drop*. Un *tool* è usato per piazzare i controlli. Gli attributi di tali controlli sono modificabili in opportune barre (*VB* fornisce di partenza dei valori di default). Ogni volta che si clicca su un particolare controllo posizionato nel form, il programma ci trasla nella rappresentazione in codice della finestra, inserendo in automatico il particolare codice di quel controllo. Agendo su questo codice è possibile quindi implementare altre funzioni più complesse.
Il grande vantaggio dell'ambiente di sviluppo *VB* è la capacità di passare da una veste grafica in cui si opera sui vari comandi, li si posiziona nel *form*, si impostano le loro funzionalità di *default*, a una veste '*di codice*' in cui ci si trova di fronte a quanto si è fatto graficamente ma espresso secondo la sintassi del linguaggio *VB*.

1.10 Conclusioni

Come era stato introdotto in questo capitolo ci si è occupati di fornire una trattazione il più accurata possibile delle tematiche alla base del progetto, descrivendo sia i vari componenti sia le problematiche ad essi correlati.
Conclusa tale trattazione, che permette di avere piena padronanza degli aspetti astratti alla base di tutto il lavoro, ci si addentrerà ora nella descrizione della soluzione pratica del progetto.

Secondo capitolo 2

2.1 Introduzione

Il secondo capitolo di questo elaborato di tesi si occupa di descrivere la realizzazione pratica del progetto, sviluppandosi quindi con un ordine che ricalca l'ordine logico (e cronologico) da noi seguito nell'affrontare le varie problematiche.
In primo luogo vengono mostrate le prove effettuate sul motore per ricavare i parametri necessari per la sua caratterizzazione. Fatto questo si mostrano i risultati della simulazione effettuata con *Simulink* sul modello del motore e sul modello del controllo di velocità.
Giunti a questo punto incomincia la descrizione del lavoro pratico vero e proprio. Viene infatti descritta come è stata sviluppata la comunicazione fra il generico *pc* utente e la chiave digitale connessa al trenino, per poi analizzare tutto il processo tramite cui si è giunti all'implementazione del controllo. Nel descrivere la genesi della comunicazione e del controllo l'attenzione sarà sempre incentrata nel mostrare le scelte effettuate a fronte dei vari problemi che via via si sono presentati, con particolare attenzione a motivarle e descriverne gli effetti. Minimo sarà il ricorso alla descrizione del codice vero e proprio, in quanto una volta effettuata una data scelta come poi questa si traduca in un adeguato codice *C* sorgente è un atto pratico di scarso interesse descrittivo.
Il capitolo si concluderà quindi con l'analisi e i commenti dei risultati ottenuti.

2.2 Prove sul motore, creazione del modello Simulink e simulazioni del controllo

2.2.1 Prove e modello del motore DC

Il modello del motore viene realizzato dopo avere ricavato i parametri del motore da prove sperimentali in laboratorio e dopo aver ipotizzato e verificato alcune ipotesi.
L'induttanza *L* viene trovata tramite misura con un ponte *RLC* imponendo un'alimentazione di *7.4V* e risulta pari a *1.117 mH*.

La resistenza *R* viene trovata tramite prova a rotore bloccato: infatti, a regime, a rotore bloccato, *R* è semplicemente data dal rapporto tra tensione di alimentazione e corrente assorbita (viene trascurata la caduta di tensione sulle spazzole).
Alimentando e misurando la corrente assorbita si ottiene una resistenza media *R* pari a *2.11 Ω*.
Realizziamo quindi una prova a rotore libero, alimentando il circuito a *3.0 V*: la corrente assorbita viene misurata pari a *113 mA* e la velocità del rotore a *2365 rpm*; la corrente a rotore bloccato è invece pari a *1350 mA*.
La differenza tra le due correnti (*1350mA – 113mA*) rappresenta il contributo di corrente della *fem*, assente nella prova a rotore bloccato. Di conseguenza, la *fem* è data dal prodotto tra questa differenza di corrente e la resistenza *R* ed è pari a *2.61 V*.
Dai valori misurati, abbiamo ricavato la costante di velocità *K (K=Kv=Km)*, data dal rapporto tra *fem* e velocità del rotore e pari a *1.103 mV/rpm*; la costante elettrica di macchina risulta invece pari a *0.529 ms* e data dal rapporto tra induttanza e resistenza.
Di conseguenza la corrente dovrebbe andare a regime in circa *2.64ms*, pari a 5 volte la costante elettrica (τ_{elet}).
Si è quindi cercato di stimare il momento d'inerzia del motore (J_m) tramite la coppia: calcolata infatti quest'ultima come prodotto tra la costante di velocità e la corrente assorbita a rotore libero, pari a *0.124 mN*$_m$, possiamo ricavare l'inerzia come il rapporto tra la coppia e la velocità del rotore, ottenendo $J_m = 52.431 \cdot 10^{-9}$.
Infine si è ipotizzata un momento d'inerzia resistente (J_r) nullo in quanto il motorino si presenta privo di carico e con un rotore di dimensioni molto ridotte; viceversa viene stimato un coefficiente di attrito dinamico $D_m = 1 \cdot 10^{-7}$, ottenuto tramite validazione del modello coi valori reali di tensione e velocità del rotore.

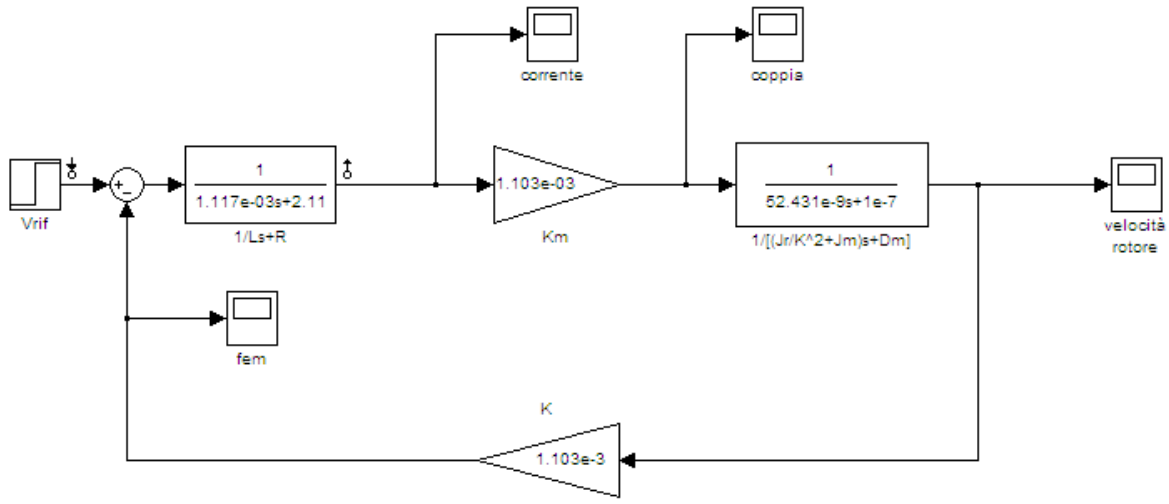


Figura 76 - schema Simulink del modello del motore DC

Simulando con *Simulink* tale sistema si ottiene il seguente andamento della velocità del rotore, espressa in *rpm*, a fronte di un gradino di tensione di *3 V* in ingresso al modello.

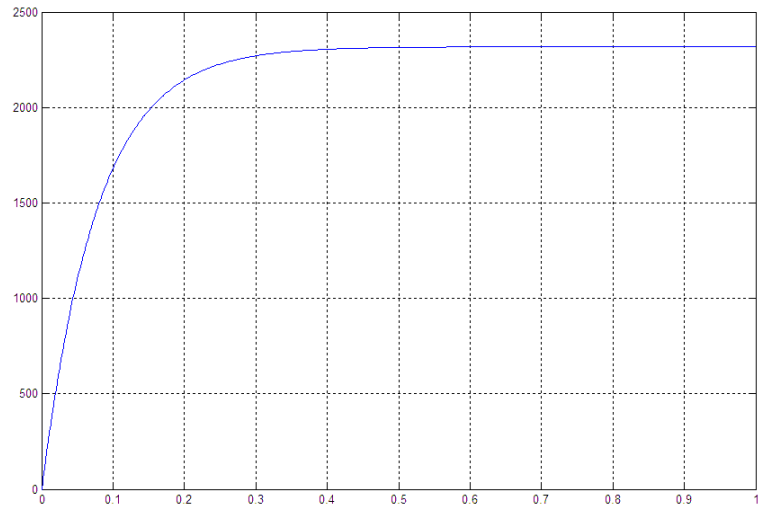


Figura 77 – Andamento della velocità del rotore a fronte di uno scalino di 3V

2.2.2 Simulazione del sistema di controllo

Lo schema di riferimento è del tipo in cascata, con un anello interno per il controllo della corrente e uno esterno per il controllo della velocità angolare, come già introdotto nella parte di teoria del controllo.

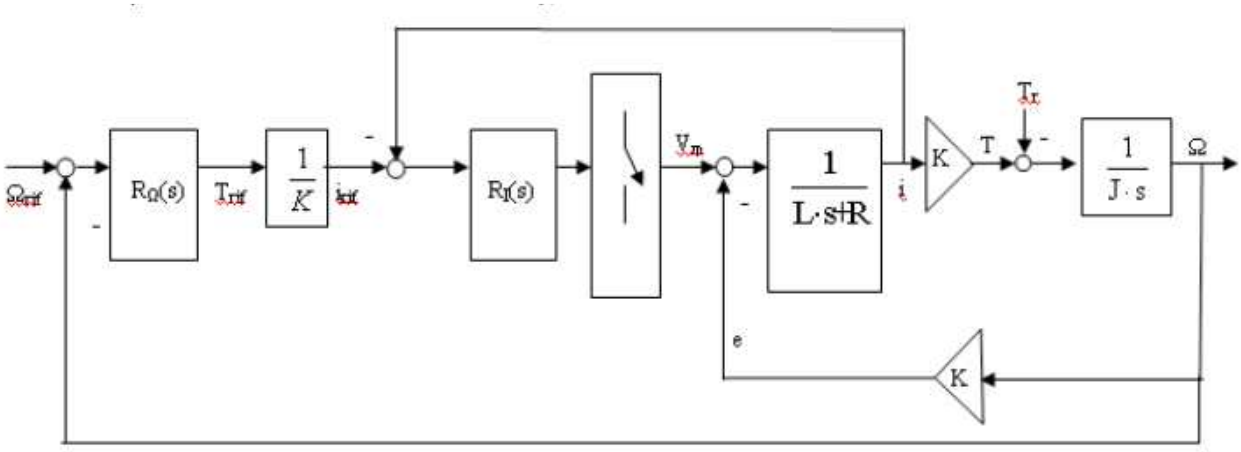


Figura 78 - Schema a blocchi del controllo senza compensazione della f.e.m.

Si è scelta direttamente la configurazione senza compensazione di *fem* in quanto la stima di tale variabile risulta spesso poco precisa e ciò comporterebbe l'introduzione di imprecisione nel

controllo. Come già introdotto basta considerare la *fem* come un disturbo addittivo e calcolare le bande passanti in maniera tale da rigettarlo.
Si può ipotizzare che la funzione di trasferimento del convertitore sia unitaria e non affetta da ritardo di conversione. Queste ipotesi sono valide in quanto la risposta del sistema meccanico è molto più lenta della risposta del sistema elettrico (ipotesi generalmente valide per tutti i servo-meccanismi).

2.2.2.1 Anello di corrente

Per il progetto dell'anello di regolazione della corrente è necessario considerare la funzione di trasferimento tra la tensione fornita al motore (variabile di controllo) e la corrente circolante nel circuito (variabile controllata):

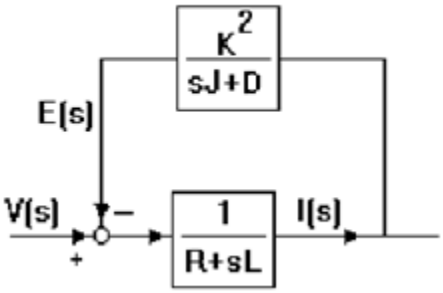


Figura 79 - Schema a blocchi del sistema per la determinazione della funzione di trasferimento I/V

Secondo l'algebra degli schemi a blocchi la funzione di trasferimento cercata è data da:

$$G_I(s) = \frac{1/(R+sL)}{1+\frac{1}{R+sL}\,\frac{K^2}{sJ_m+D_m}} = \frac{sJ_m+D_m}{s^2J_mL+s(J_mR+LD_m)+RD_m+K^2} =$$

$$= \frac{[52.431\cdot 10^{-9}s+10^{-7}]}{5.856\cdot 10^{-11}s^2+1.107\cdot 10^{-7}s+1.427\cdot 10^{-6}}$$

Tale funzione di trasferimento è caratterizzata da uno zero nell'origine e due poli reali:

z = - 1.91 rad/s
p_1 = - 13 rad/s
p_2 = - 1880 rad/s

ed è rappresentata dai *diagrammi di Bode* di modulo e fase di Figura 80.

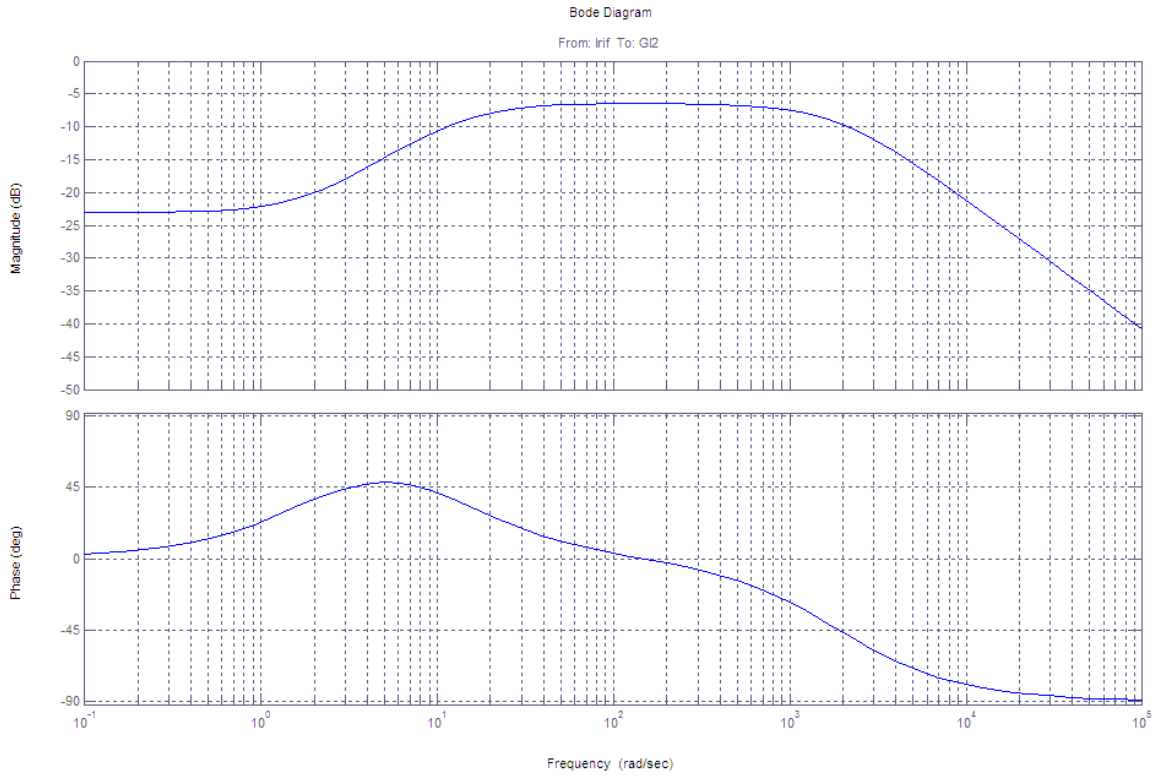


Figura 80 - Diagrammi di Bode del modulo e della fase della F.d.T. GI(s)

Lo schema dell'anello di controllo della corrente prevede l'inserimento di un *PI* opportunamente tarato per conferire al sistema complessivo una banda passante in anello chiuso di circa *1000 rad/s*. Il progetto del regolatore è stato eseguito cancellando con lo zero il polo p_2 in alta frequenza e imponendo un guadagno tale da ottenere la banda passante richiesta. Ricordando che l'espressione di un *PI* è:

$$R_I(s) = K_{PI}\left(1 + \frac{1}{sT_{iI}}\right) = \frac{K_{PI}}{sT_{iI}}(1 + sT_{iI})$$

per ottenere quanto detto è necessario che T_{il} abbia valore tale da permettere la cancellazione del polo p_2 del sistema da controllare, ovvero:

$$T_{iI} = -\frac{1}{p_2} \cong 5.3191 \cdot 10^{-4} s$$

Per determinare il guadagno proporzionale del regolatore K_{pI} si consideri il *diagramma di Bode* della funzione d'anello *LI'(s)*, assumendo per il parametro K_{pI} il valore 1. La funzione risulta pari a:

$$L_I^{'}(s) = \frac{(1 + sT_{iI})}{sT_{iI}} G_I(s) = \frac{(1 + 5.3191 \cdot 10^{-4} s)}{5.3191 \cdot 10^{-4} s} G_I(s)$$

E viene rappresentata dai *diagrammi di Bode* di modulo e fase di Figura 81.

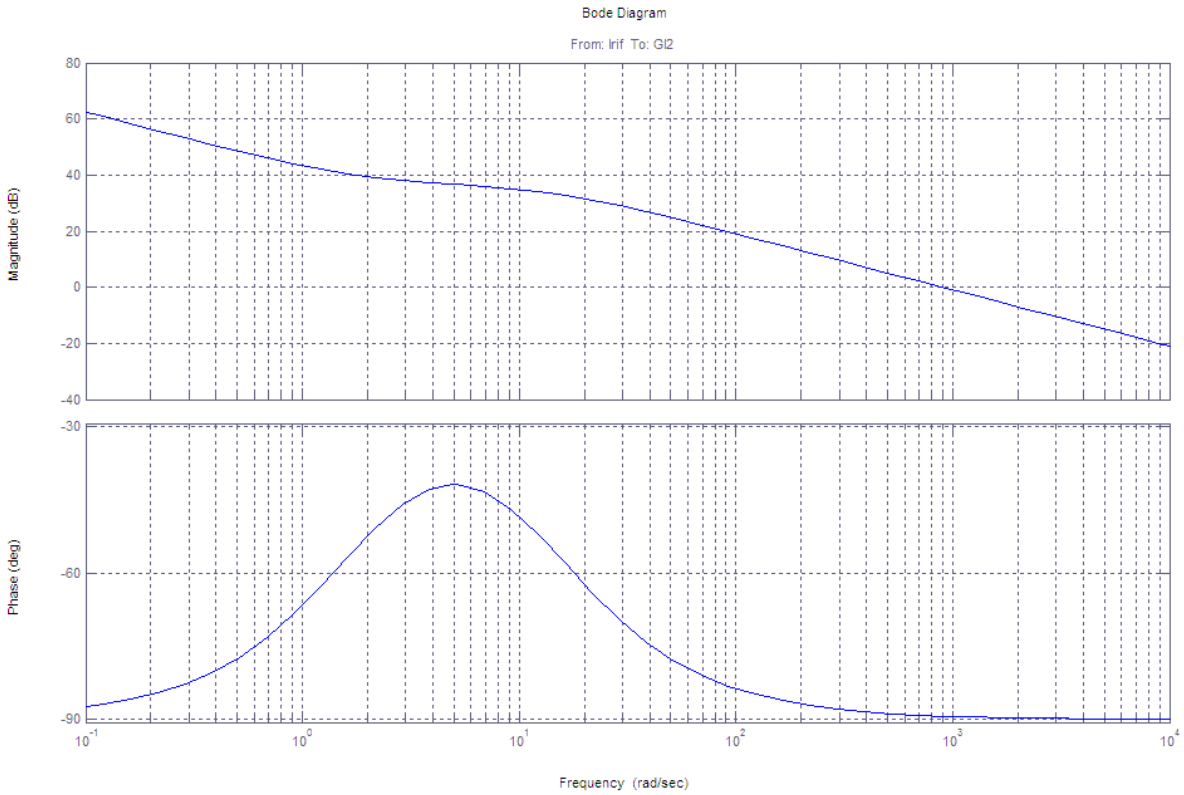


Figura 81- diagrammi di Bode della funzione d'anello aperto LI'(s)

Come si intuisce dal diagramma del modulo della **Errore. L'origine riferimento non è stata trovata.**, la unzione taglia già l'asse degli 0dB a 1000rad/s, e quindi si può utilizzare il regolatore precedentemente tarato con $K_{pI} = 1$, guadagno unitario.

Si ottengono allora le seguenti espressioni per la funzione d'anello $L_I(s)$ e la funzione di anello chiuso $F_I(s)$:

$$L_I(s) = L_I'(s) \rightarrow F_I(s) = \frac{L_I(s)}{1 + L_I(s)}$$

Lo schema a blocchi dell'anello di corrente e il *diagramma di Bode* della funzione di anello chiuso sono mostrati in Figura 82 e in Figura 83.

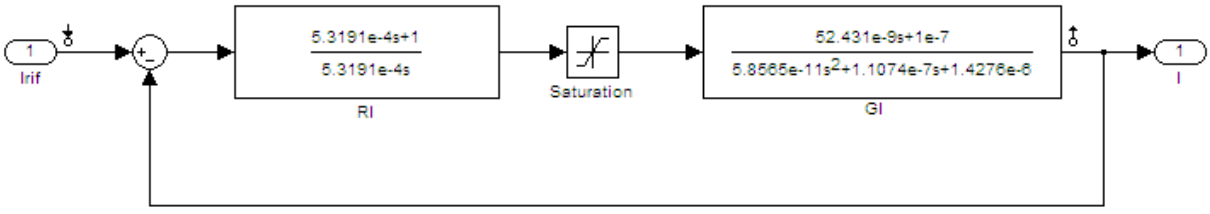


Figura 82 - schema a blocchi dell'anello di controllo della corrente

La saturazione inserita serve a rappresentare il limite dell'alimentatore che non può erogare più di 12V.

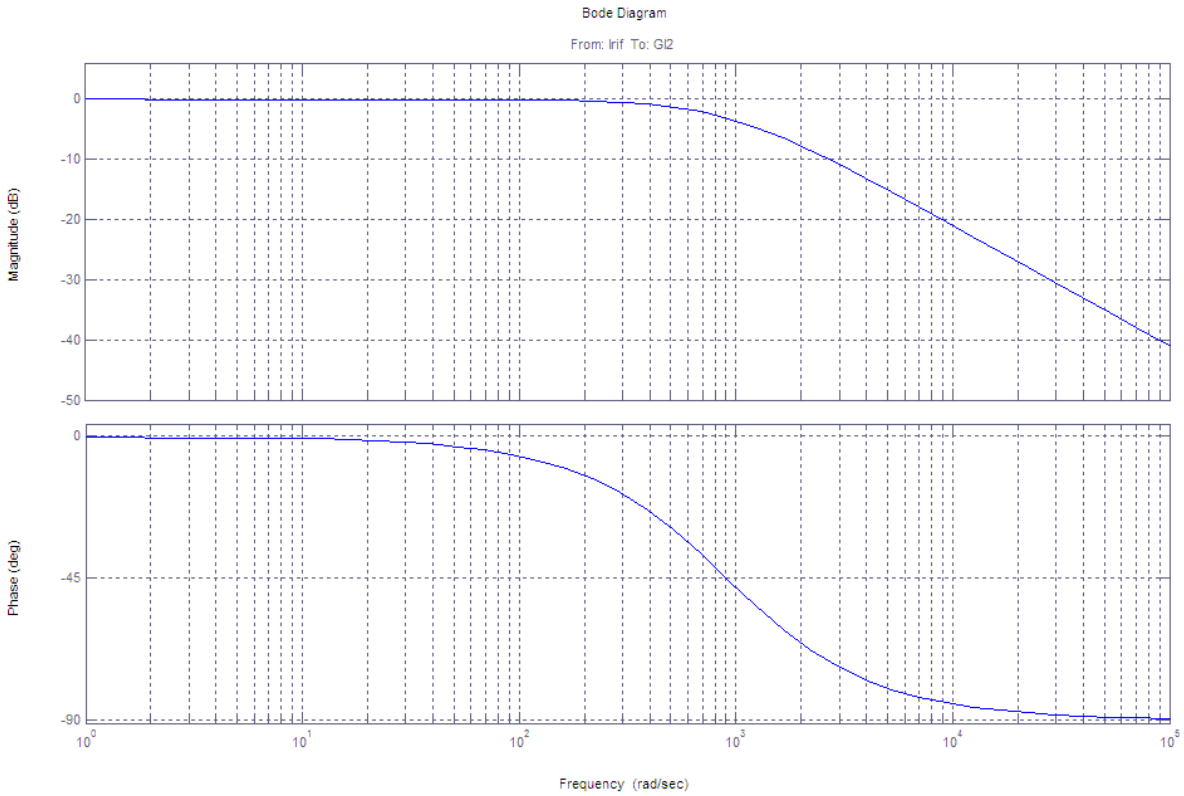


Figura 83- diagrammi di Bode di FI(s)

Per questo anello di regolazione si hanno margini di fase e guadagno pari a:

$\varphi_m \cong 135°$ $\mu_m \cong \infty$

2.2.2.2 Anello di velocità

Il progetto dell'anello di regolazione della velocità avviene in modo del tutto analogo a quello dell'anello di regolazione della corrente, con l'accorgimento di progettare una banda passante inferiore di almeno una decade rispetto a quella di quest'ultimo.
Definita $L_I(s)$ come la funzione d'anello del regolatore di corrente, trascurando per semplicità la presenza del trasduttore di corrente $M_I(s)$, è facile ricavare la funzione di trasferimento del sistema da controllare per l'anello di velocità, $G_V(s)$:

$$G_V(s) = \frac{F_I(s)}{sJ}$$

Di seguito sono riportati lo schema a blocchi del controllo di velocità (Figura 84) e i *diagrammi di Bode* del modulo e della fase (Figura 85) di $G_v(s)$.

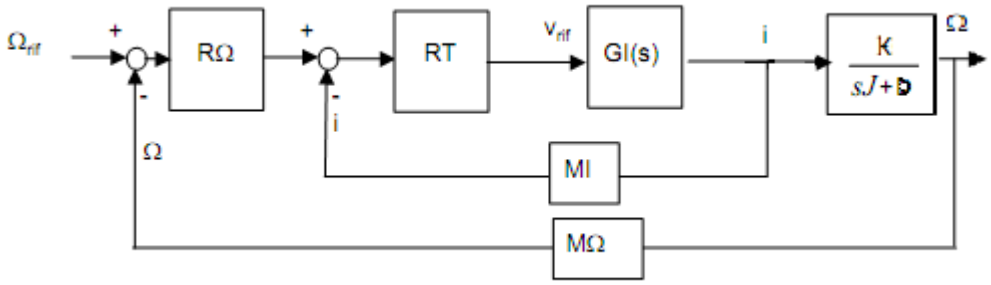


Figura 84 - schema a blocchi della struttura ad anelli annidati per la regolazione della corrente e della velocità senza compensazione della fem

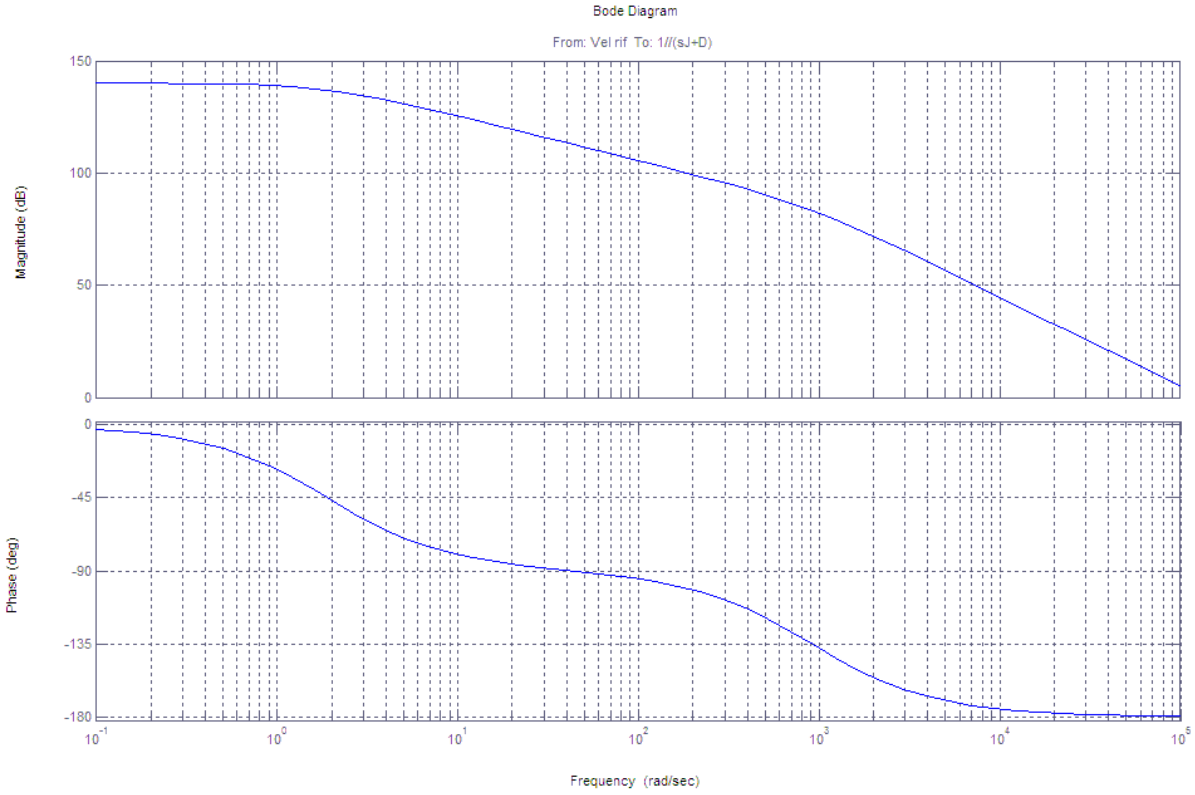


Figura 85 - diagrammi di Bode del modulo e della fase della f.d.t. GV(s)

Il sistema da controllare contiene intrinsecamente un'azione integrale che consentirebbe di progettare il regolatore di velocità come un semplice proporzionale con guadagno tale da ottenere la banda passante desiderata, pari a circa 100rad/s. Analizzando il *diagramma di Bode* del modulo di $G_V(s)$ si ricava, per ottenere la banda passante voluta, un valore di amplificazione pari a -50dB, che corrisponde a un fattore di guadagno pari a:

$$K_{pV} = 10^{(-50/20)} = 0.0032$$

Il regolatore proporzionale sarà quindi dato da:

$$R_{pV}(s) = K_{pV}$$

Lo schema a blocchi e i *diagrammi di Bode* della funzione complessiva d'anello $L_V(s)$ e della funzione d'anello chiuso $F_V(s)$ sono riportati in **Errore. L'origine riferimento non è stata trovata.** e in Figura 88:

$$L_V(s) = R_{pV}(s) G_V(s) \Rightarrow F_V(s) = \frac{L_V(s)}{1 + L_V(s)}$$

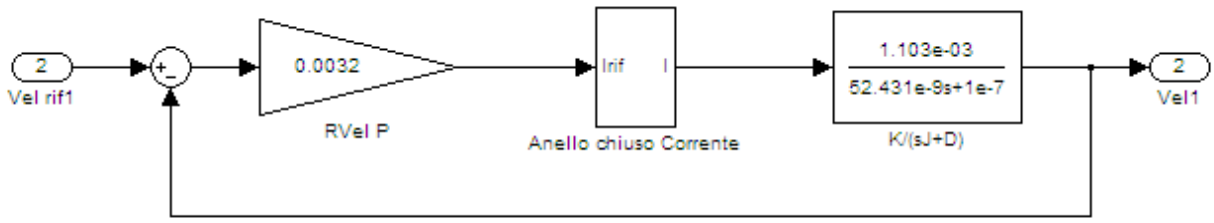


Figura 86 - schema a blocchi dell'anello di controllo della velocità

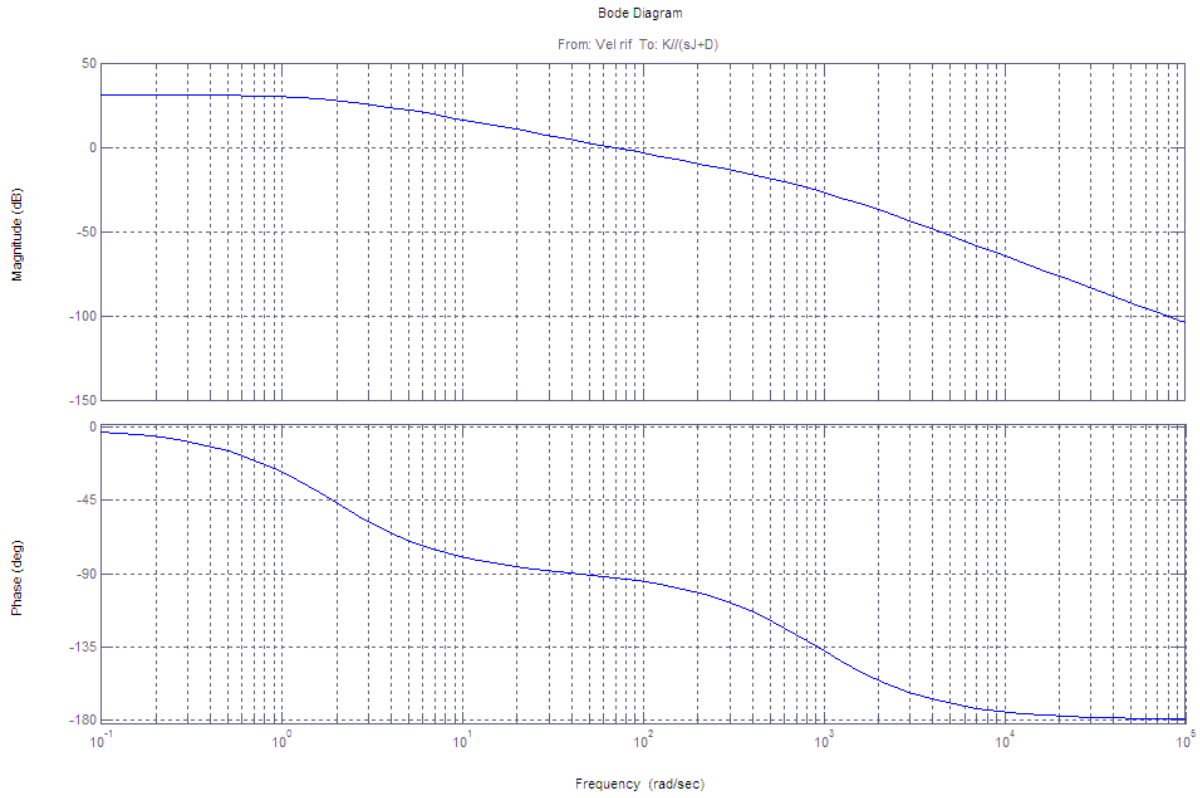


Figura 87 - diagrammi di Bode di LV(s)

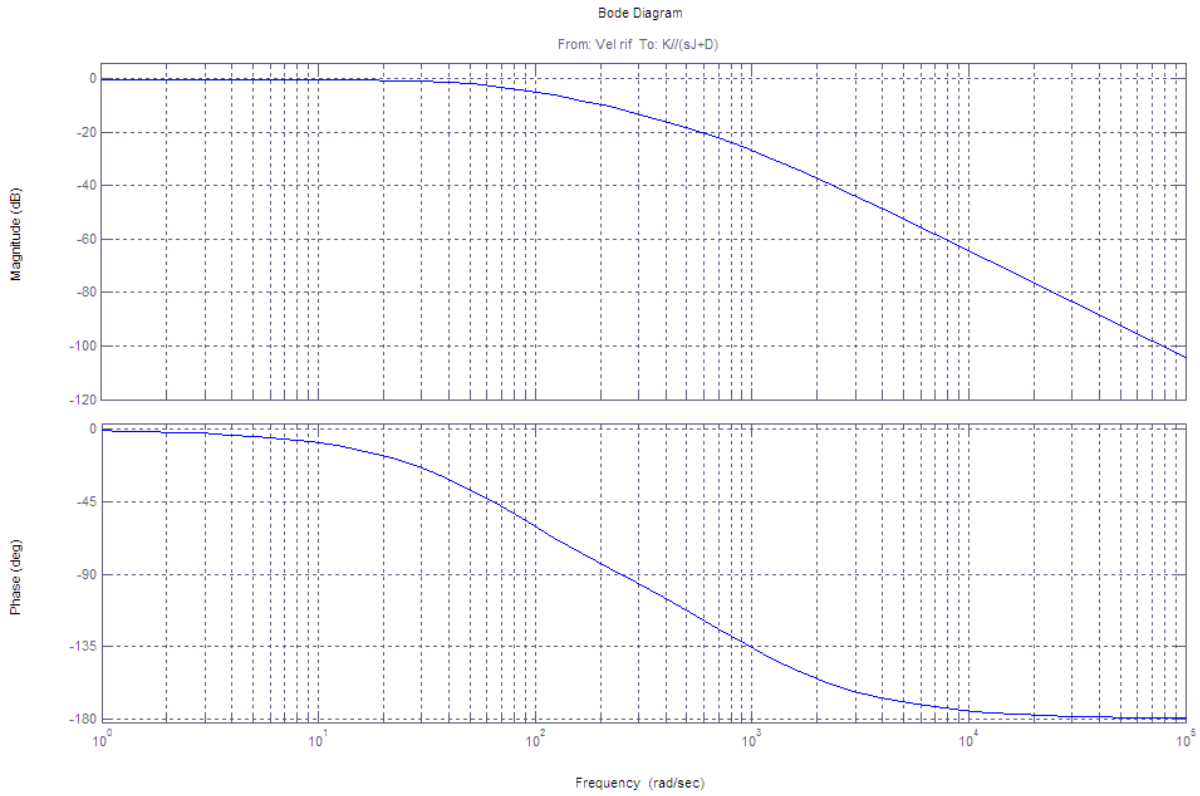


Figura 88 - diagrammi di Bode di FV(s)

I margini di fase e guadagno per l'anello di controllo così tarato risultano:

$\varphi_m \cong 90°$ $\mu_m = \infty$

Volendo progettare un regolatore di tipo PI anche per il controllo dell'anello di corrente, è necessario posizionare lo zero in modo da ottenere un adeguato recupero di fase nell'intorno della banda passante desiderata, ovvero almeno una decade prima della banda passante d'anello.

$T_{iV} = 0.03s$

L'introduzione dell'integratore e dello zero non modifica il guadagno complessivo dell'anello nell'intorno della banda passante richiesta, per modo che il parametro K_{pV} mantiene il valore calcolato in precedenza. Lo schema a blocchi e i *diagrammi di Bode* della funzione d'anello aperto $L_V(s)$ e di anello chiuso $F_V(s)$ sono riportati in **Errore. L'origine riferimento non è stata trovata.**, Figura.90 e Figura.91.
Per questa taratura del regolatore si ottengono margini di fase e guadagno pari a:

$$\varphi_m \cong 80° \qquad \mu_m \cong \infty$$

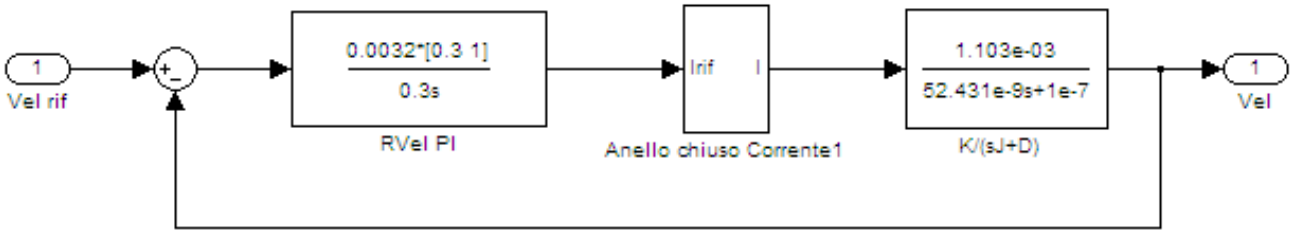


Figura 89 - schema a blocchi dell'anello di controllo della velocità

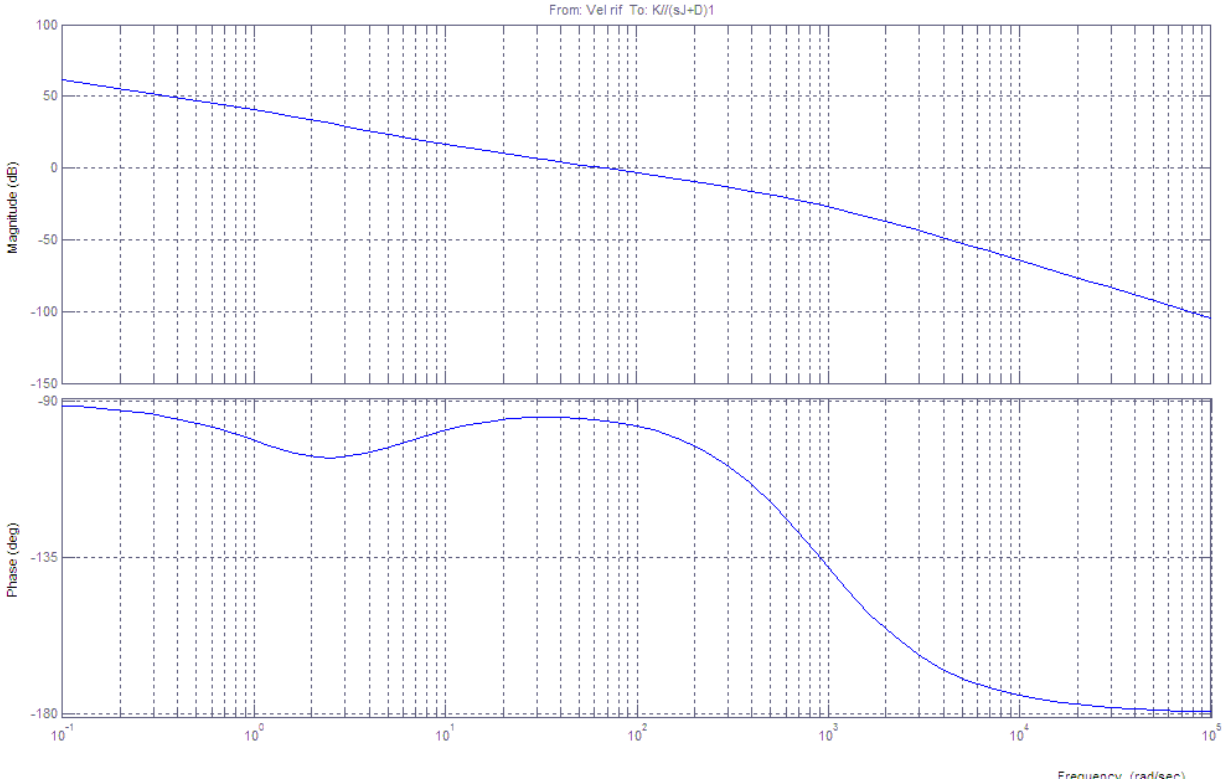


Figura 90 - diagrammi di Bode di LV(s)

Ovviamente mentre per quanto riguarda l'anello di corrente si presuppone che ci sarà una certa convergenza fra i valori di *Kp* e *Ki* trovati tramite analisi *Simulink* e quelli che verranno poi utilizzati nel microprocessore, ciò appare meno vero per quanto riguarda i parametri di controllo dell'anello di velocità. Infatti in *Simulink* si è supposto di retroazionare direttamente la velocità del motore, ma nel progetto reale questa viene stimata, in modalità diverse. Ognuna di queste modalità però comporta un accumularsi di imperfezioni di misura e di errori di digitalizzazione impossibili da rendere con un programma di simulazione, cosicchè molto difficilmente i valori così tarati corrisponderanno a quelli realmente implementati.

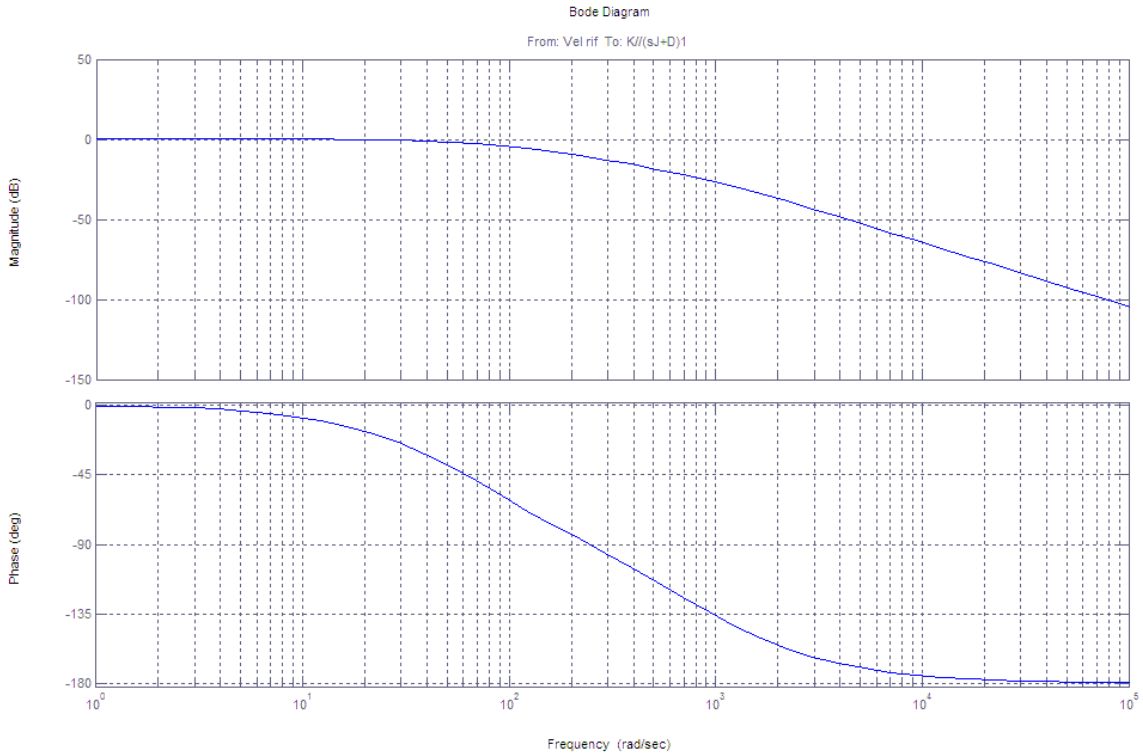


Figura 91 - diagrammi di Bode di FV(s)

2.2.3 Discretizzazione dei regolatori

2.2.3.1 Aspetti teorici

Per discretizzare i regolatori a tempo continuo si sono seguiti questi passi:

1) scelta del tempo di campionamento;
2) scelta del metodo di discretizzazione.

Scelta del tempo di campionamento

a. <u>Regolatore di corrente</u>
La banda passante scelta per l'anello di corrente è pari a:

$$\omega_{CI} = 1000\ rad/s$$

Per scegliere il tempo di campionamento bisogna considerare il *teorema di Shannon*. Si ha quindi:

$$\omega_{SI} \geq 2 \cdot \omega_{CI} \ \rightarrow \ \frac{2\pi}{T_{SI}} \geq 2 \cdot \omega_{CI} \ \rightarrow \ T_{SI} \leq \frac{2\pi}{2\omega_{CI}} = \frac{\pi}{1000} = 3.14 \cdot 10^{-3}$$

Tenendo conto che è sempre meglio andare oltre ai requisiti minimi del teorema, si sceglie un tempo di campionamento per l'anello di corrente pari a:

$$T_{SI} = 0.1\ ms$$

b. Regolatore di velocità

La banda passante scelta per l'anello di velocità è pari a:

$$\omega_{CV} = 100\ rad/s$$

Per scegliere il tempo di campionamento si ripetono le considerazioni e i calcoli precedentemente usati per l'anello di corrente, ottenendo tale valore:

$$T_{SV} = 1\ ms$$

In realtà i tempi di campionamento sono imposti dal progetto ma questa verifica ha consentito comunque di giudicare la validità delle bande passanti progettate per i regolatori di velocità e corrente.

Scelta del metodo di discretizzazione

Esistono diversi metodi per potere discretizzare una funzione a tempo continuo, di seguito se ne analizzeranno essenzialmente tre:

a. Eulero in avanti

La sostituzione alla base della discretizzazione tramite il metodo di *Eulero in avanti* è la seguente:

$$s = \frac{z-1}{T_s}$$

Si tratta di un metodo molto semplice e computazionalmente 'leggero', ma è possibile ottenere regolatori discretizzati instabili a partire da regolatori a tempo continui stabili.

b. Eulero all'indietro

In questo caso si procede con la sostituzione:

$$s = \frac{z-1}{z \cdot T_s}$$

Con questo metodo tutti i controllori a tempo continuo stabili vengono trasformati in controllori a tempo discreto stabili; esiste pure la possibilità che controllori a tempo continuo instabili vengano trasformati in controllori a tempo discreto stabili.

c. Tustin

Si procede con la seguente sostituzione:

$$s = \frac{2 \cdot (z-1)}{T_s \cdot (z+1)}$$

Con questo metodo tutti i controllori a tempo continuo stabili mantengono la stabilità anche quando discretizzati, ma i poli ad alta frequenza del controllore a tempo discreto tendono a -1, come si può notare osservando la seguente espressione:

$$z = \frac{1 + s \cdot \frac{T_s}{2}}{1 + s \cdot \frac{T_s}{2}}$$

Quindi il segnale di controllo discreto potrebbe presentare delle oscillazioni (effetto di poli con modulo minore di uno ma negativi).

In quanto comunemente utilizzato negli ambienti di simulazione, per discretizzare i regolatori del progetto si è scelto di utilizzare il metodo di *Tustin*.

2.2.3.2 Discretizzazione

Il regolatore di corrente discretizzato assume quindi la seguente forma:

$$R_I(s) \rightarrow \left(s = \frac{2}{0.1 \cdot 10^{-3}} \frac{z-1}{z+1}\right) \rightarrow \; R_I(z) = \frac{1.094z - 0.906}{z-1}$$

Nel caso del regolatore di velocità, allorquando non si usi quello puramente proporzionale, che quindi rimane uguale a quello continuo, si ha la seguente trasformazione:

$$R_V(s) \rightarrow \left(s = \frac{2}{1 \cdot 10^{-3}} \frac{z-1}{z+1}\right) \rightarrow \; R_V(z) = \frac{0.0032z - 0.0032}{z-1}$$

2.2.4 Simulazione del comportamento del sistema

Simulando il comportamento del sistema a fronte di uno scalino del riferimento di velocità di *2500 rpm* si ottengono i seguenti risultati, in termini di velocità in uscita e andamento della tensione sul motore. Per i primi due grafici si è utilizzato un controllo avente il regolatore di corrente discreto e come regolatore di velocità quello puramente proporzionale. Nel secondo caso entrambi i regolatori sono quelli discreti.

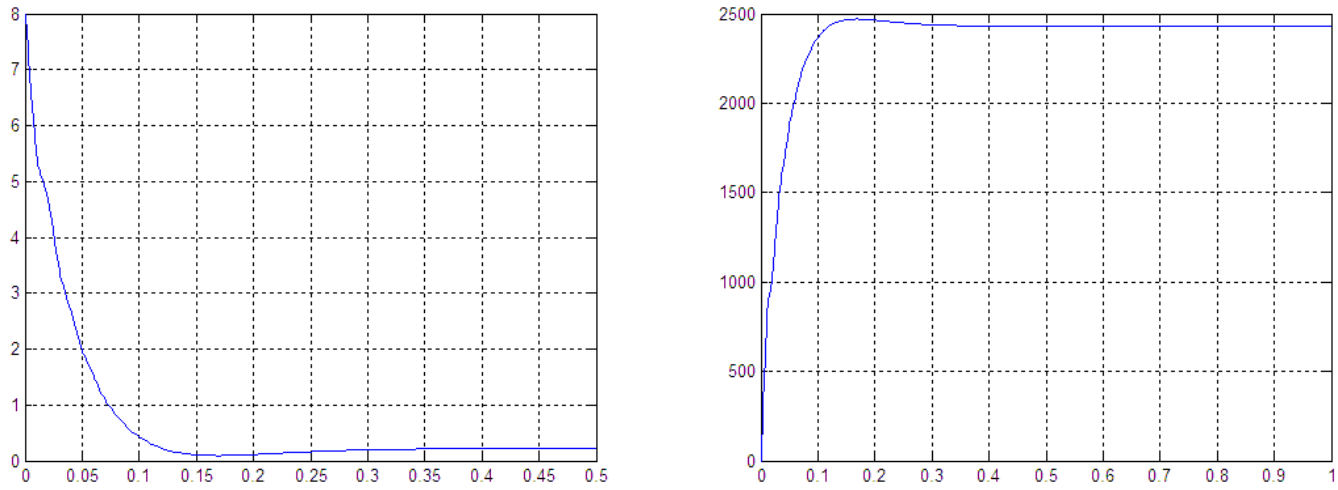


Figura 92 – andamento della tensione sul motore e della velocità del rotore con P-PI continui a fronte di uno scalino sul riferimento di velocità

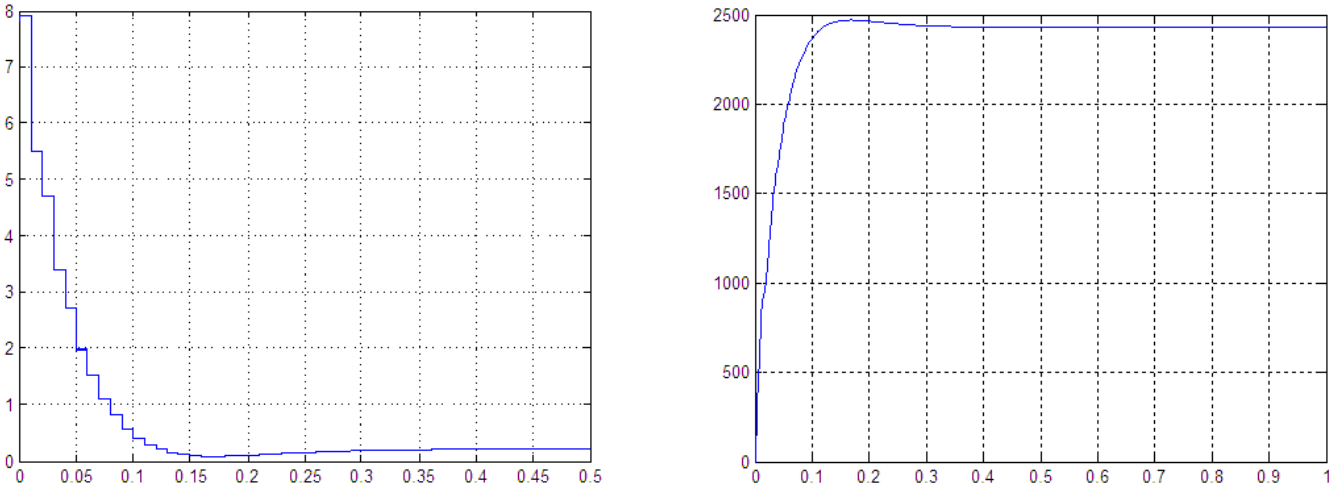


Figura 93 - andamento della tensione sul motore e della velocità del rotore con PI-PI discreti a fronte di uno scalino sul riferimento di velocità

Come si può notare non si nota una grande differenza in termini di risposta della velocità del rotore (si potrebbe dire che l'uso di un integratore nell'anello esterno è praticamente inutile) mentre è interessante notare l'effetto dell'uso del regolatore discreto di velocità sulla tensione, che ovviamente assume un andamento '*a gradini*'.

Compiute queste simulazioni ci si è concentrati sulla digitalizzazione del controllo, ovvero il lavoro pratico sul codice del microprocessore, tralasciando di effettuare simulazioni *Simulink* sul sistema digitalizzato, ma accontentandosi dei risultati e delle informazioni fornite dall'analisi continua e discreta del sistema.

2.3 Sviluppo comunicazione

Una volta effettuate le misure e le prove sul motore, e avere analizzato il sistema con le simulazioni *Matlab*, si è entrati nella parte pratica del progetto, ovvero l'implementazione sulle schede della comunicazione e del controllo.
Le due schede a disposizione – la scheda *zigkey* avente funzione di trasmettitore, collegata tramite *USB* al computer, la scheda *zigkey_trenino* collegata al motore – avevano già impostato un codice predisposto alla comunicazione. Di conseguenza la prima parte del lavoro è stata analizzare tale codice, testarlo, e nel caso modificarlo per i nostri scopi.
L'obbiettivo dello sviluppo della comunicazione era trasferire dei pacchetti dati da un generico pc alla scheda *zigkey_trenino*, secondo questa catena di passaggi:

- Definire il formato dei pacchetti che verranno inviati dal pc alla scheda *zigkey_trenino*;
- Creazione interfaccia *VB* per inviare dati attraverso una porta seriale (o una porta *USB* virtualmente resa seriale) alla chiavetta *zigkey* connessa al pc;
- Testare ed eventualmente modificare il codice implementato in tale chiavetta che gestisce la comunicazione tra l'ingresso *USB* e la memoria del Micro. La chiavetta utilizza per la comunicazione tramite la sua porta *USB* lo *UART (*attraverso un transcreiver della *FTDI)*;
- Testare ed eventualmente modificare il codice implementato per inviare tramite wireless i dati dalla chiavetta *zigkey* alla *zigkey_trenino*. In questo caso il codice si basa sul protocollo *MiWi P2P*;
- Testare il funzionamento dell'intera catena mandando un messaggio dall'interfaccia *VB* fino alla scheda *zigkey_trenino*.

2.3.1 Scelta formato pacchetti

La scelta del formato dei pacchetti da spedire tramite interfaccia *Visual Basic* al *PIC32* destinato alla gestione del controllo del trenino elettrico è stata fatta prendendo spunto dal sito *DCCWorld*. Questo sito si propone di diffondere il sistema di controllo digitale *DCC* (*Digital Command Control*) per il modellismo ferroviario. Oltre a mettere a disposizione informazioni utili per introdurre i neofiti al mondo del digitale, spiegare nel dettaglio come realizzare un controllo e disquisire sui vari tipi di componenti utilizzabili, questa comunità aderisce agli standard *DCC* proposti dall'associazione *NMRA* (*National Model Railroad Association*), condividendone gli obiettivi.
Lo standard *DCC Communications Standard S-9.2* del Luglio 2004 prevede un pacchetto base di 3 Byte, annunciato da un preambolo di dodici bit 1, più un bit 0 di separazione; il pacchetto base deve terminare con un bit 1 posto dopo il terzo byte. Dei 3 byte del pacchetto base il primo byte contiene l'indirizzo del *decoder digitale*, ovvero del trenino che si vuole controllare in quel

momento, il secondo byte contiene il *payload* e il terzo byte è usato per verificare eventuali errori di trasmissione.

Avendo noi scelto di utilizzare lo standard *MiWi* ed avendo un solo trenino elettrico da controllare, prenderemo spunto solo dal secondo byte del pacchetto base, quello del *payload*, e realizzeremo il nostro formato basandoci sulle funzionalità che vogliamo implementare nel nostro progetto. Tutta la parte di bit dedicata al controllo, invio/ricezione, etc… viene gestita direttamente dal protocollo *MiWi* e dal codice pre-implementato, che hanno il compito di incapsulare il *payload* da noi creato e spedirlo secondo le loro specifiche.

Per prima cosa sono state elencate le funzioni a cui deve rispondere il trenino, e per ognuna delle quali deve esistere un determinato pacchetto che la rappresenti. Queste sono:

4. 16 livelli di velocità per ogni direzione;
5. 8 livelli di sfumatura per luci ad intensità variabile;
6. luci anteriori on/off;
7. luci posteriori on/off;
8. accensione/spegnimento del sistema;
9. segnale sonoro tramite buzzer.

A questo punto sono state definite tre diverse tipologie di pacchetto, ciascuna di un byte:

4. un pacchetto per la direzione e la velocità;
5. un pacchetto per le luci sfumate;
6. un pacchetto per le restanti funzionalità sopra elencate.

Il pacchetto (1) assume la forma:

Bits	1	2	1	4
	0	10	D	SSSS

Figura 94 – pacchetto direzione e velocità

bit 8 sempre zero;
bit 7-5 identificativi del ruolo del pacchetto: in questo caso 10=‘pacchetto velocità’;
bit 4 (D) indica la direzione desiderata di movimento: D=1 avanti, D=0 indietro;
bit 3-0 (SSSS) rappresentano i 16 livelli di velocità a disposizione: SSSS=1111 massima velocità, SSSS=0001 minima velocità.

Il pacchetto (2) è definito invece in questa maniera:

Bits	1	2	2	3
	0	11	YY	III

Figura 95 – pacchetto luci sfumate

bit 8 sempre zero;
bit 7-5 identificativi del ruolo del pacchetto: in questo caso 11=‘pacchetto luci sfumate’;
bit 4-3 (YY) non danno alcuna informazione utile: YY=00;
bit 2-0 (III) rappresentano gli 8 livelli di sfumatura che possono assumere le luci ad intensità variabile: III=111 massima intensità, III=001 minima intensità.

Il pacchetto (3) infine è così definito:

Bits	1	2	3	1	1
	0	01	ZZZ	A	P

Figura 96 – pacchetto luci-buzzer-on/off

bit 8 sempre zero;
bit 7-5 identificativi del ruolo del pacchetto: in questo caso 01=‘pacchetto luci-buzzer-on/off’;
bit 4-2 (ZZZ) identificano 3 sottotipologie di pacchetto;
bit 1-0 (TT) cambiano significato a seconda della sottotipologia indicata.

Le tre sottotipologie di pacchetto sono:

- ZZZ=001: ‘pacchetto luci anteriori e posteriori’.
  bit 1 (A) indica lo stato della luce anteriore: A=1=ON, A=0=OFF.
  bit 0 (P) indica lo stato della luce posteriore: P=1=ON, P=0=OFF.
- ZZZ=010: ‘pacchetto accensione/spegnimento’.
  bit 1-0 (AP) non contengono informazione utile: AP=00. Il pacchetto infatti prevede l'inversione dello stato del sistema e non ha quindi bisogno di informazioni dedicate per accensione e spegnimento.
- ZZZ=000: ‘pacchetto buzzer’.
  bit 1-0 (AP) non contengono informazione utile: AP=000. Il pacchetto infatti innesca un azione autorientrante: quando viene ricevuto il buzzer emette un segnale sonoro per un intervallo di tempo finito.

Quando la scheda *ZigKey_trenino*, destinata al controllo del trenino, riceve un pacchetto dati dalla sua periferica *SPI* utilizza una serie di maschere per isolare i bit di interesse e quindi svolgere le opportune azioni di controllo.
A differenza però del *PIC32* della scheda *ZigKey_trenino*, che, come appena spiegato, legge il pacchetto dati come sequenza di bit, tutte le componenti a monte di esso, ovvero il *PIC32* della scheda *ZigKey*, i due moduli *MRF* e l'interfaccia *VB* da cui partono i dati, trattano i bit dei pacchetti dati come un singolo carattere di 8 bit, un byte, sfruttando il loro equivalente nelle tabelle *ASCII*.
Grazie a questa scelta, la nostra interfaccia *VB* è stata progettata con l'obiettivo di trasmettere opportuni caratteri lungo tutta la catena *hardware* fino ad arrivare al *PIC32* che comanda il trenino, dove i caratteri vengono finalmente 'aperti' e letti come semplice sequenza di bit.
Questo rende tutta la trasmissione più semplice e chiara soprattutto a livello di controllo utente sulle varie parti di codice.
Riportiamo quindi nelle Tabelle 6, 7 e 8 collocate nelle prossime pagine la corrispondenza binario-*ASCII* che abbiamo utilizzato per i nostri pacchetti.

Pacchetto velocità:

A	0100 0001	> AVANTI, velocità minima
B	0100 0010	
C	0100 0011	
D	0100 0100	
E	0100 0101	
F	0100 0110	
G	0100 0111	
H	0100 1000	
I	0100 1001	
J	0100 1010	
K	0100 1011	
L	0100 1100	
M	0100 1101	
N	0100 1110	
O	0100 1111	> AVANTI, velocità massima
P	0101 0000	> VELOCITA' NULLA
Q	0101 0001	> INDIETRO, velocità minima
R	0101 0010	
S	0101 0011	
T	0101 0100	
U	0101 0101	
V	0101 0110	
W	0101 0111	
X	0101 1000	
Y	0101 1001	
Z	0101 1010	
[	0101 1011	
\	0101 1100	
]	0101 1101	
^	0101 1110	
_	0101 1111	> INDIETRO, velocità massima

Tabella 6 – corrispondenza fra caratteri ASCII inviati per i comandi della velocità e corrispondente valore in bit

Pacchetto luci sfumate:

A minuscolo	0110 0001	> Intensità minima
B minuscolo	0110 0010	
C minuscolo	0110 0011	
D minuscolo	0110 0100	
E minuscolo	0110 0101	
F minuscolo	0110 0110	
G minuscolo	0110 0111	> Intensità massima

Tabella 7 – corrispondenza fra caratteri ASCII inviati per i comandi delle luci sfumati e corrispondente valore in bit

Pacchetto luci-buzzer-on/off:

#	0010 0011	> Buzzer
S	0010 0100	> LUCI off
%	0010 0101	> LUCI posteriori on
&	0010 0110	> LUCI anteriori on
'	0010 0111	> LUCI on
(	0010 1000	> TRENO on/off

Tabella 8 – corrispondenza fra caratteri ASCII inviati per i comandi di buzzer, luci e on/off e corrispondente valore in bit

2.3.2 Creazione dell'interfaccia grafica con Visual Basic

Una volta deciso il formato del payload da inviare l'operazione successiva è stata creare un'interfaccia tramite *Visual Basic*. Scopo di tale interfaccia è inviare i caratteri precedentemente individuati dal computer alla chiave *ZigKey*, tramite porta *USB*.
Preliminarmente è stato necessario trasformare le porte *USB* del pc in porte seriali, utilizzando un opportuno programma che permette di '*vedere*' – agli occhi del computer - tali uscite come uscite seriali, numerandole come *COM1, COM2, …* . Si è dovuto procedere in questo modo in quanto il protocollo *UART* utilizzato dalla chiavetta *ZigKey* per comunicare col pc prevede di utilizzare porte seriali, o al più, come in questo caso, porte *USB* virtualmente trasformate in seriali.
Questa necessità è comunque tornata molto utile dal punto di vista dell'interfaccia *VB* in quanto *Visual Basic 2010* fornisce già di suo un controllo, *SerialPort*, che implementa nel form in costruzione tutti i comandi necessari per dialogare tramite porta seriale.
Una volta aggiunto tale controllo si è quindi costruito un codice molto semplice, sulla stregua di quelli che vengono forniti nei vari tutorial come esempio di comunicazione seriale, che permettesse di scegliere la porta seriale utilizzata, impostare il *baud rate*[7] indicato di *default* nella

[7] **Baud rate** (detto anche **symbol rate**): indica il numero di simboli trasmessi al secondo in un sistema di trasmissione digitale.

chiavetta *ZigKey*, ricevere e visualizzare caratteri. Una volta ottenuto questo bisognava solo aggiungere – sul *form* e nel codice - i comandi necessari a inviare i caratteri precedentemente trovati, associando a ogni pulsante e barra l'invio di uno di essi. Alla fine si è ottenuta questa interfaccia:

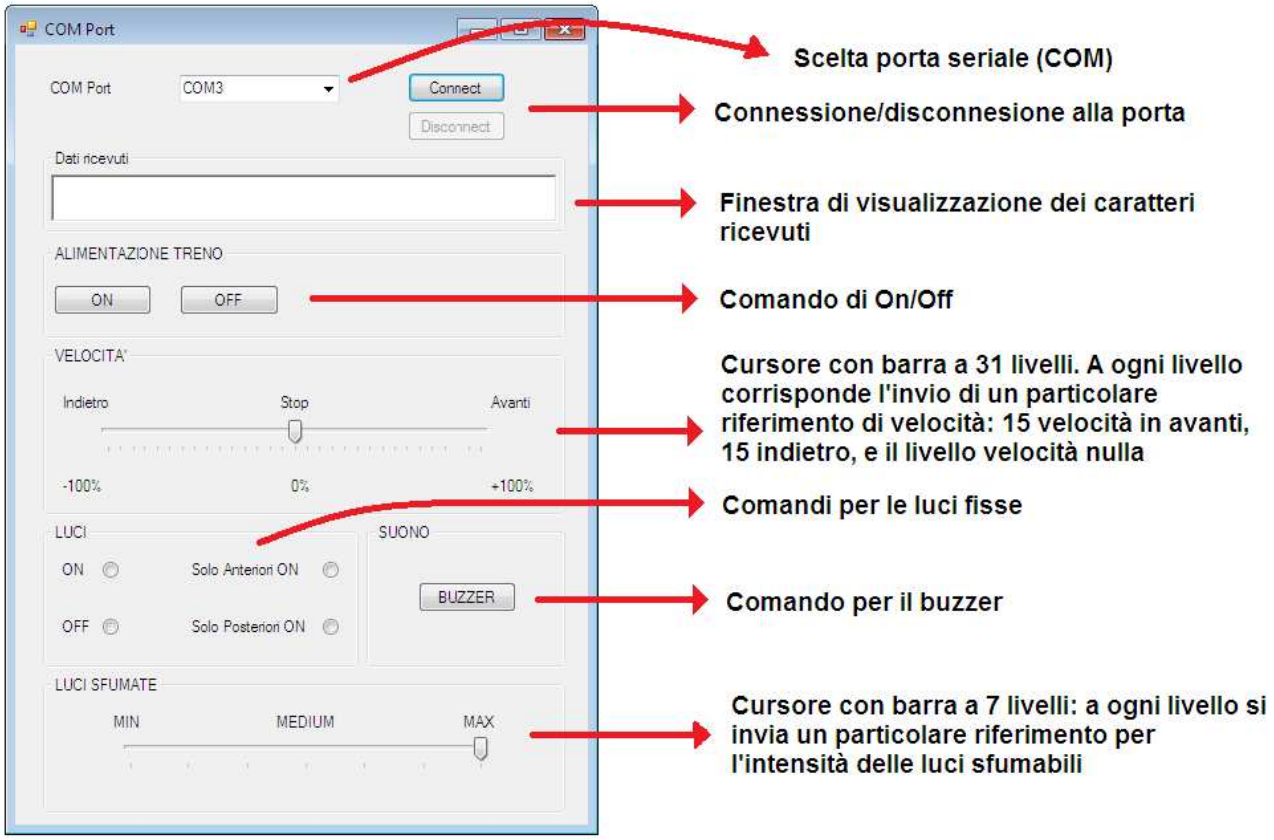


Figura 97 – aspetto grafico dell'interfaccia VB così come si mostra al generico utente

All'atto pratico, in realtà, i comandi *on/off* e *buzzer*, seppur implementati in *VB*, nella comunicazione e nel *PIC32* della scheda *ZigKey_trenino*, non sono attuabili a causa dell'hardware a nostra disposizione.

2.3.3 Implementazione e modifiche al codice pre-impostato per la comunicazione tramite UART fra Pc e scheda Zigkey

Creata l'interfaccia *VB* è stato necessario impostare la comunicazione con la chiavetta *ZigKey*, ovvero fare in modo che, a fronte dell'invio di alcuni caratteri di prova, il *PIC32* della chiavetta reagisse azionando i *led* con una certa logica, in maniera tale da provare l'effettiva bontà della connessione e dell'interfaccia stessa.

La scheda è stata fornita con già implementato un codice per la comunicazione seriale tramite protocollo *UART*. All'interno di tale codice si è trovata un'*interrupt service routine* che gestisce l'interrupt di ricezione, che scatta allorquando, dall'esterno, venga inviato un pacchetto tramite porta seriale. Tale *isr* contiene al suo interno una macchina a stati che permette di gestire il numero di caratteri inviati attraverso una logica *switch-case*.
Per prima cosa va specificato che la scheda ha a sua disposizione un *buffe*r, definito *RTDMRxBuffer*, a 64 livelli in grado di contenere parole a 8 bit.
La macchina a stati ha il seguente funzionamento:

- STATUS_IDLE: se si invia un '?' il PIC risponde con '0' se non ci sono dati pronti all'invio, con '1' se ci sono. Se si invia un '*' il PIC imposta lo status su STATUS_CMD, la posizione del buffer a zero (posBuffer=0) ed esce dallo switch-case. Se invio un carattere diverso non si ha effetto;
- STATUS_CMD: il carattere inviato viene salvato in RTDMRxBuffer[posBuffer++]. Lo status è aggiornato a STATUS_LEN dopodichè avviene il break;
- STATUS_LEN: se il carattere inviato è diverso da zero, allora viene salvato in RTDMRxBuffer[posBuffer++] e la variabile lenBuffer assume il medesimo valore. Lo status viene posto su STATUS_DATI e avviene il break; se invece il carattere è uguale a zero allora, essendo la lunghezza del buffer richiesta pari appunto a zero, si alza il flag di messaggio ricevuto e si imposta lo stato su STATUS_IDLE. Dopodichè avviene il break;
- STATUS_DATI: il carattere inviato viene salvato in RTDMRxBuffer[posBuffer++], se (posBuffer>=lenBuffer+2) allora viene alzato il flag di messaggio ricevuto, viceversa si continua a rimanere all'interno di questo stato.

Questo *switch-case*, all'apparenza molto complicato, fornisce molte possibilità. Infatti è possibile scegliere la lunghezza del *buffer*, ovvero quante sue celle riempire di dati prima di considerare il messaggio ricevuto. Per esempio, se voglio inviare la scritta 'ciao' dovrò far precedere a questa l'invio del numero 4 a significare che finchè non arriverò a riempire la quarta cella, quella destinata alla 'o', il messaggio non sarà considerato raggiunto, e il *flag* relativo non sarà alzato (l'elevamento del *flag* induce nel *PIC* l'azione di leggere dentro il buffer e utilizzare quei caratteri, cosa che con flag=0 non viene svolta).
Tutto questo però è parecchio ridondante per i bisogni del progetto. Infatti noi dobbiamo inviare un unico carattere, che poi il *PIC* leggerà dal buffer e ritrasmetterà tramite wireless. In quest'ottica quindi indicare la lunghezza del *buffer* è superfluo, basterebbe che a ogni '*giro*' di codice il carattere inviato riempia sempre *RTDMRxBuffer[0]*, il *PIC* lo legga, lo svuoti e si prepari per l'arrivo di un nuovo dato.
A fronte di ciò il codice è stato modificato, soprattutto nello STATUS_DATI, che è diventato così:

- STATUS_DATI: il carattere inviato viene salvato in RTDMRxBuffer[posBuffer++], se (posBuffer>=lenBuffer+2) allora viene alzato il flag di messaggio ricevuto, viceversa lo stato viene posto a STATUS_IDLE.

In altre parole si è fornita un'alternativa all'*if* che permette di non continuare a rimanere nello STATUS_DATI finchè la condizione dell'*if* non sia stata soddisfatta. Successivamente *VB* è stato modificato per inviare, a ogni comando, la sequenza: *X, dove l'asterisco è il '*lasciapassare*' per uscire dallo STATUS_IDLE, e 'X' il generico carattere associato al comando. In questa maniera la condizione dell'*if* non viene mai soddisfatta, perché non viene mai inviato un numero che rappresenti la lunghezza voluta del *buffer*, e quindi ogni volta che la prima cella del *buffer* si riempie con 'X' subito il dispositivo ritorna allo STATUS_IDLE pronto a ricevere un nuovo *X che sovrascriva quello vecchio.
Il metodo può essere un po' grezzo, o meglio non sfruttare appieno le potenzialità del codice, anzi piegarlo in '*malo modo*' al nostro scopo, ma è risultato molto affidabile e più semplice da gestire nella doppia trasmissione *UART-MiWi*.
L'ultima questione da risolvere era quella del flag, ovvero di '*avvisare*' il *PIC* di leggere e utilizzare il carattere salvato nella cella del *buffer*. Per far questo invece del metodo precedentemente impostato si è anche qui optato per una scelta molto più semplice: si è modificato *VB* per inviare, per ogni comando, la sequenza *aX.
Fermo restando il significato dell'asterisco e della 'X', 'a' è una sorta di controllo, o avviso. In pratica ogni volta che *VB* trasmette la sequenza *aX la macchina a stati modificata salva 'a' in *RTDMRxBuffer[0]* e 'X' in *RTDMRxBuffer[1]*, dopodiché torna allo STATUS_IDLE. Dall'altra parte il *PIC* a ogni giro del codice si trova di fronte a un *if* così strutturato:

if (RTDMRxBuffer[0]=='a') {…}

che essendo sempre vero lo introduce in una parte di codice dedicata alla lettura del carattere 'X' e al suo invio tramite *MiWi*. Il rischio che mandando rapidi comandi alcuni vengano persi non sussiste in termini pratici perché il *PIC* opera a velocità molto elevate, più della capacità umana di passare da un comando all'altro. Oltretutto il fatto che comandi troppi rapidi, così tanto rapidi, siano ignorati mantiene un aspetto positivo nel senso di salvaguardia dell'apparato meccanico, sennò eccessivamente '*stressato*'.

2.3.4 Implementazione della comunicazione wireless fra la scheda ZigKey e la scheda ZigKey_trenino

Per quanto riguarda l'invio e ricezione tramite trasmettitore (e relativo protocollo *MiWi P2P*) dei caratteri inviati a monte da *VB* non è stato necessario modificare, se non minimamente, il codice pre-implementato. Infatti il codice fornisce già le seguenti funzioni per svolgere le funzioni di invio e ricezione di un pacchetto tramite i trasmettitori *MRF24J40*. Per l'invio:

- MiApp_FlushTx(). Questa macro resetta il puntatore del buffer di trasmissione (TXbuffer);

- MiApp_WriteData(BYTE). Questa macro scrive un byte del payload dell'applicazione nel TXbuffer.

Un esempio di codice risulta quindi essere:

<code>
MiApp_FlushTx();
MiApp_WriteData(AppPayload[0]);
MiApp_WriteData(AppPayload[1]);
<code>

Nel nostro specifico caso il codice assume la seguente forma:

```
if (RTDMRxBuffer[0]=='a')
        {
        MiApp_FlushTx();
       MiApp_WriteData(RTDMRxBuffer[1]);
        }
```

nella quale si nota la condizione *if (RTDMRxBuffer[0]=='a')* che corrisponde a livello logico, come precedentemente spiegato, ad un *flag* segnalante che tramite *UART* è giunto un pacchetto, seguita poi dal codice di invio vero e proprio.
Per quanto riguarda la ricezione le funzioni già implementate sono:

- BOOL MiApp_MessageAvailable(void). Questa funzione restituisce un boolean se è giunto un pacchetto tramite MiWi;
- void MiApp_DiscardMessage(void). Questa funzione rilascia il pacchetto ricevuto per l'applicazione e notifica allo strato di protocollo (protocol layer) che è pronto a ricevere il prossimo messaggio.

Nel nostro caso il codice assume la seguente forma:

```
if (MiApp_MessageAvailable())
    {
    h = rxMessage.Payload[0];
    MiApp_DiscardMessage();
    }
```

dove nella variabile '*h'* viene salvato il contenuto del *buffer di ricezione rxMessage.Payload*.

In questa maniera è stato possibile creare la catena di comunicazione descritta nell'introduzione di questa parte, catena che è stata testata agendo sui led della scheda *ZigKey_trenino*, comandati ad illuminarsi secondo precise logiche temporali a seconda del carattere inviato dall'interfaccia *VB*.

2.4 Sviluppo del sistema di controllo

2.4.1 Inizializzazione Oscillatore, Porte, OutputCompare, Timers, ADC

Oscillatore: la famiglia *PIC32* offre diversi clock interni che derivano da sorgenti di clock interne o esterne; alcune di queste sorgenti sono fornite di *PLLs (Phase-Locked Loops)* e di divisore programmabile in ingresso e/o in uscita, utili per ottenere la frequenza desiderata. I clock principali sono quello di sistema *(SYSCLK)*, usato dalla *CPU* e da alcune periferiche, ed il *Peripheral Bus Clock (PBCLK)*, usato dalla maggioranza delle periferiche. Noi abbiamo scelto di utilizzare l'oscillatore interno *Fast RC* a *8 MHz* nominali e di sfruttare la modalità *PLL*. In questa modalità il divisore in ingresso al *PLL* è obbligatoriamente pari a 2 per fornire un input a *4 MHz* al modulo *PLL* mentre il moltiplicatore ed il divisore in uscita sono stati da noi impostati rispettivamente a 20 e 2 per avere in uscita la frequenza di lavoro desiderata di *40 MHz* per il dispositivo (*SYSCLK*). Il *PBCLK* è stato impostato a *20 MHz* e *l'SPI BaudRate* a *10 MHz*.

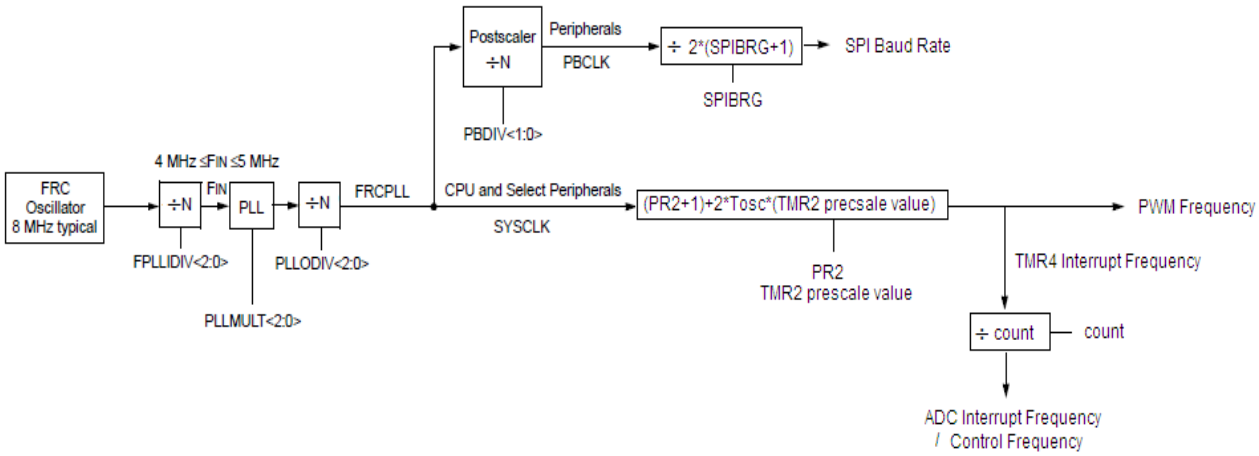


Figura 98 - schema della generazione e propagazione del segnale di clock

Porte: i pin a scopo generico di *input/output* permettono al microcontrollore di monitorare e controllare gli altri dispositivi. Per aumentare la flessibilità e la funzionalità, alcuni di questi pin sono multiplexati con altre funzioni, legate alle periferiche presenti sul dispositivo. In base alle nostre necessità abbiamo impostato degli ingressi per:

1. l'interrupt esterno proveniente dal modulo *MRF24J40MA;*
2. i dati provenienti via *SPI* dal modulo *MRF24J40MA (Serial Data Input);*
3. i segnali da campionare tramite il modulo *ADC*, definiti poi come analogici.

E delle uscite per:

4. i dati destinati via *SPI* al modulo *MRF24J40MA (Serial Data Output);*
5. il clock che regola la comunicazione col modulo *MRF24J40MA;*
6. i comandi di *enable, wake* e *reset* destinati al modulo *MRF24J40MA;*
7. i segnali di comando del *ponte-H;*
8. i segnali di comando dei due *LED* presenti sulla scheda del motore.

OutputCompare: vengono utilizzati per generare le onde quadre con *duty* variabile da fornire in ingresso al *Ponte-H*. Verrebbe spontaneo utilizzare la modalità "*PWM*" ma per motivi che spiegheremo in seguito abbiamo scelto di utilizzare la modalità "*Dual Compare*"; questa modalità vede un'uscita pilotata inizialmente bassa fino a che il valore settato nel registro *OCxR* è maggiore rispetto al valore assunto dal *timer* legato all'*OutputCompare* (che inizializzeremo qui di seguito). Quando i due valori coincidono, l'uscita viene pilotata alta finchè il valore del *timer* e quello del registro *OCxRS* non coincidono. All'incontro dei due, l'uscita torna pilotata bassa. Per il corretto funzionamento ovviamente dovrà essere $\mathrm{OCxR} < OCxRS < PRy$ dove *PRy* è il periodo del *timer y* associato all'*OC x*, in corrispondenza del quale il *timer* si azzera e riparte.

Timers: il *timer* associato agli *OutputCompare* è il *Timer2*, che si incrementa ogni periodo del *Peripheral Bus Clock*. Valore di *prescale* e periodo sono stati scelti al fine di ottenere una frequenza di onda quadra pari a *50 KHz*, adempiente i vincoli del *ponte-H*, il quale accetta in ingresso frequenze comprese tra i *20* e i *100 KHz*. Un altro timer, il *Timer4*, è stato inizializzato in modo identico al *Timer2*, con la sola differenza di essere sfasato con esso di metà del periodo: di questo *timer* sfrutteremo l'interrupt generato nell'istante in cui il conteggio equivale al periodo (*TMR4=PR4*).

ADC: dichiarati analogici gli ingressi al convertitore analogico-digitale, quali la tensione di alimentazione (*Vdcbus*) e la misura di corrente assorbita dal motore (ed eventualmente altri ingressi come le misure utili ad ottenere la *f.e.m.*), si è deciso di sfruttare solo uno dei due *multiplexer* a disposizione dell'*ADC* ed attivare su questo la scansione degli ingressi. Per i valori campionati è stato scelto il formato "*Unsigned Integer 16 bit*", che in realtà utilizza 10 bit senza segno. L'interrupt legato all'*ADC* è stato impostato dopo ogni 2 sequenze *sample/convert*, ricordando che le uniche due misure effettuate sono quelle di corrente e *Vdcbus*. Il tempo di *sample/convert*, multiplo del periodo di *Peripheral Bus*, è stato settato pari a *8.6 µs*, perciò l'*interrupt* di *ADC* sarà generato *17.2 us* dopo l'inizio del *sample* del primo ingresso. La conversione è stata settata come automatica conseguenza al campionamento, mentre per quanto riguarda il campionamento si sono utilizzate due modalità diverse: se si è finita la conversione del primo ingresso, si passa in automatico al campionamento del secondo ingresso; se si è finita la conversione del secondo ingresso ed è quindi generato l'*interrupt*, il campionamento viene momentaneamente fermato in attesa di un preciso istante. Questo particolare istante e questa scelta di modalità di campionamento sono collegate alla scelta della modalità "*Dual Compare*" per gli *OC*.

2.4.2 Metodo di campionamento della corrente di indotto

Al fine di ottenere delle misure di corrente affidabili, requisito fondamentale per realizzare un buon controllo, si è pensato di realizzare il segnale ad onda quadra da fornire al *ponte-H* in modo tale che il livello alto fosse centrato nel mezzo del periodo del *Timer2* e di attivare in modalità manuale il campionamento della corrente proprio in questo istante. Per far questo è necessario sfruttare l'interrupt del *Timer4*, sfasato di mezzo periodo rispetto al *Timer2*, che è legato alla generazione degli *OutputCompare*.
Utilizzando la modalità "*PWM*" e settando il campionamento automatico, si sarebbe corso il rischio di avere delle misure istantanee di corrente errate (o addirittura nulle), come dimostra il la Figura 99 mostrata nella pagina seguente.

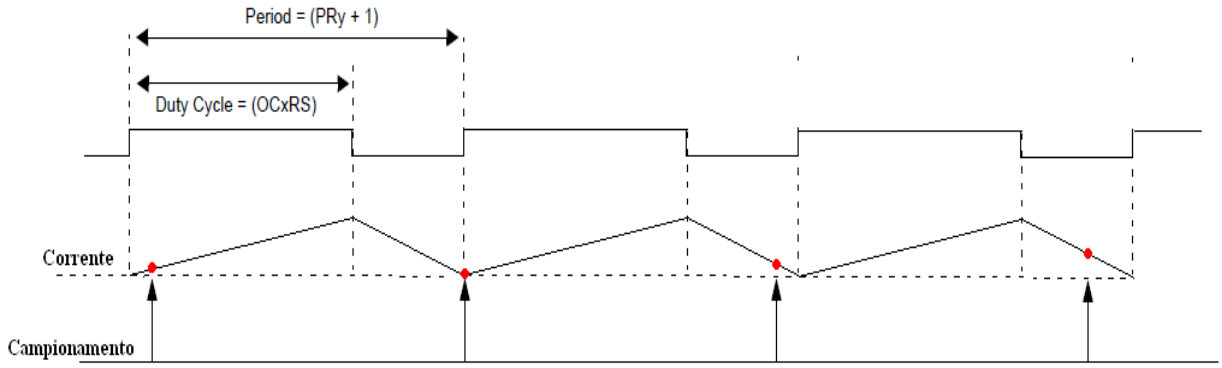


Figura 99 - campionamento della corrente in modalità PWM

Con il metodo utilizzato, invece, si è certi che nell'istante di campionamento la corrente sia sempre a metà della sua salita, ottenendo così delle misure affidabili della corrente media circolante.

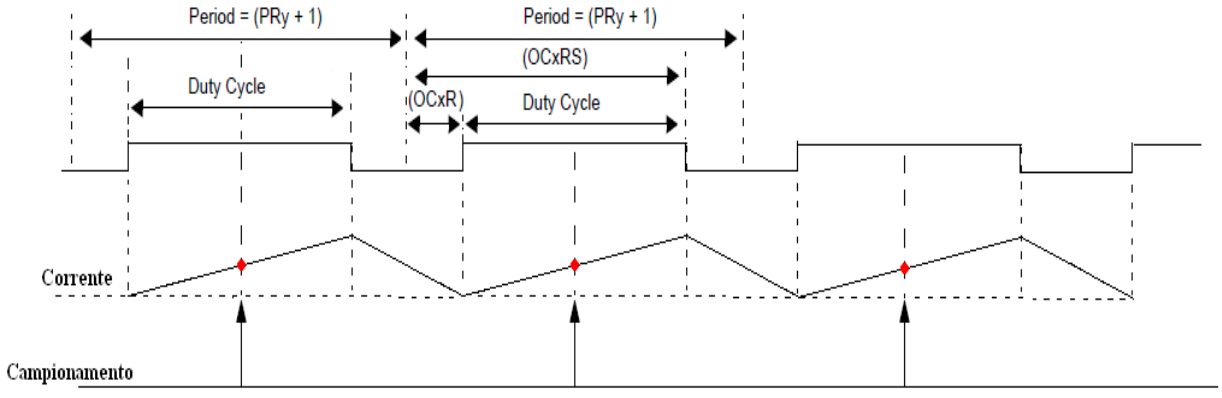


Figura 100 - campionamento della corrente in modalità Dual Compare

La misura di corrente viene fatta sul primo canale analogico in ingresso all'*ADC*, mentre la misura di *Vdcbus* sul secondo canale: si può sfruttare l'interrupt generato dopo il *sample/convert* di

queste due misure per fermare le operazioni automatiche e l'interrupt del *Timer4* per abilitare il campionamento della misura di corrente nell'istante opportuno.
Per avere il livello alto dell'onda quadra dove desiderato, i valori dei registri OCxR e OCxRS vengono calcolati in funzione del periodo del *Timer2* $(\mathrm{PR2}+1 = 400)$ e della variabile di controllo generata dall'anello interno di corrente (*contr*); ad esempio:

$$\mathrm{OC1R} = (400 - \mathrm{contr}) >> 1\,;\; OC1RS = (406 + \mathrm{contr}) >> 1\,;$$

Come vediamo, per *contr = 0*, il che corrisponde a richiedere duty nullo, si ha *OC1R = 200* e *OC1RS = 203*, mentre per *contr = 360*, corrispondente al *90%* di *duty cycle* impostato da noi come massimo, si ha *OC1R = 20* e *OC1RS = 383*.
I livelli alti di onda quadra appaiono così essere centrati circa intorno a metà del periodo del *Timer2*. Come si può notare, nel codice è necessario distanziare minimamente gli istanti di salita e discesa dell'onda quadra, anche quando idealmente vorremmo essi coincidessero, per avere *duty cycle* perfettamente pari allo *0%*.
In fase successiva si è deciso di effettuare le misure ed il controllo presente nello stesso *ADC interrupt* ogni 6 periodi di *PWM,* sfruttando un semplice contatore all'interno dell'*interrupt* del *Timer4*. Così facendo le misure ed il controllo avvengono con frequenza pari a *8.333 KHz*.

2.4.3 Gestione delle Interrupt Service Routines

Si è scelto di mantenere la modalità "*Multi-Vector*" già impostata nel codice originale della *ZigKey* e di apporvi le opportune modifiche. La modalità "*Multi-Vector*", a differenza della modalità "*Single-Vector*", ha un ISR dedicata per ciascun interrupt e questo ha il vantaggio di poter impostare priorità maggiore agli interrupt considerati più urgenti, esaltandone la velocità di risposta.
Le ISRs di nostro interesse sono tre, corrispondenti ai tre interrupt da noi utilizzati:

1. *void __ISR(_EXTERNAL_1_VECTOR, ipl4) _INT1Interrupt(void){...}* viene eseguita in corrispondenza dell'interrupt esterno inviato dal modulo *MRF24J40MA*. Questa ISR avverte il *PIC* della presenza di un dato da leggere nell'apposito buffer di ricezione, procede alla lettura e controlla che la comunicazione tra i due moduli sia avvenuta con successo. A questa ISR è stata assegnata la priorità maggiore, così da avere una risposta immediata a ciascun comando inviato da *Visual Basic*.
2. *void __ISR(_TIMER_4_VECTOR, ipl3) _TMR4Interrupt(void){...}* viene eseguita in occasione dell'interrupt generato in corrispondenza del "riempimento" del *Timer4*. Questa ISR gestisce l'istante di campionamento della corrente e contiene il contatore che impone la frequenza di campionamento dell'ADC e conseguentemente quella di controllo. Ad essa è stata assegnata priorità intermedia, dato che questa impone la frequenza d'esecuzione dell'*ADC interrupt*.

3. *void __ISR(_ADC_VECTOR, ipl1) _ADCInterrupt(void){...}* viene eseguita al termine di ogni 2 sequenze di *sample/convert* dell'ADC, ovvero ogni qualvolta siamo in possesso di misure aggiornate sia di corrente che di *Vdcbus*. All'interno di questa ISR si trova, oltre alla lettura delle misure, lo stimatore di velocità, tutta la parte di controllo relativa al motore, consistente nel regolatore esterno di velocità e quello interno di corrente, e la logica abilitante l'inversione di moto, utile in frenatura. Ad essa è stata assegnata la priorità minima, dipendendo fortemente dalle altre due ISRs.

2.4.4 Caratterizzazione del sensore di corrente

Per la caratterizzazione sono state eseguite alcune prove a rotore bloccato in cui veniva visualizzata sull'oscilloscopio la corrente assorbita dal motore e, sfruttando i led presenti sulla scheda, si andava passo passo a cercare il corrispondente valore digitale misurato. Da queste prove, tramite interpolazione, si è ottenuta la seguente caratteristica:

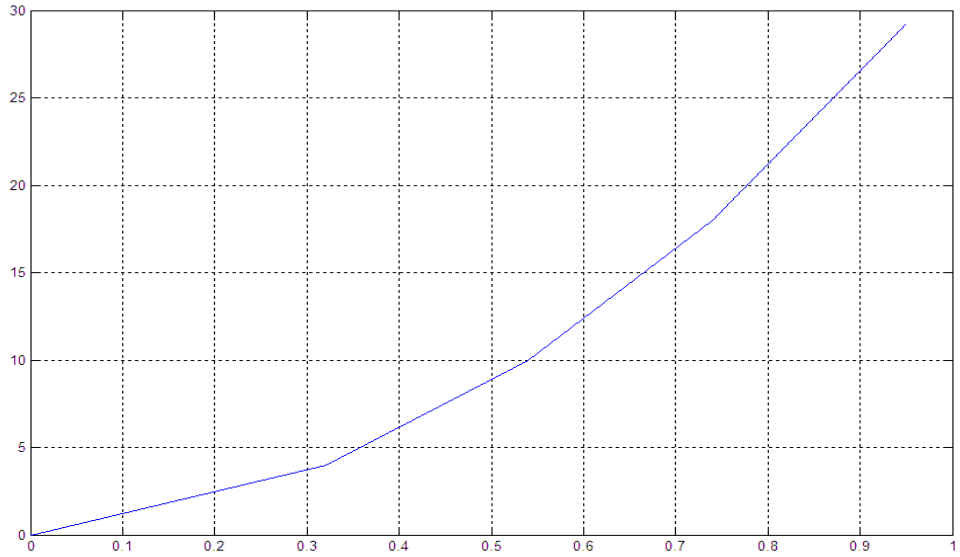


Figura 101 - caratteristica del sensore di corrente

Utilizzando come sensore uno *shunt*, ovvero una resistenza, la caratteristica dovrebbe essere perfettamente lineare. La leggera non linearità a bassa corrente è dovuta all'imprecisione connessa all'utilizzo dei led per valutare il valore in uscita unita alla variabilità di detta corrente a bassi valori. Altro elemento che tende a sporcare la misura è che, per come è stato impostato, il guadagno dell'amplificatore connesso allo shunt non permette di sfruttare appieno la dinamica a disposizione della corrente (al valore massimo di *850mA* corrisponde un valore digitale di *35*, su *255* livelli a disposizione), ma bensì solo il *13%*. Di conseguenza le misure di corrente *basse*, già imprecise, ne risentono maggiormente.

2.4.5 Regolatore di Corrente

Il progetto del regolatore di corrente si è basato sul classico regolatore *alla Guardabassi*. Ponendo:

$\frac{Kp}{Ki} = \tau$, dove $\tau = \frac{L}{R}$ costante elettrica del motore

si è sicuri di ottenere un margine di fase di 90° per qualunque pulsazione. Avendo deciso di avere una banda passante *Wc* pari a *1000 rad/s*, possiamo calcolare *Kp* semplicemente ponendo $Kp = L \cdot Wc$. Discretizzando il regolatore con una trasformata di *Eulero in avanti*, si ottiene un regolatore del tipo:

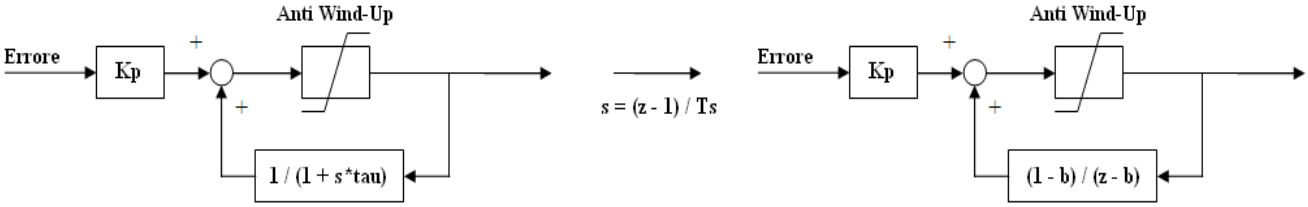


Figura 102 - regolatore di corrente in configurazione anti-wind up: discretizzazione con Eulero in avanti

dove $b = 1 - \frac{Ts}{\tau}$.

In ingresso al regolatore si è deciso di normalizzare la corrente, sia misurata che di riferimento, rispetto al valore massimo che si desidera circoli nel *ponte-H*, pari a circa *850 mA*. Ovviamente nel codice questa normalizzazione è stata realizzata rispetto al valore digitale corrispondente a *850mA*, ovvero in *base 14*. Sia la misura che il riferimento di corrente, inizialmente definite come *unsigned*, vengono rese *signed*, imponendo il segno in base alla direzione di marcia del trenino (in avanti corrente positiva, indietro corrente negativa).
Tarato in questo modo il regolatore e limitata in uscita la variabile di controllo del *duty* al 90%, si è passati alle prove a rotore bloccato per verificare la validità della regolazione di corrente. La necessità di effettuare tali a prove a rotore bloccato risiede nel fatto che a rotore libero subentra, nell'equazione del motore, la fem. Di conseguenza la corrente, essendo limitata dalla fem, non riesce ad andare sopra un certo valore di regime, nel nostro caso ca 200mA, e il regolatore, a cui son forniti riferimenti man mano sempre più alti fino al valore massimo di 850mA, inizia a vedere un errore costante che però la sua azione non può ridurre. Viceversa a rotore bloccato la corrente assorbita dal motore può assumere tutti i valori da *0* a *850mA*, permettendo così di osservare il comportamento '*corretto*' del regolatore.
Per effettuare le prove è stato modificato il codice così che i riferimenti di velocità forniti tramite interfaccia *VB* corrispondessero a riferimenti di corrente, con range *0-850mA*.
Nella Figura 102 si può apprezzare la risposta a un gradino positivo (70 → 400mA) e alla relativa discesa (400 → 70mA).

Come si può notare nel caso della salita la corrente va a regime in *ca 200µs*. Il regolatore effettua il suo compito come ci si aspettava, e questo può essere colto osservando l'andamento del *duty cicle* nell'immagine (il *duty cicle* mostrato è quello della prima gamba del *ponte-H* ed è in logica negata). Infatti finchè il riferimento di corrente è pari a *70mA* il *duty cicle* è pari allo *10%*. Appena arriva il riferimento a *400mA* il regolatore vede l'errore e apre completamente il *duty cicle*, che per tutta la salita vale *90%* (il *duty cicle* è limitato a tale valore massimo).

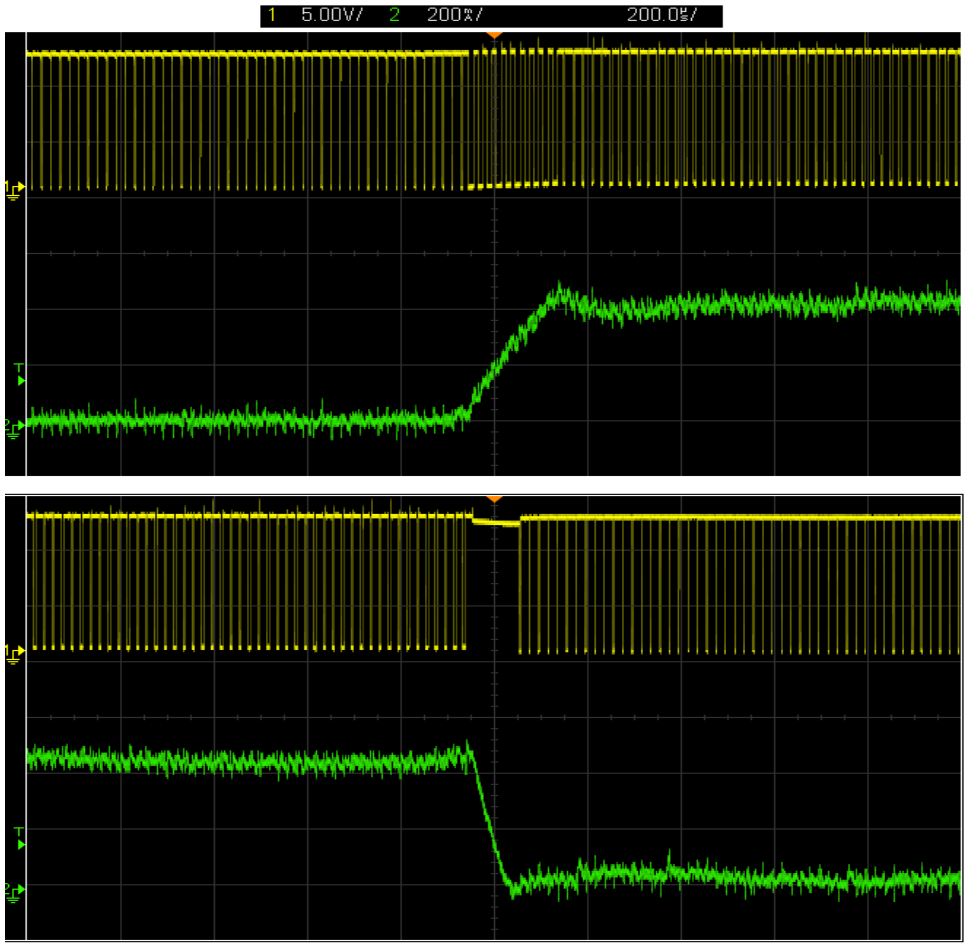


Figura 103 – andamento della corrente in risposta a un gradino da 0 a 400 mA (sopra) e di uno da 400 a 0 mA (sotto)

Superato il riferimento, causa leggera sovraelongazione, il controllo pone nuovamente a *0* il *duty cicle*, per fare abbassare la corrente al valore esatto. Non appena la corrente raggiunge il valore esatto, il regolatore impone un *duty cicle* di regime di *ca 50%*.
Per quanto riguarda la discesa, questa avviene in *ca 80µs*. Come si può notare il *duty cicle* passa dal valore di regime del *50%* a un valore nullo nella fase di transitorio, per poi assestarsi sul *10%* corrispondente al riferimento di *70mA*. Se nella fase in cui il *duty cicle* è nullo si osservasse il *duty cicle* della seconda gamba questo risulterebbe pari al *90%* (il controllo prevede infatti l'inversione del controllo quando l'errore cambia segno).
La maggiore velocità del transitorio di discesa rispetto al transitorio di salita è dovuta al fatto che durante il transitorio da *0* a *400 mA* il duty passa dal *10%* al *90%*, quindi al motore viene fornito un gradino di *ca 9V*, mentre nel transitorio da *400* a *0 mA* si passa da un *duty cicle* a regime del *50%*,

positivo, a un *duty cicle* del *90%*, negativo. Quindi ai capi del motore si ha un passaggio dai *ca +5.5V* forniti al motore a regime a *ca - 10V*, con un gradino effettivo di tensione di *ca 15.5V*, maggiore di quello fornito durante il transitorio da *0* a *400mA*. Ovviamente ciò comporta che la corrente nels econdo caso avrà un transitorio verso il suo stato di equilibrio più rapido.

2.4.6 Stimatori di velocità

2.4.6.1 Stimatore di velocità tramite stima della f.e.m.

Realizzato il regolatore di corrente si è quindi passati all'implementazione della stima di velocità, così da poter poi intraprendere il progetto del regolatore esterno di velocità. Il primo stimatore, dimostratosi in seguito la scelta migliore, è quello basato sul calcolo della velocità a partire della stima della fem effettuata tramite l'equazione *E=V-RI.*
Tenendo conto che l'induttanza del motore è molto piccola, si è deciso di ignorare il suo contributo. Le misure di cui si necessita sono quella di corrente, di cui il sensore è già stato tarato, e quella della tensione sul motore. Non essendoci un sensore della V_{motore} tale misura è stata ottenuta tramite l'operazione:

$$V_{motore} = V_{DC\ BUS} \cdot \delta$$

Dove δ è il *duty cicle*. Infatti $V_{DC\ BUS}$ è un valore fisso (12V = 202/255 in livelli digitali) e il *duty cicle* è la variabile di controllo, che quindi il microcontrollore già conosce e 'possiede'. Una volta ottenuta la misura della *fem* per passare da *fem* a velocità basta dividere la prima per la costante *K* di velocità (in tutto ciò va tenuto conto della caduta di tensione sugli interruttori statici, pari a *ca 1V*).

2.4.6.2 Stimatore di velocità tramite misura della f.e.m.

Il secondo stimatore testato è quello basato sul calcolo della velocità a partire della misura diretta della *fem*. Per tale misura sono stati predisposti sulle uscite del *ponte-H*, ai morsetti del motore, due partitori di tensione in grado di fornire due misure di tensione la cui differenza, in una particolare condizione di funzionamento del *ponte-H*, coincide con la misura della *fem.*
Lo svantaggio principale di questo metodo è proprio legato a questa condizione di funzionamento: infatti, per poter misurare la *fem,* è necessario aprire il ponte completamente (tutte e quattro le valvole aperte) cosicchè la tensione ai capi del motore, essendo scollegata l'alimentazione,

corrisponderà a *E* (corrente assorbita i_i= 0). Per poter mantenere ben aggiornata la misura ciò deve essere fatto frequentemente, con la conseguente perdita di potenza del motore durante il campionamento delle due tensioni. Oltretutto tale misura può essere affetta da grandissimo rumore, in quanto se si vuole evitare di far perdere troppa potenza al motorino è necessario aprire il *ponte_H* per poco, e ciò comporta che l'effetto delle commutazioni si sovrapponga alla misura stessa della *fem*.
Ignorando per un istante queste problematiche in ogni caso l'utilizzo di questo stimatore comporta le seguenti modifiche al codice:

- abilitare gli ingressi analogici per le due nuove misure;
- abilitare tali ingressi alla scansione da parte del *multiplexer* in uso lasciando inalterato il numero di sequenze *sample/convert* dopo le quali avviene l'interrupt dell'ADC;
- Implementare all'interno dell'*ADC interrupt* un contatore che riconosca se l'*ADC interrupt* corrente stia misurando la coppia *Corrente-Vdcbus* o le due tensioni da cui dipende la *fem;*
- imporre quindi l'apertura del *ponte-H* prima del campionamento delle tensioni e subito dopo ritornare al controllo normale del *ponte-H*. Il fatto di compiere la misura di corrente e la misura di *fem* in due istanti diversi introduce un ulteriore componente d'errore al controllo);
- modificare la logica di attivazione del campionamento della corrente, aumentando la frequenza di controllo.

Una volta acquisite le due misure di tensione, la differenza viene fatta via software e quindi utilizzata, scalata per l'inverso della costante di motore, come stima di velocità. Da notare che con questo stimatore la misura di *Vdcbus* è di fatto inutilizzata.
Dopo alcuni test con questo stimatore, si è dimostrato che le problematiche legate alla perdita di potenza del motore e all'assoluta rumorosità delle misure (praticamente inutilizzabili) impedivano qualsiasi ulteriore idea di sviluppo di questo stimatore e suo utilizzo in anello di controllo. E' quindi stato deciso di passare ad analizzare un ulteriore metodo di stima della velocità.

2.4.6.3 Stimatore di velocità tramite stima della frequenza dei picchi di corrente assorbita

Questo stimatore, si basa sull'analisi della componente di *ripple* della corrente; infatti, come già introdotto, mentre la componente continua fornisce potenza al motore, la componente di *ripple* è legata al passaggio delle spazzole da un segmento di collettore al successivo e si presenta come un'onda triangolare. Rilevando i picchi di questa onda e monitorando il tempo trascorso tra essi è così possibile ottenere una stima della velocità. Infatti, maggiore sarà la velocità, maggiore sarà la frequenza dei picchi.

Per prima cosa si è voluto osservare tramite sperimentazione pratica se effettivamente tramite il conteggio dei picchi fosse possibile ottenere una stima reale della velocità.
Ci si è quindi muniti di un contagiri a inerzia: tale strumento, visibile in Figura 103, messo in rotazione dall'albero motore misura i giri al minuto (*rpm*) del motore.

Figura 104 – sensore di velocità analogicoinerziale connesso al motore

A questo punto alimentando il motore in anello aperto con diverse tensioni si sono effettuate le misure di velocità col contagiri e nello stesso tempo rilevato, tramite oscilloscopio, le distanze fra i picchi della corrente, visibili nell'immagine seguente.

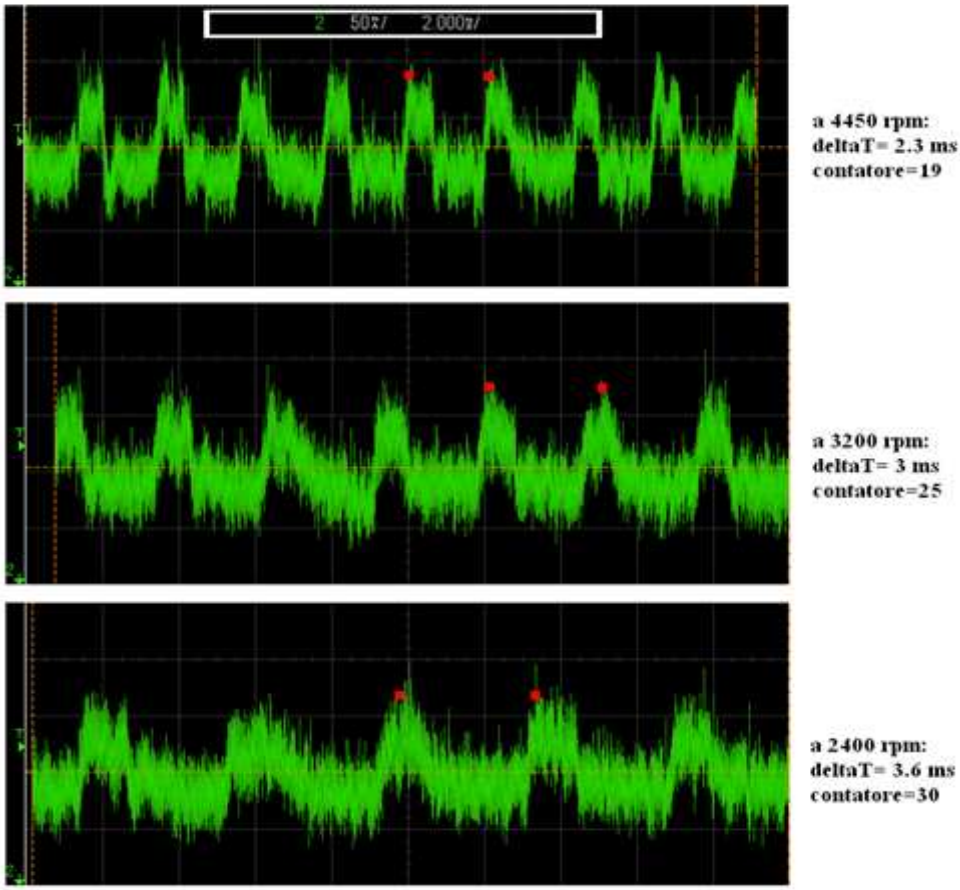


Figura 105 – picchi nelle corrente di indotto a diverse velocità di funzionamento

Il risultato ottenuto è stato stupefacente per precisione e linearità della caratteristica, come si può notare dal seguente grafico, ottenuto interpolando i valori ottenuti.

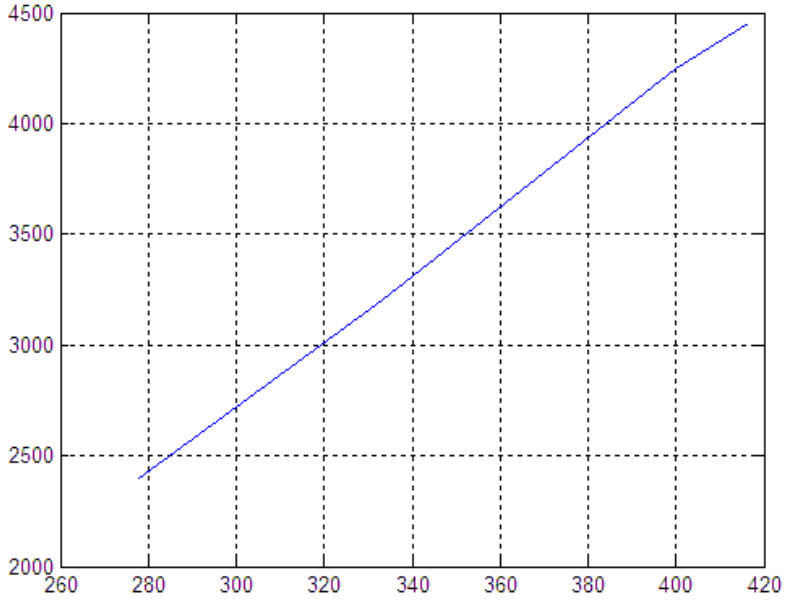


Figura 106 – relazione fra le misure della frequenza dei picchi e la misura del numero di giri al minuto

Dimostrata la bontà del metodo si è voluto approcciare una possibile soluzione dal punto di vista del codice. Mantenendo il codice usato con il primo stimatore di velocità, si è andati a modificare solo la parte legata alla stima. L'idea è la seguente: ogni qualvolta si riceva un nuovo comando di velocità da *Visual Basic*, si utilizzano le prime misure di corrente per valutare il valore di picco e di valle dell'onda triangolare, quindi si utilizzano le successive misure per valutare la frequenza dei picchi. Per evitare valutazioni errate causate dalla presenza di picchi spuri dovuti al rumore, in realtà non si misurerà la frequenza tra i picchi bensì la frequenza tra gli intorni dei picchi.
Tramite test all'oscilloscopio, si è ricavato che alla velocità massima la distanza tra questi picchi è di circa *2 ms* e che per velocità minori questo tempo aumenta: sfruttando l'*ADC interrupt*, il quale scatta ogni *120 µs*, per contare gli istanti intercorsi tra gli intorni di due picchi successivi, possiamo superare il problema del rumore sulla triangolare accettando solo valori del contatore maggiori di 15. Infatti al minimo si avrà $\frac{2}{0.12} = 16$ circa.
Per conferma, grazie all'oscilloscopio, si può osservare che per velocità basse, la distanza tra i picchi si aggira intorno ai *22 ms*, per cui il contatore assumerebbe circa il valore di 183. Il contatore risulta quindi inversamente proporzionale alla velocità e può rappresentarne una stima.

L'implementazione di questo stimatore è stata approcciata ma non portata a termine, in quanto andava oltre gli obiettivi prefissati nel nostro progetto ed in quanto si era già ottenuto uno stimatore, il primo, che assolve bene il suo dovere. Ciò nonostante fornisce uno spunto molto interessante per futuri sviluppi del progetto.
Di seguito si è proceduto con la sintesi del regolatore di velocità utilizzando il primo stimatore.

2.4.7 Regolatore di velocità

Avendo scelto di utilizzare il primo stimatore di velocità, si sono normalizzate la velocità stimata e quella di riferimento rispetto al valore massimo e sono state portate anch'esse, come la corrente, in *base 14*; infine si è assegnato loro il segno sfruttando la variabile "*Direction*" del motore (ovvero a seconda della direzione del treno cambia il loro segno). Per rendere più semplice la sintesi del controllo inizialmente si è utilizzato uno stimatore ridotto, del tipo *V=E*, che fornisce sicuramente una stima della velocità errata (si ignora il contributo della corrente) ma elimina i disturbi dovuti al rumore della misura di corrente rendendo più semplice una prima taratura; in seguito il termine *RI* è stato reintrodotto (tutti i grafici che verranno mostrati a fine capitolo riguarderanno il controllo con stimatore completo).
Il progetto del regolatore è il classico regolatore *PI*, discretizzato con *Tustin*:

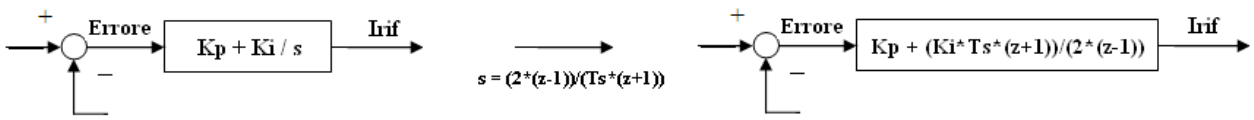


Figura 107 – regolatore PI: discretizzazione con Tustin

e la legge di controllo che ne deriva è del tipo:

$$I_{rif}(k) = I_{rif1}(k) + I_{rif2}(k)$$
$$I_{rif1}(k) = K_{pdig} \cdot Errore(k)$$
$$I_{rif2}(k) = I_{rif2}(k-1) + K_{dig} \cdot \big(Errore(k) + Errore(k-1)\big)$$

La componente integrale $I_{rif2}(k)$ viene opportunamente limitata nel codice così da garantire l'*antiwind-up*. L'errore in ingresso al regolatore è dato dalla differenza tra velocità di riferimento e velocità stimata. L'uscita del regolatore è invece la corrente di riferimento. Questa viene limitata al valore digitale corrispondente alla corrente massima assorbita da noi desiderata, pari a circa *850 mA*, normalizzata rispetto ad esso e quindi portata in *base 14*.
Per quanto riguarda i coefficienti K_{Pdig} e K_{Idig}, questi sono stati cercati utilizzando il metodo euristico di *Ziegler-Nichols*: tramite diversi test, prima si è alzato K_{Pdig} finchè il sistema non ha iniziato ad assumere un comportamento oscillante, quindi si è abbassato leggermente K_{Pdig} e si è alzato gradualmente K_{Idig} fino a raggiungere il comportamento del sistema desiderato.

Progettato così il regolatore si è osservato il comportamento del sistema controllato.

Si è fornito al motore, lasciato a rotore libero, uno scalino di velocità (riferimento finale comunque basso rispetto alla massima velocità) permettendogli di andare a regime. A questo punto si è bloccato il rotore con una rapida frenatura. L'effetto sulla corrente è quello visualizzato in Figura

107. La corrente è salita rapidamente fino al suo valore di saturazione, *ca 850mA*, dimostrando la bontà del regolatore. Infatti venendo rallentato l'errore di velocità è cresciuto, e il regolatore ha reagito imponendo una corrente (e quindi una coppia) sempre più alta, fino a quando non è stato possibile fornirne di più, assestandosi al valore massimo di *850ma*.

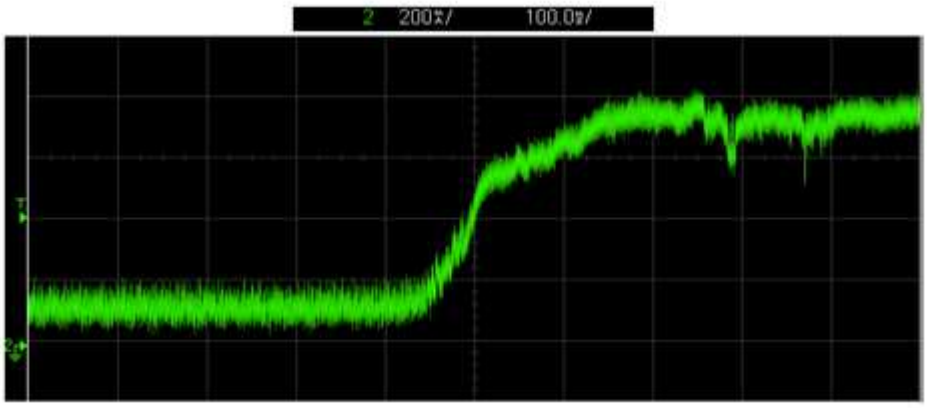


Figura 108 – andamento della corrente nel passaggio da rotore libero a rotore bloccato

Per precisione si mostra poi l'andamento dei *duty-cycle* delle due gambe (Figura 108: giallo prima gamba, verde seconda gamba) nel caso di massimo riferimento di velocità positivo e negativo (gli andamenti sono presi in situazione di regime, a rotore libero). Nell'immagine di sinistra si ha il *duty-cycle* della prima gamba al *90%* e quello della seconda nullo (il motore sta andando alla massima velocità in avanti). Nell'immagine di destra l'esatto contrario (il motore sta andando alla massima velocità indietro). Questo permette di mostrare che a massima velocità effettivamente il *duty-cycle* raggiunge il suo valore massimo, come ci si aspettava.
(Si ricorda che le misure rappresentate nell'immagine sono inverse: al *10%* di *duty cycle* rappresentato corrisponde il reale *90%* e al valore *High* il valore *Low*).

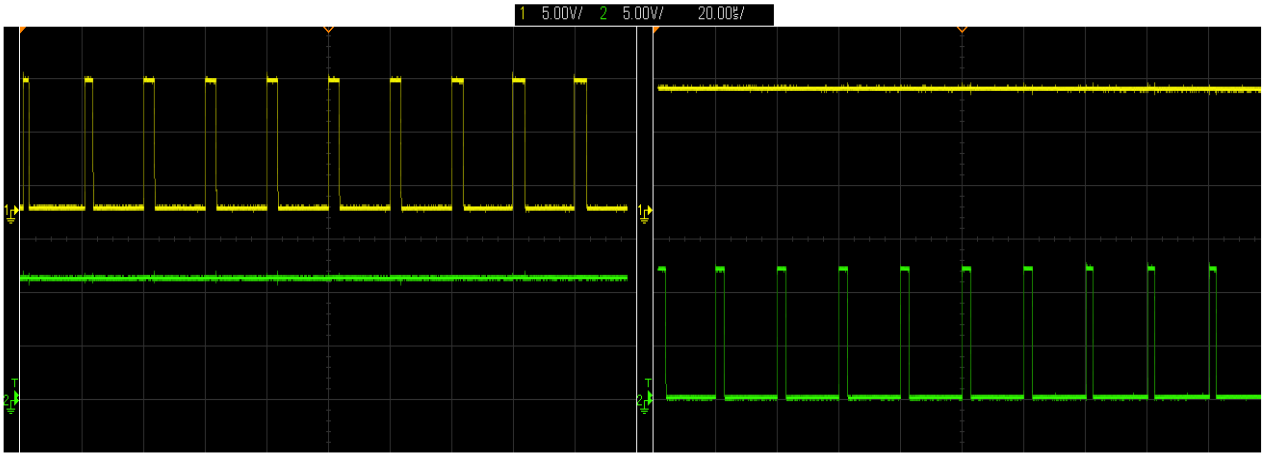


Figura 109 – duty cicles di entrambe le gambe a velocità massima positiva (sinistra) e negativa (destra)

Non avendo a disposizione encoder o un'uscita analogica sul microcontrollore, per avere un immagine mostrante il transitorio di velocità si sarebbe potuta fare una modifica hardware che

però, data la miniaturizzazione delle componenti, sarebbe stata rischiosa per una mano poco esperta.

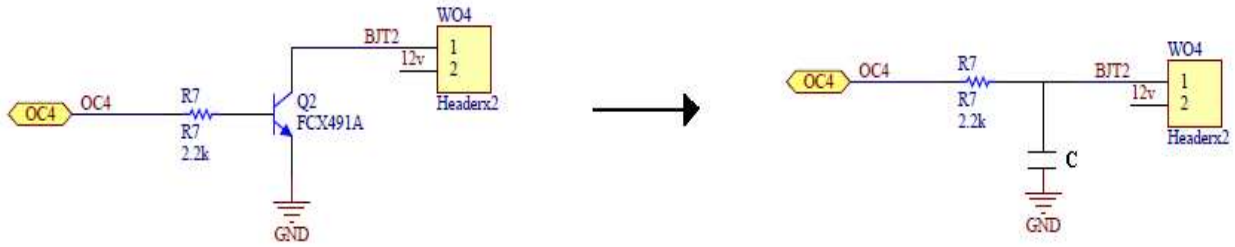


Figura 110 - schema circuitale della modifica hardware necessaria per avere una misura stimata della velocità

Questa modifica consisteva nello sfruttare una delle uscite *OutputCompare* predisposte del microcontrollore, dissaldare il transistor e saldare un opportuno condensatore *C* adatto a realizzare un filtro passa-basso. Così, copiando i valori OCxR e OCxRS responsabili del *duty cycle* su questa nuova uscita OC e filtrando opportunamente, a fronte di un gradino di velocità imposto da *Visual Basic*, si sarebbe potuta ottenere la misura del transitorio di velocità.
Per evitare problemi, si è deciso di salvare i valori X-Y del *duty cycle* direttamente dall'oscilloscopio, così da rielaborarli in *Matlab*.
Dopo aver importato in automatico dal file *.csv* i valori di tensione e tempo nei vettori *second* e *Volt*, si è realizzato il filtro:

$$y(k) = 0.995 \cdot y(k-1) + 0.005 \cdot x(k)$$

utilizzando la funzione "*filter*" in un codice del tipo:

$A = [1, -0.995]';$
$B = [0.005, 0]';$
$V = filter(B, A, Volt);$
$Vel = V \cdot 636.9545;$
$plot(second, Vel);$

Ottenuta in questo modo la tensione filtrata, si è passati alla velocità moltiplicando per l'opportuno coefficiente.
In questa maniera è stato possibile ottenere l'immagine del transitorio di velocità, mostrata nella Figura 111 a pagina seguente.

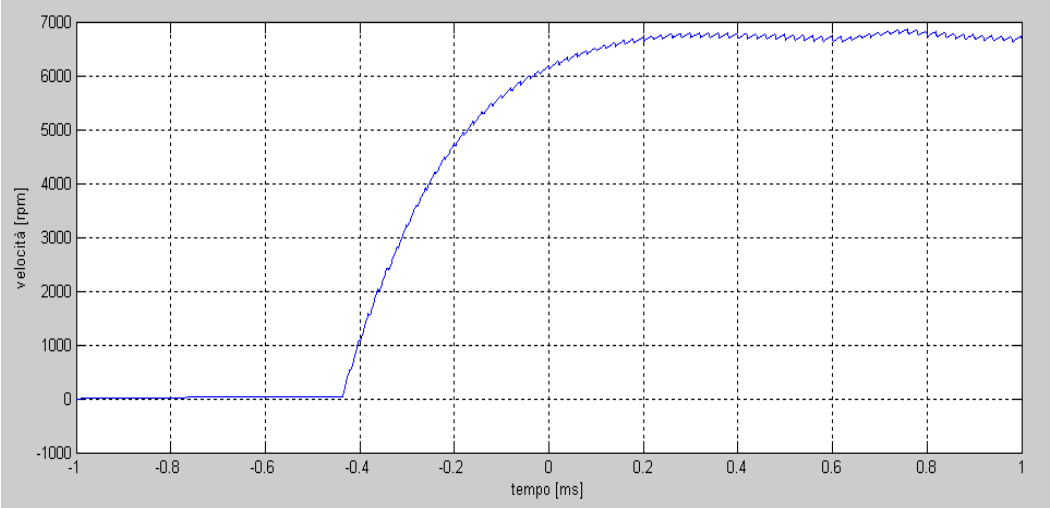
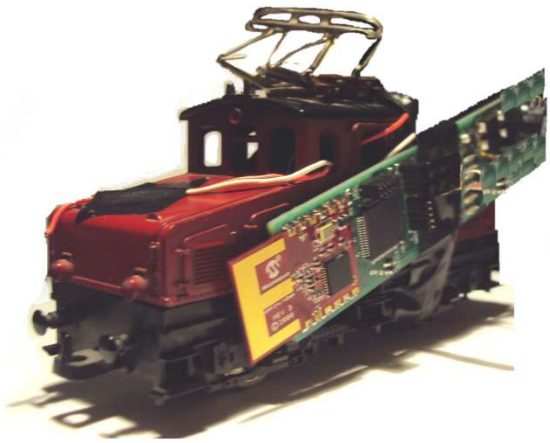


Figura 111 – andamento della velocità a fronte di un gradino sul riferimento di velocità di ampiezza massima

2.5 Adattamento del controllo al trenino

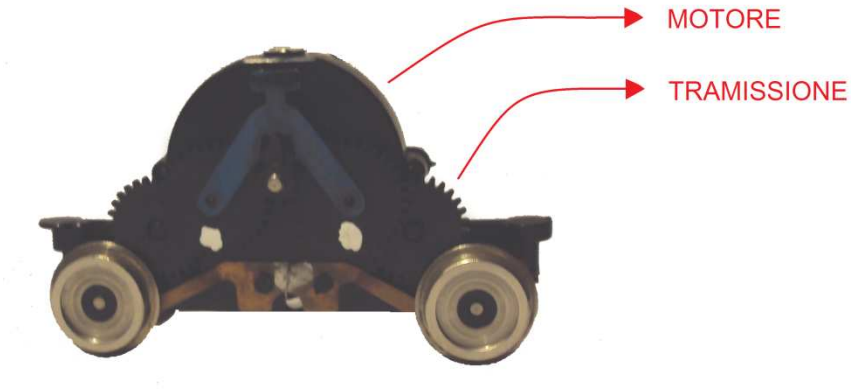
Tutto quanto è stato finora descritto è stato realizzato, per comodità operativa, su di un motore DC a 12 V senza carico. Nella realtà tale controllo deve però essere utilizzato su di un trenino, equipaggiato con un motore simile. Tale trenino rappresenta un sistema più complesso, in quanto entrano in gioco tutte le inerzie dovute alla trasmissione che prima non erano state considerate. Nelle Figure 111, 112 e 113 si può osservare il trenino dall'esterno, con la scheda collegata ad esso, e all'interno, con in evidenza il blocco motore-trasmissione-ruote. In un classico trenino da modellismo l'energia passa dai binari in tensione al motore tramite le ruote, a cui poi sono collegati dei conduttori che tramite due fili si connettono ai due morsetti del motore. Per inserire la scheda digitale e permettere che fosse questa a fornire la tensione al motore è stato necessario tagliare i due cavi creando i seguenti collegamenti: i due cavetti collegati ai conduttori a contatto con le ruote vengono connessi alla scheda digitale, permettendone tra l'altro l'alimentazione. Sull'uscita del Ponte-H della scheda vengono saldati due cavetti che si collegano coi morsetti del motore DC, permettendo quindi l'alimentazione del motore secondo la logica imposta dal controllore e fornita dal Ponte-H.



eda, visione dall'esterno

Per realizzare il controllo è stato necessario effettuare delle prove sul motore del trenino per ricavare i suoi parametri.

Si è quindi ottenuto:



e lato trasmissione

$L = 5.6$ mH;

$R = 16\ \Omega$;

$K = 0.779$ mV/rpm;

$\tau_{elet} = 1.49$ ms;

$J_m = 0.05 \cdot 10^{-6}$;

$J_r = 1 \cdot 10^{-7}$.

dove J_r è stata stimata validando il modello del motore con rispetto ai valori misurati.

A questo punto sono stati modificati i valori di *Kp* e *b* del regolatore di corrente al fine di ottenere ancora banda passante di *1000 rad/s* e si sono effettuati i test già fatti per il motore precedente ritrovando risultati pressochè uguali.
Soddisfatti del comportamento del regolatore di corrente si è quindi passati al regolatore esterno di velocità. Si sono ricercati quindi i nuovi valori di *Kpdig* e *Kidig* utilizzando il metodo di *Ziegler-Nichols*..

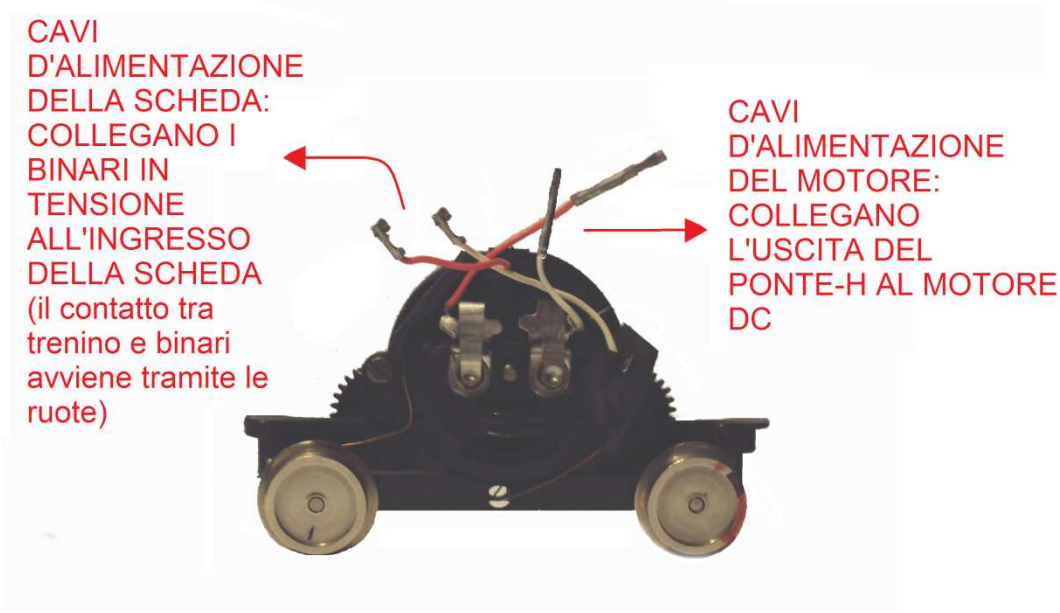


Figura 114 - blocco motore-trasmissione-ruote lato uscite/ingressi d'alimentazione

Purtroppo in questa fase si è messo in risalto un problema legato al motore del trenino che ha impedito una taratura ottima del controllore e di proseguire con l'adattamento. Infatti, superati i riferimenti di velocità più bassi, che venivano ben gestiti dal regolatore e dal motore, quando si richiedevano velocità man mano superiori la scheda andava in *reset* togliendo quindi l'alimentazione al motore.
Dopo innumerevoli test e controlli, si è arrivati alla conclusione che il problema non era legato al controllo, alle frequenze o a correnti eccessive che superavano quindi i limiti del Ponte-H. Probabilmente il problema era dovuto a dei picchi di *fem* causati dalle spazzole molto usurate (questo motore essendo molto vecchio produceva un rumore molto forte e sgradevole, segno dell'usura interna di cui era afflitto) che introducevano dei disturbi. Tali disturbi probabilmente causavano il cortocircuito del *Ponte-H* o, propagandosi nella scheda, direttamente il *reset* del Microcontrollore.
Non dipendendo da difetti del controllo si è deciso di non proseguire con l'adeguamento ma di fidarsi dei valori *Kpdig* e *Kidig* ottenuti in prima istanza lavorando a basse velocità, sapendo comunque che la bontà del controllo non veniva intaccata da tale problema.

2.6 Conclusioni

Come introdotto, in questo capitolo si sono descritte nella maniera più esauriente possibile le scelte e le procedure con cui si è giunti a portare a termine il lavoro, ovvero ottenere un controllo in velocità del trenino tramite una comunicazione *wireless* fra il generico utente e il sistema controllato. Per la visione dei codici generati e '*caricati*' sul Microcontrollore si rimanda all'appendice della tesi, dove sono riportati i codici più significativi.

Conclusioni

Il progetto ultimato ha portato al soddisfacimento degli obiettivi prefissati: la comunicazione è affidabile e che il controllo del trenino risulta stabile e performante.

Ripensando a quanto abbiamo realizzato si può sicuramente dire che si è trattato di un progetto di grande interesse: infatti ha permesso di addentrarsi per la prima volta ma in maniera molto dettagliata nel mondo dell'elettronica *embedded*, un campo che sta assumendo sempre maggiore importanza. Sotto questo aspetto è stato stimolante scontrarsi con la differenza fra la creazione di un controllo inteso come idea teorica e la sua realizzazione pratica, a livello di codice ed *hardware*, con tutte le difficoltà e approssimazioni del caso.

Siamo convinti che le tematiche di maggiore rilievo emerse da questo progetto siano fondamentalmente le seguenti: l'utilizzo del protocollo di comunicazione *wireless MiWi P2P* e l'implementazione, o comunque lo studio, laddove non sia stato possibile svilupparle, di tecniche di controllo *sensorless*.
La prima tematica rappresenta una novità molto recente nel campo delle reti locali *wireless*, ma, forte del supporto di *Microchip*, sta guadagnando vasti settori di mercato, rendendone quindi fondamentale la conoscenza. Per quanto riguarda le tecniche di controllo *sensorless* del motore, questa è una tematica di grande interesse sia da un punto di vista teorico, in quanto vastissima e stimolante è la ricerca che vi sta *a monte*, sia da un punto di vista pratico, in quanto per piccoli azionamenti rappresenta la scelta più ovvia ed economicamente logica.

Le prospettive del nostro lavoro non terminano con la redazione del presente elaborato: anzi, numerosi sono gli spunti per continuare tale progetto. Per esempio è possibile provare a sviluppare in maniera completa quelle tecniche *sensorless* da noi solo sfiorate, come la tecnica basata sul conteggio dei picchi della corrente di indotto.
Probabilmente però lo spunto più interessante è relativo al potenziamento della comunicazione *wireless*, rendendo cioè possibile comandare più trenini allo stesso tempo. Per far questo è necessario implementare una comunicazione *wireless* a stella, in cui ogni trenino abbia un codice di riconoscimento e, a seguito di un particolare comando inviato dall'utente, sia in grado di stabilire una connessione diretta tale per cui i comandi di velocià, o luce, giungano solamente a lui. Tutto ciò è già teoricamente previsto dal protocollo *MiWi P2P*, si tratta quindi di operare sul codice e sull'interfaccia *VB* per renderlo possibile.

Lasciando al futuro tali progetti, siamo soddisfatti di avere portato a termine quanto ci era stato richiesto ottenendo dei risultati molto gratificanti, ma soprattutto riconosciamo come la presente ricerca abbia apportato un arricchimento alle nostre conoscenze e competenze, permettendoci di investigare una parte di quello che potrà divenire il nostro mondo del lavoro.

Bibliografia

AcarnIey P.P., AI-Tayie J. K. [1997] *Estimation of speed and armature temperature in a brushed DC drive using the extende Kalman filter*, IEE Proceedings Electr. Power Appl., Vol. 144, No.1

Afjei E., Nadian Ghomsheh A., Karami A. […] *Sensorless Speed/Position Control of Brushed DC Motor*, […]

Bittanti S., [2005] *Serie temporali e processi casuali*, Pitagora editrice

Bolzern P., Scattolini R., Schiavoni N. [2004] *Fondamenti di controlli automatici*, McGraw-Hill Companies

Bowling S. [2010] *AN718 Brush-DC Servomotor Implementation using PIC17C756A*, Microchip Technology Inc.

Halvorson M. [2010] *Microsoft Visual Basic 2010 step-by-step*, Microsoft Press

Haugen F. [2010] *Tuning of PID controllers*, TechTeach

Li Y., Ang K.H., Chong G.C.Y. [2006] *PID control system analysis and design*, IEEE Control Systems Magazine 26(1)

Kurose J.F., Ross K.W. [2001] *Internet e reti di calcolatori*, McGraw-Hill Companies

MadhusudhanaRao G., SankerRam B.V. [2009] *A Neural Network Based Speed Control for DC Motor,* International Journal of Recent Trends in Engineering, Vol.2, No.6

Manigrasso R., Mapelli F.L., Mauri M. [2007] *Azionamenti elettrici*, Pitagora editrice

Mohan N., Undeland T.M., Robbins W.P. [2009] *Elettronica di potenza*, Hoepli editore

Nottelmann J.B. [2010] *Sensorless Rotation Counting in Brush Commutated DC motors*, IDEAdvance White Paper

Rajaram S., Murugesan S. [1978] *A New Method for Speed Measurement/Control of DC Motors*, IEEE Transactions on Instrumentation and Measurement, Vol.IM-27, No.1

Spirito P. [2006] *Elettronica digitale*, McGraw-Hill Companies

Vázquez-Sánchez E., Gómez-Gil J., Gamazo-Real J.C., Díez-Higuera J.F. [2012] *A New Method for Sensorless Estimation of the Speed and Position in Brushed DC Motors Using Support Vector Machines*, IEEE transactions on Industrial Elettronics, Vol.59, No.3

Yang Y., Flowers D. [2010] *AN1066 Microchip MiWi™ Wireless Networking Protocol Stack*, Microchip Technology Inc.

Yang Y. [2010] *AN1204 Microchip MiWi™ P2P Wireless Protocol***,** Microchip Technology Inc.

Yang Y. [2010] *AN1284 Microchip Wireless (MiWi™) Application Programming Interface – MiApp*, Microchip Technology Inc.

Yang Y. [2010] *AN1283 Microchip Wireless (MiWi™) Media Access Controller – MiMAC*, Microchip Technology Inc.

[2004] *Electrical Standards For Digital Command Control*, NMRA - National Model Railroad Association

[2004] *Communications Standards For Digital Command Control*, NMRA - National Model Railroad Association

[2006] *Extended Packet Formats For Digital Command Control*, NMRA - National Model Railroad Association

[2010] *PIC32MX3XX/4XX Data Sheet*, Microchip Technology Inc.

[2010] *PIC32 Family Reference Manual*, Microchip Technology Inc.

[2010] *MRF24J40 Data Sheet*, Microchip Technology Inc.

[2010] *18 V Max H-bridge Drivers Technical Notes*, Rohm Semiconductor

Allegati

Di seguito sono riportati i listati di codice più importanti. In particolare per primi si mostrano i codici *C* del programma *MPLAB IDE* caricati sul *Microcontrollore*. Di tali listati sono riportati quelli direttamente modificati da noi per implementare la comunicazione e il controllo, tralasciando quelli già presenti nel codice fornito e che non hanno subito modifiche.
Più precisamente per quanto riguarda la chiavetta *ZigKey* viene riportato il file *main.c*, da noi modificato per l'invio dei caratteri tramite protocollo *MiWi*, e il file *console.c*, contenente la macchina a stati per la ricezione dei caratteri tramite porta seriale. Riguardo alla chiavetta *ZigKey_trenino* verranno mostrati i file *inipict32.h* e *initpic32.c* - contenenti le inizializzazioni dei vari oscillatori, *ADC*, etc - e il file *main.c*, contenente il codice per la ricezione dei pacchetti, per la misura di corrente e tensione, la stima della velocità e l'algoritmo di controllo vero e proprio.
In seconda battuta verrà mostrato il codice Visual Basic con cui si è creata l'interfaccia grafica per la comunicazione da *PC*.

Chiavetta ZigKey: file main.c

```
********************************************************************************************
* FileName:                main.c
********************************************************************************************
************************************ HEADERS ***********************************************
#include "include\Common\Console.h"
#include "include\Transceivers\Transceivers.h"
#include "include\Common\SymbolTime.h"
#include "include\WirelessProtocols\MCHP_API.h"
#include "include\Common\LCDBlocking.h"
#include "include\common\timer.h"
#include <plib.h>
#include "SRAMDriver.h"

extern unsigned char RTDMRxBuffer[];
extern struct {
                        unsigned MessageReceived  :          1;
                        unsigned TransmitNow      :          1;
                        unsigned unused :                14;
                } RTDMFlags;
extern int iniBuffer;
extern int finBuffer;

extern char data_ready;
```

```
************************************** VARIABLES ***********************************************
#define LIGHT   0x01
#define SWITCH  0x02
*************************************************************************************************
#if ADDITIONAL_NODE_ID_SIZE > 0
   BYTE AdditionalNodeID[ADDITIONAL_NODE_ID_SIZE] = {SWITCH};
#endif


*************************************************************************************************
// The variable myChannel defines the channel that the P2P connection is operate on. This variable //will be only effective if energy scan  (ENABLE_ED_SCAN) is not turned on. Once the energy scan //is turned on, the operating channel will be one of the channels available with  least amount of //energy (or noise).
*************************************************************************************************
BYTE myChannel = 25;
*************************************************************************************************
//#pragma config FPLLMUL = MUL_20, FPLLIDIV = DIV_2, FPLLODIV = DIV_1, FWDTEN = OFF
//#pragma config POSCMOD = HS, FNOSC = PRIPLL, FPBDIV = DIV_1


int main(void)
{
   BYTE i;
          int fin;
          //SYSTEMConfig(SYS_FREQ, SYS_CFG_WAIT_STATES | SYS_CFG_PCACHE);


   *********************************************************************************************
   // Initialize the system
   *********************************************************************************************
   BoardInit();
   ConsoleInit();
    LED_1 = 1;
          delay_ms(100);
    LED_1=0;
    *********************************************************************************************
   // Initialize Microchip proprietary protocol. Which protocol to use depends on the configuration
  // in ConfigApp.h
   *********************************************************************************************
   MiApp_ProtocolInit();
   // Set default channel
   if( MiApp_SetChannel(myChannel) == FALSE )
```

```
{
   PrintDec(myChannel);
   myprintf(" is not supported in current condition.\r\n");
   return 0;
}
else
{
   myprintf("\r\nSelection of channel ok");
}
*********************************************************************************************
//Function MiApp_ConnectionMode defines the connection mode. The possible connection
//modes are:
//ENABLE_ALL_CONN: Enable all kinds of connection
//ENABLE_PREV_CONN: Only allow connection already exists in connection table
//ENABL_ACTIVE_SCAN_RSP: Allow response to Active scan
//DISABLE_ALL_CONN: Disable all connections.
*********************************************************************************************
MiApp_ConnectionMode(ENABLE_ALL_CONN);
   myprintf("\r\nCoonecting Peer...");
*********************************************************************************************
//Function MiApp_EstablishConnection try to establish a new connection with peer device.
//The first parameter is the index to the active scan result, which is acquired by discovery
//process (active scan). If the value of the index is 0xFF, try to establish a connection with any
//peer.
//The second parameter is the mode to establish connection, either direct or indirect. Direct
//mode means connection within the radio range;
//Indirect mode means connection may or may not in the radio range.
*********************************************************************************************
while( (i = MiApp_EstablishConnection(0xFF, CONN_MODE_DIRECT)) == 0xFF )
        {           delay_ms(50);
                        LED_1 ^= 1;
        }
        LED_1 = 1;

ConsolePut('T');
ConsolePut('r');
ConsolePut('e');
ConsolePut('n');
ConsolePut('o');
```

```
  ConsolePut(' ');
  ConsolePut('c');
  ConsolePut('o');
  ConsolePut('n');
  ConsolePut('n');
  ConsolePut('e');
  ConsolePut('s');
  ConsolePut('s');
  ConsolePut('o');
  ConsolePut(' ');

  while(1)
  {
    delay_ms(1500);
    if (RTDMRxBuffer[0]=='a')
                {
                MiApp_FlushTx();
                MiApp_WriteData(RTDMRxBuffer[1]);
                RTDMRxBuffer[0]='0';
                MiApp_BroadcastPacket(FALSE);
                #if 0
                        if(MiApp_UnicastConnection(0, TRUE) == FALSE)
                                {
                                for (i=0;i<10;i++)
                                  {
                                   LED_1=0;
                                   delay_ms(50);
                                   LED_1=1;
                                  delay_ms(50);
                                   }
                                }
                #endif
                }
  }
}
*******************************************************************************************
```

Chiavetta ZigKey: file console.c

```
********************************************************************************************
* FileName:                 console.c
********************************************************************************************
#include "include\Common\Console.h"
#include "SystemProfile.h"
#include "include\Compiler.h"
#include "GenericTypeDefs.h"

#if defined(ENABLE_CONSOLE)
#define BAUD_RATE 57600

********************************************************************************************
// Received data is stored in array RTDMRxBuffer
char RTDMRxBuffer[RTDM_RXBUFFERSIZE];
int iniBuffer=0;
int finBuffer=0;
char * RTDMRxBufferIndex = RTDMRxBuffer;



//Structure enclosing the RTDM flags
struct {
        unsigned MessageReceived  :     1;
        unsigned TransmitNow      :     1;
        unsigned unused :                       14;
        }       RTDMFlags;

char data_ready=0;

************************************* VARIABLES ***********************************************
ROM unsigned char CharacterArray[]={'0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F'};

************************************* FUNCTIONS ***********************************************
********************************************************************************************
//Function:         void ConsoleInit(void)
//PreCondition:     none
//Input:        none
//Output:           none
```

```
//Side Effects:         UART2 is configured
//Overview:             This function will configure the UART for use at  8 bits, 1 stop, no flowcontrol
//mode
*****************************************************************************************
void ConsoleInit(void)
{
        #define config1   UART_EN | UART_IDLE_CON | UART_RX_TX | UART_DIS_WAKE |      UART_DIS_LOOPBACK
| UART_DIS_ABAUD | UART_NO_PAR_8BIT | UART_1STOPBIT |        UART_IRDA_DIS    |   UART_DIS_BCLK_CTS_RTS|
UART_NORMAL_RX | UART_BRGH_SIXTEEN


        // define setup Configuration 2 for OpenUARTx
                //IrDA encoded UxTX idle state is '0'
                //Enable UxRX pin
                //Enable UxTX pin
                //Interrupt on transfer of every character to TSR
                //Interrupt on every char received
                //Disable 9-bit address detect
                //Rx Buffer Over run status bit clear
        #define config2   UART_TX_PIN_LOW        |        UART_RX_ENABLE        |        UART_TX_ENABLE        |
        UART_INT_TX|UART_INT_RX_CHAR|UART_ADR_DETECT_DIS |UART_RX_OVERRUN_CLEAR
        unsigned int pbFreq;
        //Peripheral Bus Frequency = System Clock / PB Divider
        pbFreq = (DWORD)CLOCK_FREQ / (1 << mOSCGetPBDIV() );

        //Open UART2 with config1 and config2
                OpenUART2( config1, config2, pbFreq/16/BAUD_RATE-1);     // calculate actual BAUD
        generate value.

        //Configure UART2 RX Interrupt with priority 2
        //ConfigIntUART2(UART_INT_PR2 | UART_RX_INT_EN|UART_TX_INT_EN);
        ConfigIntUART2(UART_INT_PR2 | UART_RX_INT_EN);
        //Must enable glocal interrupts - in this case, we are using multi-vector mode
        // INTEnableSystemMultiVectoredInt();
        RTDMFlags.MessageReceived = 0;
}

*****************************************************************************************
//Function:         void ConsolePut(BYTE c)
//PreCondition:     none
//Input:            c: character to be printed
```

```
//Output:               none
//Side Effects:         c is printed to the console
//Overview:             This function will print the inputed character
********************************************************************************************
void ConsolePut(BYTE c)
{
        while(U2STAbits.TRMT == 0);
        U2TXREG = c;
}

********************************************************************************************
//Function:         void PrintChar(BYTE toPrint)
//PreCondition:     none
//Input:        toPrint - character to be printed
// Output:              none
//Side Effects:         toPrint is printed to the console
//Overview:             This function will print the inputed BYTE to the console in hexidecimal form
********************************************************************************************
void PrintChar(BYTE toPrint)
{
        BYTE PRINT_VAR;
        PRINT_VAR = toPrint;
        toPrint = (toPrint>>4)&0x0F;
        ConsolePut(CharacterArray[toPrint]);
        toPrint = (PRINT_VAR)&0x0F;
        ConsolePut(CharacterArray[toPrint]);
        return;
}

********************************************************************************************
//Function:         void PrintDec(BYTE toPrint)
//PreCondition:     none
//Input:        toPrint - character to be printed. Range is 0-99
//Output:           none
//Side Effects:         toPrint is printed to the console in decimal
//Overview:             This function will print the inputed BYTE to the console in decimal form
********************************************************************************************
void PrintDec(BYTE toPrint)
{
```

```
	ConsolePut(CharacterArray[toPrint/10]);
	ConsolePut(CharacterArray[toPrint%10]);
}
void PrintHex(BYTE toPrint)
{
	ConsolePut('0');
	ConsolePut('x');
	ConsolePut(CharacterArray[toPrint/16]);
	ConsolePut(CharacterArray[toPrint%16]);
}

BYTE posBuffer;
#define STATUS_IDLE	0
#define STATUS_CMD	1
#define STATUS_LEN	2
#define STATUS_DATI 3
int stato_rx=STATUS_IDLE;
char lenBuffer;

void __ISR(_UART2_VECTOR, ipl2) IntUart2Handler(void)
{
	char c;
	//Is this an RX interrupt?
	if(IFS1bits.U2RXIF)
	{
		c=U2RXREG;
		switch(stato_rx)
		{
			case STATUS_IDLE:
				if (c=='?')
				{
					if (data_ready==1)
					{
						U2TXREG='1';
					}
					else
					{
						U2TXREG='0';
					}
```

```
                }
                if (c=='*')
                {
                        posBuffer=0;
                        stato_rx=STATUS_CMD;
                }
                break;
        case STATUS_CMD:
                RTDMRxBufferIndex[posBuffer++] = c;
                stato_rx=STATUS_LEN;
                break;
        case STATUS_LEN:
                if (c!=0)
                {
                        RTDMRxBufferIndex[posBuffer++] = c;
                        lenBuffer=c;
                        stato_rx=STATUS_DATI;
                }
                else
                {
                        RTDMFlags.MessageReceived = 1;
                        finBuffer=posBuffer;
                        stato_rx=STATUS_IDLE;
                }
                break;
        case STATUS_DATI:
                RTDMRxBufferIndex[posBuffer++] = c;
                if (posBuffer>=lenBuffer+2)
                {
                        RTDMFlags.MessageReceived = 1;
                        finBuffer=posBuffer;
                        stato_rx=STATUS_IDLE;
                }
                else
                        stato_rx=STATUS_IDLE;
                }
        IFS1bits.U2RXIF = 0;
}
// We don't care about TX interrupt
```

```
        if (IFS1bits.U2TXIF)
        {
                IFS1bits.U2TXIF = 0;
        }
}
#endif  //ENABLE_CONSOLE
*********************************************************************************************
```

Chiavetta ZigKey_trenino: file main.c

```
*********************************************************************************************
* FileName:                main.c
*********************************************************************************************
****************************************** HEADERS ***************************************
#include "include\Common\Console.h"
#include "include\Transceivers\Transceivers.h"
#include "include\Common\SymbolTime.h"
#include "include\WirelessProtocols\MCHP_API.h"
#include "include\Common\LCDBlocking.h"
#include "include\common\timer.h"
#include <plib.h>
#include "SRAMDriver.h"
#include <p32xxxx.h>
#include "initpic32.h"
#include "routinepic32.h"

************************************* ALCUNE VARIABILI GLOBALI ********************************
#define NBB 14
#define mult(c,d) (int)(((long)c*d)>>NBB)
#define mult1(a,b) (unsigned short)(((unsigned int)a*b))

long int contr=0x0;
int rData=0x0;
int command=0x0;
int TransMask=0x0;
int Type=0x0;
unsigned int Direction=0x1;
int LuceAnt=0x0;
int LucePost=0x0;
```

```
int LuceSfum=0x0;

int RifCorrente=0x0;
int RifCorrente1=0x0;
int ErroreCorrente=0x0;

long int duty=0x0;
unsigned long int duty1=0x0;
unsigned short SpeedRef=0x0;
unsigned short RifVelocita=0x0;
int RifVelocita1=0x0;
int ErroreVelocita=0x0;
int Err_Vel_Int=0x0;
int Err_Vel_Old=0x0;
int integrV=0x0;

unsigned int temp1=0x0;
unsigned int temp2=0x0;

int count=0x0;

**************************VARIABILI GLOBALI REGOLATORE PI VELOCITA*****************************
#define KpVdig 4000
#define KiVdig 40
******************* VARIABILI GLOBALI REGOLATORE PI CORRENTE ALLA GUARDABASSI**********************
long int U=0x0;
long int Up=0x0;
long int Ui=0x0;

#define b 12665
#define bo 3719
#define KpIdig 18301

*********************************************************************************************
extern unsigned char RTDMRxBuffer[];
extern struct {
                        unsigned MessageReceived  :         1;
                        unsigned TransmitNow      :         1;
```

```
                unsigned unused :           14;
        }       RTDMFlags;
extern int iniBuffer;
extern int finBuffer;
extern char data_ready;

************************************** VARIABLES ***************************************
#define LIGHT   0x01
#define SWITCH  0x02
*****************************************************************************************

#if ADDITIONAL_NODE_ID_SIZE > 0
    BYTE AdditionalNodeID[ADDITIONAL_NODE_ID_SIZE] = {SWITCH};
#endif

BYTE myChannel = 25;

int main(void)
{
    BYTE i;

    *****************************************************************************************
    //Initialize the system
    *****************************************************************************************
    BoardInit();
    ConsoleInit();
    LED_1 = 1;
    delay_ms(100);
    LED_1=0;
    MiApp_ProtocolInit();

    //Set default channel
    if( MiApp_SetChannel(myChannel) == FALSE )
    {
        PrintDec(myChannel);
        myprintf(" is not supported in current condition.\r\n");
        return 0;
    }
    else
```

```
{
        myprintf("\r\nSelection of channel ok");
}

MiApp_ConnectionMode(ENABLE_ALL_CONN);
myprintf("\r\nCoonecting Peer...");

while( (i = MiApp_EstablishConnection(0xFF, CONN_MODE_DIRECT)) == 0xFF )
{       delay_ms(50);
        LED_1 ^= 1;
}
LED_1 = 0;

*****************************************************************************

delay_ms(500);

*****************************INIZIALIZZAZIONE*****************************************
initIO();
initADC();
initOCeTMR();

IFS0CLR = 0x00010000;
IEC0SET = 0x00010000;
IPC4SET = 0x0000000C;

T2CONSET=0x8000;
T4CONSET=0x8000;

TMR2=0;
TMR4=200;

DisableWDT();
OC1CONSET = 0x8000;
OC2CONSET = 0x8000;
OC3CONSET = 0x8000;
mAD1SetIntPriority(1);
mAD1SetIntSubPriority(1);
mAD1IntEnable(1);
```

```
INTEnableSystemMultiVectoredInt();
ADIF=0;
EnableADC10();
*****************************************************************************
while(1)
{
        ***************************RICEZIONE COMANDO**********************************
        if (TRUE == MiApp_MessageAvailable())
        {
                rData = rxMessage.Payload[0];
                MiApp_DiscardMessage();

                ********SPACCHETTAMENTO CON MASCHERE DI TRANSIZIONI**********************

                TransMask=0x000000e0;
                command= rData & TransMask;

                if (command==0x00000040)
                {   TransMask=0x00000010;
                    temp1= rData & TransMask;
                    TransMask=0x0000000f;
                    temp2=rData & TransMask;    }

                if (command==0x00000060)
                {   TransMask=0x00000007;
                    LuceSfum=rData & TransMask;
                   OC3RS=LuceSfum*57;           }

                if (command==0x00000020)
                {   TransMask=0x0000001c;
                    Type=rData & TransMask;
                    if (Type==0x00000004)
                    {    TransMask=0x00000002;
                         LuceAnt=rData & TransMask;
                         TransMask=0x00000001;
                         LucePost=rData & TransMask;
                         if (LuceAnt==0x00000002)
                         {   LED_1=1;PORTDSET=0x10;    }
                         else
```

```
                                    {   LED_1=0;PORTDCLR=0x10;   }
                                    if (LucePost==0x00000001)
                                    {   LED_2=1;PORTDSET=0x20;    }
                                    else
                                    {   LED_2=0;PORTDCLR=0x20;   }
                                }
                            }
                            Direction=temp1;
                            SpeedRef=(temp2<<10);

                    }    //fine if message available
            }             //fine while(1)
}                   //fine main

**********************************INTERRUPT SERVICE ROUTINE************************************
void __ISR(_TIMER_4_VECTOR, ipl3) _TMR4Interrupt( void)
            {
            if (IFS0=0x10)
                    {
                    count++;
                    if (count==6)
                            {
                            count=0;
                            AD1CON1SET=0x2;
                            AD1CON1SET=0x4;
                            }
                    IFS0CLR = 0x00010000;
                    }
            }

void __ISR(_ADC_VECTOR, ipl1) _ADCInterrupt( void)
            {
            unsigned short MisuraCorrente=0x0;
            int MisuraCorrente1=0x0;
            unsigned short MisuraVdcbus=0x0;
            unsigned short Vmotore=0x0;
            unsigned short RxI=0x0;
            unsigned short StimaFEM=0x0;
            unsigned short MisuraVelocita=0x0;
```

```
int MisuraVelocita1=0x0;
if(ADIE && ADIF)
        {
        MisuraCorrente = ADC1BUF0;
        MisuraVdcbus = ADC1BUF1;
        MisuraCorrente=(MisuraCorrente>>2);
        MisuraVdcbus=MisuraVdcbus>>2;

        **********STIMATORE DI VELOCITA'**********
        duty=((contr*15)/400);
        if (duty>=0)                    {duty1=duty;} else {duty1=-duty;}
        Vmotore=(mult1(duty1,MisuraVdcbus))/15;
        RxI=(MisuraCorrente<<1);
        StimaFEM=Vmotore-RxI;
        MisuraVelocita=StimaFEM;
        MisuraVelocita=((MisuraVelocita<<NBB)/202);
        if (Direction==0x00000010)    {MisuraVelocita1=-MisuraVelocita;}
        else                          {MisuraVelocita1=MisuraVelocita;}
        **********
        RifVelocita=SpeedRef;
        if (Direction==0x00000010)    {RifVelocita1=-RifVelocita;}
        else                          {RifVelocita1=RifVelocita;}

        ErroreVelocita=RifVelocita1-MisuraVelocita1;

        **********REGOLATORE DI VELOCITA'**********
        Err_Vel_Int=ErroreVelocita+Err_Vel_Old;
        integrV+=mult(KiVdig,Err_Vel_Int);
        if (integrV>16400)              {integrV=16400;}
        if (integrV<(-16400))           {integrV=(-16400);}
        RifCorrente=((int)((mult(KpVdig,ErroreVelocita)+integrV)*35))>>NBB;
        Err_Vel_Old=ErroreVelocita;

        if (RifCorrente > 35)         {RifCorrente1=35;}
        else                          if (RifCorrente < (-35))   {RifCorrente1=(-35);}
                                      else                       {RifCorrente1=RifCorrente;}

        **********
        if (Direction==0x00000010)  {MisuraCorrente1=-MisuraCorrente;}
```

```
                else                            {MisuraCorrente1=MisuraCorrente;}
                MisuraCorrente1=MisuraCorrente1*468;
                RifCorrente1=RifCorrente1*468;
                ErroreCorrente=RifCorrente1-MisuraCorrente1;

                ***********REGOLATORE DI CORRENTE ALLA GUARDABASSI*****************************
                Up=mult(KpIdig,ErroreCorrente);
                U=((long int)((Ui+Up)*400))>>NBB;
                if (U > 360)                    {contr=360;}
                else                            if (U < (-360))         {contr=(-360);}
                                                else                    {contr=U;}
                Ui=((mult(b,Ui))+(mult(bo,contr)));

                ********************IMPOSTAZIONE DIREZIONE DUTY*********************************
                if (temp2==0)
                {   OC1R=0x00C8;
                    OC1RS=0x00CB;
                    OC2R=0x00C8;
                    OC2RS=0x00CB;
                    integrV=0;
                }
                else {                          if (contr>=0){
                                                OC1R = (400-contr)>>1;
                                                OC1RS =(406+contr)>>1;
                                                OC2R=0x00C8;
                                                OC2RS=0x00CB;
                      }
                if (contr<0)
                {
                OC1R=0x00C8;
                OC1RS=0x00CB;
                OC2R = (400+contr)>>1;
                OC2RS =(406-contr)>>1;
                }
        }
        ADIF=0;
        }       //end if
}               //chiude ADCInterrupt
*****************************************************************************************
```

Chiavetta ZigKey_trenino: file initpic32.c

```
*********************************************************************************************
* FileName:                initpic32.c
*********************************************************************************************
#include <p32xxxx.h>
#include "initpic32.h"
*******************************INIZIALIZZAZIONE OSCILLATORE E PLL*********************************

#pragma config FPLLMUL = MUL_20, FPLLIDIV = DIV_2, FPLLODIV = DIV_2, FWDTEN = OFF
#pragma config POSCMOD = HS, FNOSC = FRCPLL, FPBDIV = DIV_2, WDTPS = PS8192

************************************INIZIALIZZAZIONE PORTE *********************************
void initIO()
        {
        TRISD=0b1100110000000;
        PORTD=0b0001000000000;
        TRISE=0b11100111;
        PORTE=0b00000000;
        TRISB=0x7fff;
        PORTB=0x0000;
        TRISF=0b10010111;
        PORTF=0b00000000;
        TRISG=0b0111;
        PORTG=0b1000;
        }
************************************INIZIALIZZAZIONE ADC ***********************************
void initADC()
        {
        AD1PCFG=0x0000fff3;
        AD1CHS=0x00020000;
        AD1CON1=0x000000f0;
        AD1CON2=0x00000404;
        AD1CON3=0x0000F101;
        AD1CSSL=0x0000000c;
        }


*******************************INIZIALIZZAZIONE OUTPUT COMPARE e TIMERS*************************
void initOCeTMR()
        {
        OC1CON = 0x0000;
```

```
        OC1R = 0x00C8;
        OC1RS = 0x00CB;
        OC1CON = 0x0005;
        OC2CON = 0x0000;
        OC2R = 0x00C8;
        OC2RS = 0x00CB;
        OC2CON = 0x0005;
        OC3CON = 0x0000;
        OC3R = 0x0000;
        OC3RS = 0x0000;
        OC3CON = 0x0006;
        T2CON=0x00000000;
        TMR2=0;
        PR2 = 0x018F;
        T4CON=0x00000000;
        TMR4=0;
        PR4 = 0x018F;
        }
*********************************************************************************************
```

Chiavetta ZigKey_trenino: file initpic32.h

```
*********************************************************************************************
* FileName:                initpic32.h
*********************************************************************************************
*******************************DEFINIZIONE PROTOTIPI INIZIALIZZAZIONE*****************************
#ifndef __INIT_H__
#define __INIT_H__
extern void initADC(void); /*inizializzazione ADC*/
extern void initIO(void); /*inizializzazione I/O*/
extern void initOCeTMR(void); /*inizializzazione OutputCompare e Timers*/
#endif
*********************************************************************************************
```

Codice dell'interfaccia grafica in Visual Basic

```
Imports System
Imports System.ComponentModel Imports System.Threading
Imports System.IO.Ports
Public Class frmMain
```

```
    Dim myPort As Array
    Delegate Sub SetTextCallback(ByVal [text] As String)
    Private Sub frmMain_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
        myPort = IO.Ports.SerialPort.GetPortNames() 'Get all com ports available

        For i = 0 To UBound(myPort)
            cmbPort.Items.Add(myPort(i))
        Next
        cmbPort.Text = cmbPort.Items.Item(0)

        btnDisconnect.Enabled = False

    End Sub
    Private Sub btnConnect_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
btnConnect.Click
        SerialPort1.PortName = cmbPort.Text
        SerialPort1.BaudRate = 57600

        SerialPort1.Parity = IO.Ports.Parity.None
        SerialPort1.StopBits = IO.Ports.StopBits.One
        SerialPort1.DataBits = 8
        SerialPort1.Open()

        btnConnect.Enabled = False
        btnDisconnect.Enabled = True

    End Sub

    Private Sub btnDisconnect_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
btnDisconnect.Click
        SerialPort1.Close()

        btnConnect.Enabled = True
        btnDisconnect.Enabled = False
    End Sub

    Private Sub SerialPort1_DataReceived(ByVal sender As Object, ByVal e As
System.IO.Ports.SerialDataReceivedEventArgs) Handles SerialPort1.DataReceived
        ReceivedText(SerialPort1.ReadExisting())
    End Sub
    Private Sub ReceivedText(ByVal [text] As String)
        If Me.rtbReceived.InvokeRequired Then
            Dim x As New SetTextCallback(AddressOf ReceivedText)
            Me.Invoke(x, New Object() {(text)})
        Else
            Me.rtbReceived.Text &= [text]
        End If
    End Sub

    Private Sub cmbPort_SelectedIndexChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
cmbPort.SelectedIndexChanged
        If SerialPort1.IsOpen = False Then
            SerialPort1.PortName = cmbPort.Text
        Else
            MsgBox("Valid only if port is Closed", vbCritical)
        End If
    End Sub
```

```
Private Sub TrenoOnBtn_Click(sender As System.Object, e As System.EventArgs) Handles TrenoOnBtn.Click
    Dim treno_on As String = "*a("            'treno ON: 00101000
    If SerialPort1.IsOpen = True Then
        SerialPort1.Write(treno_on & vbCr)
    End If
End Sub

Private Sub TrenoOffBtn_Click(sender As System.Object, e As System.EventArgs) Handles TrenoOffBtn.Click
    Dim treno_off As String = "*a("
    If SerialPort1.IsOpen = True Then          'treno OFF: 00101000
        SerialPort1.Write(treno_off & vbCr)
    End If
End Sub

Private Sub trbVelocità_Scroll(sender As System.Object, e As System.EventArgs) Handles trbVelocità.Scroll
    Dim ascii_1 As String = "*a_"             'velocità e direzione: 010DSSSS (D=0 avanti, D=1 indietro)
    Dim ascii_2 As String = "*a^"
    Dim ascii_3 As String = "*a]"
    Dim ascii_4 As String = "*a\"
    Dim ascii_5 As String = "*a["
    Dim ascii_6 As String = "*aZ"
    Dim ascii_7 As String = "*aY"
    Dim ascii_8 As String = "*aX"
    Dim ascii_9 As String = "*aW"
    Dim ascii_10 As String = "*aV"
    Dim ascii_11 As String = "*aU"
    Dim ascii_12 As String = "*aT"
    Dim ascii_13 As String = "*aS"
    Dim ascii_14 As String = "*aR"
    Dim ascii_15 As String = "*aQ"
    Dim ascii_16 As String = "*aP"
    Dim ascii_17 As String = "*aA"
    Dim ascii_18 As String = "*aB"
    Dim ascii_19 As String = "*aC"
    Dim ascii_20 As String = "*aD"
    Dim ascii_21 As String = "*aE"
    Dim ascii_22 As String = "*aF"
    Dim ascii_23 As String = "*aG"
    Dim ascii_24 As String = "*aH"
    Dim ascii_25 As String = "*aI"
    Dim ascii_26 As String = "*aJ"
    Dim ascii_27 As String = "*aK"
    Dim ascii_28 As String = "*aL"
    Dim ascii_29 As String = "*aM"
    Dim ascii_30 As String = "*aN"
    Dim ascii_31 As String = "*aO"

    If SerialPort1.IsOpen = True Then
        Select Case trbVelocità.Value
            Case 0
                SerialPort1.Write(ascii_1 & vbCr)
            Case 1
                SerialPort1.Write(ascii_2 & vbCr)
            Case 2
                SerialPort1.Write(ascii_3 & vbCr)
            Case 3
```

```
            SerialPort1.Write(ascii_4 & vbCr)
        Case 4
            SerialPort1.Write(ascii_5 & vbCr)
        Case 5
            SerialPort1.Write(ascii_6 & vbCr)
        Case 6
            SerialPort1.Write(ascii_7 & vbCr)
        Case 7
            SerialPort1.Write(ascii_8 & vbCr)
        Case 8
            SerialPort1.Write(ascii_9 & vbCr)
        Case 9
            SerialPort1.Write(ascii_10 & vbCr)
        Case 10
            SerialPort1.Write(ascii_11 & vbCr)
        Case 11
            SerialPort1.Write(ascii_12 & vbCr)
        Case 12
            SerialPort1.Write(ascii_13 & vbCr)
        Case 13
            SerialPort1.Write(ascii_14 & vbCr)
        Case 14
            SerialPort1.Write(ascii_15 & vbCr)
        Case 15
            SerialPort1.Write(ascii_16 & vbCr)
        Case 16
            SerialPort1.Write(ascii_17 & vbCr)
        Case 17
            SerialPort1.Write(ascii_18 & vbCr)
        Case 18
            SerialPort1.Write(ascii_19 & vbCr)
        Case 19
            SerialPort1.Write(ascii_20 & vbCr)
        Case 20
            SerialPort1.Write(ascii_21 & vbCr)
        Case 21
            SerialPort1.Write(ascii_22 & vbCr)
        Case 22
            SerialPort1.Write(ascii_23 & vbCr)
        Case 23
            SerialPort1.Write(ascii_24 & vbCr)
        Case 24
            SerialPort1.Write(ascii_25 & vbCr)
        Case 25
            SerialPort1.Write(ascii_26 & vbCr)
        Case 26
            SerialPort1.Write(ascii_27 & vbCr)
        Case 27
            SerialPort1.Write(ascii_28 & vbCr)
        Case 28
            SerialPort1.Write(ascii_29 & vbCr)
        Case 29
            SerialPort1.Write(ascii_30 & vbCr)
        Case 30
            SerialPort1.Write(ascii_31 & vbCr)
    End Select
End If
```

```
  End Sub

  Private Sub LuciOnRB_CheckedChanged(sender As System.Object, e As System.EventArgs) Handles
LuciOnRB.CheckedChanged
    Dim luci_on As String = "*a'"              'luci ON: 00100111
    If SerialPort1.IsOpen = True Then
      SerialPort1.Write(luci_on & vbCr)
    End If
  End Sub

  Private Sub LuciOffRB_CheckedChanged(sender As System.Object, e As System.EventArgs) Handles
LuciOffRB.CheckedChanged
    Dim luci_off As String = "*a$"             'luci OFF: 00100100
    If SerialPort1.IsOpen = True Then
      SerialPort1.Write(luci_off & vbCr)
    End If
  End Sub

  Private Sub LuciAntOnRB_CheckedChanged(sender As System.Object, e As System.EventArgs) Handles
LuciAntOnRB.CheckedChanged
    Dim luci_ant_on As String = "*a&"          'luci anteriori ON:: 00100110
    If SerialPort1.IsOpen = True Then
      SerialPort1.Write(luci_ant_on & vbCr)
    End If
  End Sub

  Private Sub LuciPostOnRB_CheckedChanged(sender As System.Object, e As System.EventArgs) Handles
LuciPostOnRB.CheckedChanged
    Dim luci_post_on As String = "*a%"         'luci posteriori ON: : 00100101
    If SerialPort1.IsOpen = True Then
      SerialPort1.Write(luci_post_on & vbCr)
    End If
  End Sub

  Private Sub BuzzerBtn_Click(sender As System.Object, e As System.EventArgs) Handles BuzzerBtn.Click
    Dim buzzer As String = "*a#"               'buzzer: 00100011
    If SerialPort1.IsOpen = True Then
      SerialPort1.Write(buzzer & vbCr)
    End If
  End Sub

  Private Sub TrackBar1_Scroll(sender As System.Object, e As System.EventArgs) Handles TrackBar1.Scroll
    Dim luci_sfum_1 As String = "*aa"          'luci sfumate: 01100III
    Dim luci_sfum_2 As String = "*ab"          'manca 01100000 in quanto equivale a luci OFF
    Dim luci_sfum_3 As String = "*ac"
    Dim luci_sfum_4 As String = "*ad"
    Dim luci_sfum_5 As String = "*ae"
    Dim luci_sfum_6 As String = "*af"
    Dim luci_sfum_7 As String = "*ag"
    If SerialPort1.IsOpen = True Then
      If LuciOnRB.Checked = True Then
        Select Case trbVelocità.Value
          Case 0
            SerialPort1.Write(luci_sfum_1 & vbCr)
          Case 1
            SerialPort1.Write(luci_sfum_2 & vbCr)
```

```
            Case 2
                SerialPort1.Write(luci_sfum_3 & vbCr)
            Case 3
                SerialPort1.Write(luci_sfum_4 & vbCr)
            Case 4
                SerialPort1.Write(luci_sfum_5 & vbCr)
            Case 5
                SerialPort1.Write(luci_sfum_6 & vbCr)
            Case 6
                SerialPort1.Write(luci_sfum_7 & vbCr)
        End Select
    End If
  End If
End Sub


End Class
```