SAPIENZA
UNIVERSITÀ DI ROMA

Implementazione di un controllore PID con AVR per il controllo di un sistema Multi-Motore

Facoltà di Ingegneria dell'informazione, informatica e statistica
Laurea in Ingegneria Informatica e Automatica

Georgi Todorov Dimitrov
Matricola 1890039

Relatore
Prof. Giorgio Grisetti

Correlatore

Anno Accademico 2021/2022

Implementazione di un controllore PID con AVR per il controllo di un sistema Multi-Motore

Tesi di Laurea . Sapienza Università di Roma

Questa tesi è stata composta con LaTeX e la classe Sapthesis.

Email dell'autore: dimitrov.1890039@studenti.uniroma1.it

Dedicato ai folli ...

Indice

Capitolo 1

Introduzione

In questa trattazione verrà affrontato il problema del controllo di un motore DC attraverso la programmazione di un micro-controllore AVR. Tale controllore sarà utilizzato insieme a un protocollo I2C per il controllo simultaneo di più motori.

Tale sistema è alla base di molte applicazioni della robotica nel campo industriale. In questa trattazione si fornisce un controllo di un sistema multi-motore semplificato, ma facilmente estendibile a casi più complessi. Basti pensare che in ogni sistema robotico composto da giunti e collegamenti, ogni giunto può essere rappresentato da un servo motore e che deve essere controllato in maniera indipendente.
Forniamo una panoramica del problema affrontato attraverso il seguente schema del circuito elettronico:

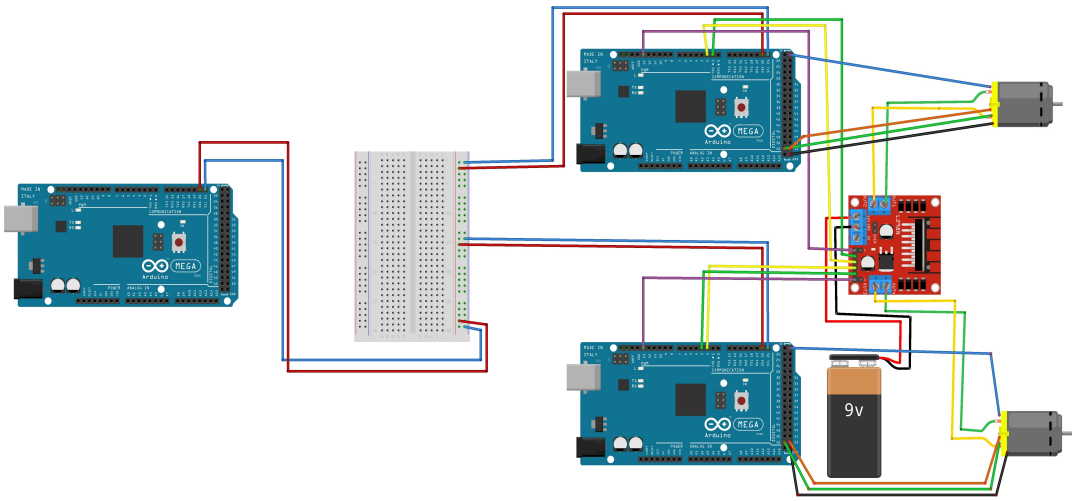


Figura 1.1. Sistema Multi Motore

Come si può osservare, si tratta di un applicazione su un sistema distribuito formato da più macchine, nelle quali in ciascuna di esse vi è un processo che si deve sincronizzare con gli altri. In questa trattazione non verranno forniti dettagli sulla comunicazione I2C attraverso l'interfaccia TWI, per maggiori dettagli si può visitare la repository del progetto. Verranno trattati i seguenti argomenti:

- nel **Capitolo 2** verranno affrontati singolarmente gli strumenti necessari, sia software che hardware, orientati alla costruzione di un controllore di un motore.
- nel **Capitolo 3** verrà affrontato come assemblare questi strumenti nel circuito elettronico e come costruire delle librerie software per la realizzazione del controllore.
- nel **Capitolo 4** verrà fornito un esempio di programma che si può caricare sull'AVR per controllare un semplice Servo Motore.
- nel **Capitolo 5** verranno tratte le conclusioni del progetto.

Tutto il codice mostrato in questo documento si può trovare nel seguente repository di cui amministratori sono Georgi Dimitrov e Sara Attiani: `https://github.com/GeoDimi99/AVR-Multi-Motor-Control.git`

Capitolo 2

Strumenti e documentazione di base

2.1 ATmega2560

L'ATmega2560 è un microcontrollore low-power CMOS a 8-bit basato sull'architettura RISC avanzata AVR. L'ATmega2560 esegue le istruzioni in un single clock cycle permettendo di raggiungere un throughput di 1 MIPS per MHz. I processori AVR si basano su un architettura Harvard, ovvero vi è separazione tra la memoria contenente i dati e quella contenente le istruzioni. L'ATmega2560 possiede le seguenti caratteristiche: 86 linee I/O a scopo generale, 32 registri di lavoro a scopo generale collegati direttamente con l'ALU (Arithmetic Logic Unit), 6 Timer/Counters e un interfaccia Seriale 2-wire.

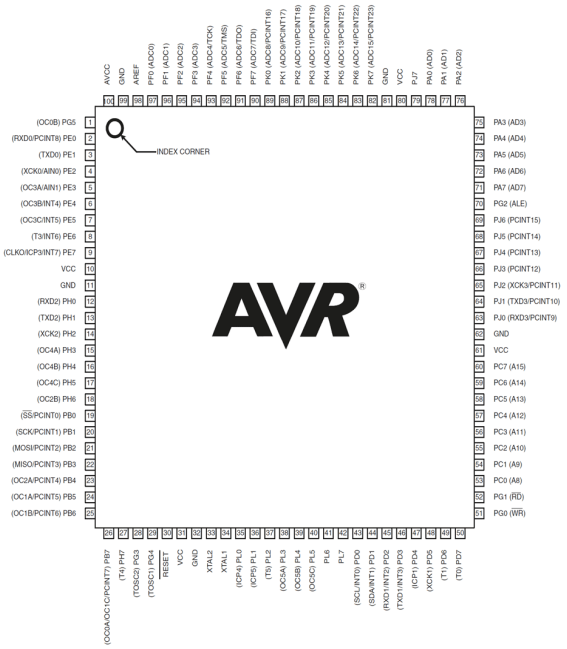


Figura 2.1. Schema equivalente di un I/O Pin

2.2 I/O-Ports

Tutte le porte AVR hanno delle funzionalità Read-Modify-Write quando sono usate come porte digitali I/O generiche. Questo significa che una porta può cambiare direzione, input o output, senza cambiare intenzionalmente la direzione delle altre porte. Le stessa considerazione vale anche quando si cambia il valore dell'unità, se la porta è configurato in output, o abilitare/disabilitare i resistori pull-up, se la porta è configurata in input. Tutti i pin delle porte hanno resistori pull-up selezionabili individualmente con una resistenza invariante della tensione di alimentazione; inoltre, tutti i pin I/O hanno dei diodi di protezione per il Vcc e il Ground.

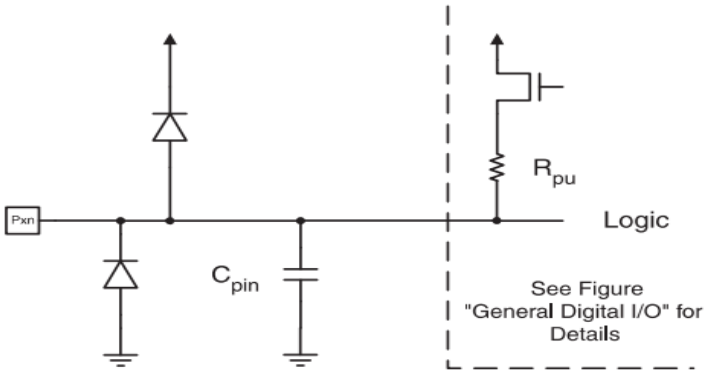


Figura 2.2. Schema equivalente di un I/O Pin

Ci sono tre aree di memoria I/O indirizzate per per ogni porta, in particolare un'area di memoria per ogni:

- **Data Register (PORTx)**: questo è un registro di lettura e scrittura e ha il compito settare in output un valore su una porta. In base a come sono settati i bit riferiti ai pin della porta, possiamo avere i seguenti comportamenti:
  - caso di bit settato a 1: se il pin è configurato come output si può leggere un valore di 5V, mentre se il pin è settato in input viene attivato il resistore pull-up;
  - caso di bit settato a 0: se il pin è configurato in output si può leggere il valore 0V;
- **Data Direction Register (DDRx)**: questo è un registro di lettura e scrittura. I bit contenuti in questo registro indicano se il corrispondente pin è in output (bit associato settato a 1) oppure in input (bit associato settato a 0).
- **Port Input Pins (PINx)**: questo è un registro di sola lettura e in base a come sono settati i bit riferiti ai pin della porta, possiamo avere i seguenti comportamenti:
  - se sul pin viene applicato un valore di 5V allora sul bit corrispondente viene letto un valore pari a 1;
  - se sul pin viene applicato un valore di 0V allora sul bit corrispondente viene letto un valore pari a 0;

2.3 Interrupts

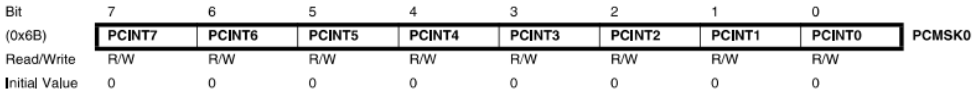
Un'**interrupt** è una notifica inviata alla CPU via hardware o eseguendo istruzioni software per segnalare un evento che richiede immediata attenzione da parte dell'AVR. L'hardware della CPU ha un input, detto **linea di richiesta dell'interruzione**, dal quale la CPU controlla lo stato dopo l'esecuzione di ogni riga di istruzione. Quando rileva il segnale di un controllore sulla linea di richiesta dell'interruzione, la CPU salva lo stato corrente e salta alla **interrupt-handler routine** (gestore delle interruzioni) che si trova in un indirizzo prefissato di memoria. Questa procedura determina le cause dell'interruzione, porta al termine l'elaborazione necessaria, ripristina lo stato e fa in modo che la CPU ricominci a lavorare dallo stato in cui era stata interrotta. Le varie cause di interruzioni e come vengono gestite, sono descritte in una tabella memorizzata all'interno della memoria, chiamata **Interrupt Vector**. In particolare ogni tipo di interruzione è associata ad un numero che corrisponde alla riga della tabella e tale riga contiene una funzione eseguita da una **Interrupt Service Routine (ISR)** che gestisce la relativa interruzione. Per sapere ogni riga della tabella a quale interruzione è associata si fa riferimento al datasheet del ATmega2560. Se il programmatore non abilita le sorgenti di interrupt, l'Interrupt Vector non viene utilizzato e lo spazio dedicato ad esso può essere occupato dal codice di un programma. La posizione dell'Interrupt Vector può essere controllata da un registro denominato **MCU Control Register**.

Le **External Interrupts** sono innescate dai pin INT7:0 o da ogni pin dei PCINT23:0. Possiamo osservare che, se abilitiamo le interrupts sono innescate anche se i pin INT7:0 o i pin PCINT23:0 sono configurate in output e questa caratteristica permette di generare un software interrupt.
Il Pin Change Interrupt PCI2 si innescherà se commuta ogni pin PCINT23:16 abilitato, il Pin Change Interrupt PCI1 si innescherà se commuta ogni pin PCINT15:8 abilitato e il Pin Change Interrupt PCI0 si innescherà se commuta ogni pin PCINT7:0. I registri PCMSK2, PCMSK1 e PCMSK0 controllano quale pin contribuisce al pin change interrupts. I pin change interrupts su PCINT23:0 sono rilevati in maniera asincrona; ciò implica che questi interrupts possono essere usati per svegliare delle parti che sono in sleep mode o in idle mode. Per poter abilitare l'interrupt da parte di uno di questi gruppi, va settato il relativo bit all'interno del registro **Pin Change Interrupt Control Register (PCICR)**:

Bit	7	6	5	4	3	2	1	0	
(0x68)	–	–	–	–	–	PCIE2	PCIE1	PCIE0	PCICR
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

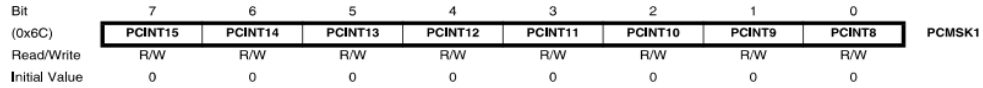
Figura 2.3. PCICR

Per settare quale pin PCINT23:0 può partecipare alla generazione di una pin change interrupt, bisogna settare il corrispondente bit a uno dei registri **Pin Change Mask Registers (PCMSK)** dei relativi gruppi:



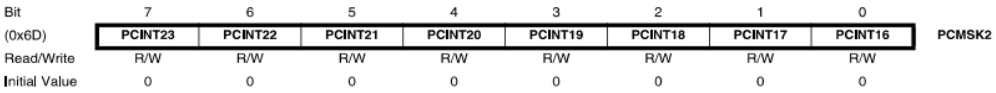
Bit	7	6	5	4	3	2	1	0	
(0x6B)	PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0	PCMSK0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 2.4. PCMSK0



Bit	7	6	5	4	3	2	1	0	
(0x6C)	PCINT15	PCINT14	PCINT13	PCINT12	PCINT11	PCINT10	PCINT9	PCINT8	PCMSK1
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 2.5. PCMSK1

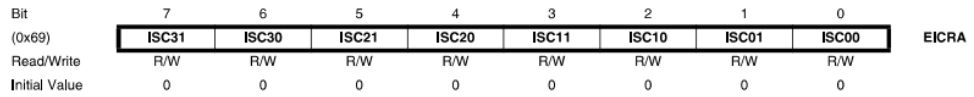


Bit	7	6	5	4	3	2	1	0	
(0x6D)	PCINT23	PCINT22	PCINT21	PCINT20	PCINT19	PCINT18	PCINT17	PCINT16	PCMSK2
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 2.6. PCMSK2

Quando uno di questi gruppi genera una interrupt, viene settato a 1 il bit associato a tale gruppo all'interno del registro **Pin Change Interrupt Flag Register (PCIFR)** e viene azzerato solo dopo che l'interrupt routine termina.
Gli External Interrupts possono essere innescati attraverso **falling edge**, **rising edge** o un **low level** sui pin INT7:0. Questo viene configurato attraverso **External Interrupt Control Register**, in particolare si divide in:

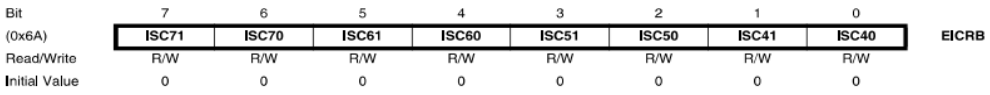
- **EICRA**: serve a configurare i pin INT3:0 e contine i seguenti bit per il controllo delle interrupt:



Bit	7	6	5	4	3	2	1	0	
(0x69)	ISC31	ISC30	ISC21	ISC20	ISC11	ISC10	ISC01	ISC00	EICRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 2.7. EICRA

- **EICRB**: serve a configurare i pin INT7:4 e contine i seguenti bit per il controllo delle interrupt:



Bit	7	6	5	4	3	2	1	0	
(0x6A)	ISC71	ISC70	ISC61	ISC60	ISC51	ISC50	ISC41	ISC40	EICRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 2.8. EICRB

Per configurare la modalità con la quale devono avvenire le interrupt, bisogna settare i valori ISCn1 e ISCn0 in questo modo:

ISCn1	ISCn0	Description
0	0	The low level of INTn generates an interrupt request
0	1	Any edge of INTn generates asynchronously an interrupt request
1	0	The falling edge of INTn generates asynchronously an interrupt request
1	1	The rising edge of INTn generates asynchronously an interrupt request

Figura 2.9. Interrupt Sense Control

I low level interrupt dei pin INT3:0 sono innescati in maniera asincrona, per questo motivo si possono utilizzare per svegliare parti che sono in sleep mode oppure in idle mode. Per abilitare l'interrupt dei pin INT7:0, bisogna settare la maschera dei pin che possono causare una interrupt attraverso il registro **External Interrupt Mask Register (EIMSK)**; in particolare per abilitare l'interruzione su un determinato pin, il relativo bit va settato a 1, altrimenti va settato a 0.

Bit	7	6	5	4	3	2	1	0	
0x1D (0x3D)	INT7	INT6	INT5	INT4	INT3	INT2	INT1	INT0	EIMSK
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 2.10. EIMSK

Infine, quando un pin genera una interrupt all'interno del registro **External Interrupt Flag Register (EIFR)** il relativo bit viene settato a 1 e solo quando l'interrupt routine termina, tale bit viene di nuovo azzerato.

2.4 Timers

I **Timer/Counter units a 16-bit** permettono, se gestita, un'accurata esecuzione del tempo di un programma, generare onde e misurazione delle temporizzazione del segnale. All'interno dei Timer/Counter units è presente un registro 16 bit soprannominato **Timer/Counter (TCNTn)** che ha il compito di tenere il tempo. Il Timer/Counter può essere sincronizzato attraverso una sorgente di clock interna, attraverso il prescaler o attraverso una sorgente di clock esterna sul pin Tn. Il blocco **Clock Select** ha il compito di scegliere la sorgente di clock e il lato del Timer/-Counter usato per incrementere(o decrementare) il suo valore. Il Timer/Counter è inattivo quando nessuna sorgente è stata selezionata. Il Clock Select restituisce come output il clock del timer (clk_{Tn}).
Nei Timer/Counter units sono presenti registri da 16 bit soprannominati **Output Compare Registers (OCRnA/B/C)**, che sono doppiamente bufferizzati e sono confrontati con il valore nel registro Timer/Counter per tutto il tempo. Il risultato del confronto può essere usato dal **Waveform Generator** per generare una **Pulse Width Modulation (PWM)**, oppure un output a frequenza variabile sull' **Output Compare pin (OCnA/B/C)**. L'evento di confronto setterà il **Compare Match Flag (OCFnA/B/C)**, il quale può essere utilizzato per generare delle richieste di interrupts.
Nei Timer/Counter units sono presenti, inoltre, registri soparnnominati **Input Caputure Register** che possono catturare il valore del registro Timer/Counter dato

un evento esterno (se innescato) sul **Input Caputure pin (ICPn)** oppure sui **Analog Comparator pins**.
Per controllare il Timer ci sono i registri a 8 bit soprannominati **Timer/Counter Control Registers TCCRnA/B/C**. Invece tutte le richieste di interrupts (Int.Req.) sono segnalate all'interno del registro **Timer Interrupt Flag Register (TIFRn)**. Tutte le interrupt devvono essere individualmente scelte attraverso il registro **Timer Interrut Mask Register (TIMSKn)**. L'intero funzionamento dei timer si può riassumere con il seguente diagramma a blocchi:

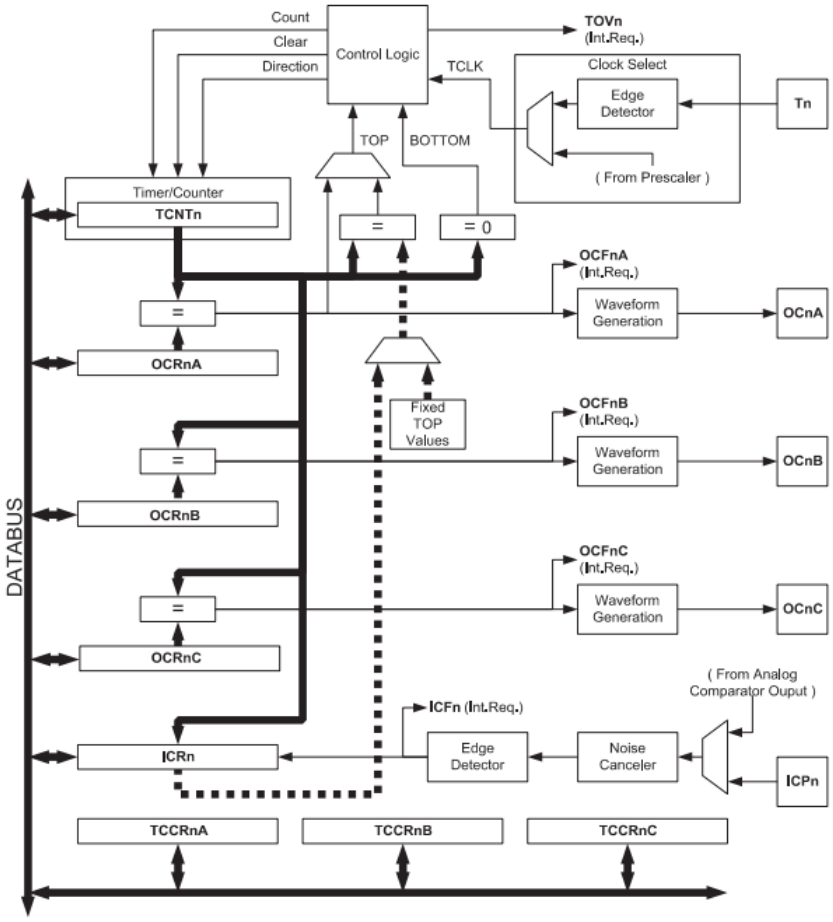


Figura 2.11. Timer/Counter Block Diagram

Prima di continuare la spiegazione diamo le seguenti definizioni:

- **BOTTOM**: il counter raggiunge il BOTTOM quando diventa 0x0000;
- **MAX**: il counter raggiunge il MAXimum quando diventa 0xFFFF;
- **TOP**: il counter raggunge il TOP quando diventa uguale al valore più alto nella count sequence. Il valore TOP può essere assegnato per essere uno dei valori fissi: 0x00FF, 0x01FF o 0x32FF, oppure uno dei valori memorizzato

all'interno dei registri OCRnA o ICRn. L'assegnazione dipende dalla modalità dell'operazione.

Bisogna stare attenti, poiché se si usa OCRnA come TOP value in modalità PWM allora il registro OCRnA non si può usare per generare un output PWM. Tuttavia, in questo caso il TOP value è doppiamente bufferizzato permettendo al TOP value di cambiare in runtime. Se è richiesto un TOP value fisso si può usare in alternativa il registro ICRn, lasciando il registro OCRnA libero per essere usato come output PWM.

Come anticipato prima, i registri TCNTn, OCRnA/B/C e ICRn sono a 16 bit e la CPU AVR può accedere solamente a dati da 8 bit. La modalità di accesso a questi registri viene trascurata in questa spiegazione, ma in breve, per accedere a un registro bisogna usare due istruzioni atomiche e tra un istruzione e l'altra potrebbe avvenire una interrupt corrompendo cosi la lettura o la scrittura. Per tale ragione questi registri si possono considerare come delle **Critical Section** e l'operazione di lettura e scrittura in questa zona va resa atomica nel seguente modo, per comodità viene descritta solo l'operazione di lettura, poiché l'operazione di scrittura è analoga:

```
1  uint16_t TIM16_ReadTCNTn(void){
2      uint8_t sreg;
3      uint16_t i;
4
5      sreg = SREG;                  //Salva il global interrupt flag
6      _disable_interrupt();         //Disabilita le interrupts
7
8      //Inizio Critical Section
9      i = TCNTn;
10     //Fine Critical Section
11
12     SREG = sreg;                 //Abbilitiamo le interrupts
13     return i;
14     }
15
```

Listing 2.1. Lettura atomica dal registro TCNTn

2.4.1 Counter Unit

La parte principale del Timer/Counter unit è il **Counter unit** prgrammabile a 16 bit bidirezionale. Il funzionamento del Counter Unit si può descrivere attraverso il seguente diagramma a blocchi:

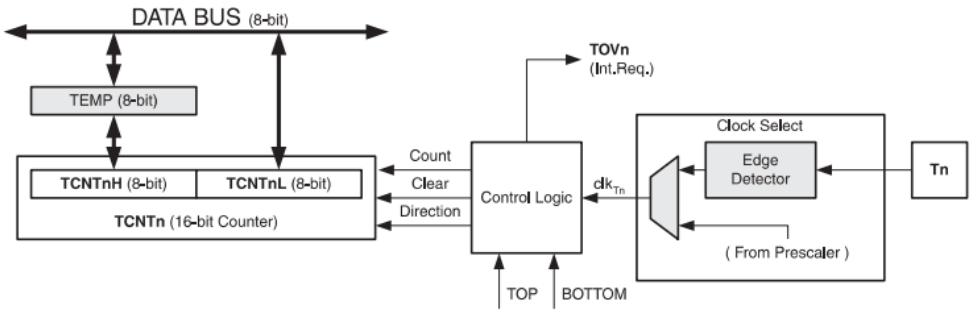


Figura 2.12. Counter Unit Block Diagram

I segnali che vengono scambiati in questo diagramma a blocchi sono:

- **Count**: incrementa o decrementa TCNTn di 1.
- **Direction**: scelta tra incrementare o decrementare.
- **Clear**: libera TCNTn (setta tutti i bit a zero).
- **$\mathbf{clk_{Tn}}$**: Timer/Counter clock.
- **TOP**: segnala che il TCNTn ha raggiunto il valore massimo.
- **BOTTOM**: segnala che il TCNT ha raggiunto il valore minimo (zero).

Il contatore da 16 bit è memorizzato in due spazzi di memoria I/O a 8 bit, in particolare **Counter High (TCNTnH)** che contiene i 8 bit più significativi e **Counter Low (TCNTnH)** che contiene i 8 bit meno significativi.

Bit	7	6	5	4	3	2	1	0	
(0x85)	TCNTn [15:8]								TCNTnH
(0x84)	TCNTn [7:0]								TCNTnL
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 2.13. TCNT

In base alla modalità dell'operazione usata il counter può essere azzerato, incrementato o decrementato a ogni clock del timer (clk_{Tn}). Il clk_{Tn} può essere selezionato attraverso il Clock Select bits (CSn2:0) all'interno del registro TCCRnB. Tuttavia il TCNTn value può essere acceduto dalla CPU in ogni momento anche se clk_{Tn} non è presente. Per selezionare la sorgente di clock del timer si settano i valori dei bit CSn2:0 in base a questa tabella:

CSn2	CSn1	CSn0	Description
0	0	0	No clock source. (Timer/Counter stopped)
0	0	1	$clk_{I/O}$/1 (No prescaling
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on Tn pin. Clock on falling edge
1	1	1	External clock source on Tn pin. Clock on rising edge

Figura 2.14. Clock Select Bit Description

La sequenza di conteggio è determinata settando il **Waveform Generation mode bits** (WGMn3:0) localizzati nei registri TCCRnA e TCCRnB.

Bit	7	6	5	4	3	2	1	0	
(0x80)	COMnA1	COMnA0	COMnB1	COMnB0	COMnC1	COMnC0	WGMn1	WGMn0	TCCRnA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 2.15. TCCRA

Bit	7	6	5	4	3	2	1	0	
(0x81)	ICNCn	ICESn	–	WGMn3	WGMn2	CSn2	CSn1	CSn0	TCCRnB
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 2.16. TCCRB

2.4.2 Input Capture Unit

Il Timer/Counter incorpora un input capture unit che può catturare tramite un evento esterno e restituire il tempo tenuto. Il segnale esterno indica un evento,o una serie di eventi, che possono essere applicati via il ICPn pin (o solo per il Timer 1 attraverso l'Analog Comparator Unit). Questa cattura si può usare per calcolare la frequenza o altre features di un segnale applicato, oppure per creare un log dei eventi. Mostriamo il funzionamento dell'Input Capture Unit attraverso il seguente diagramma a blocchi:

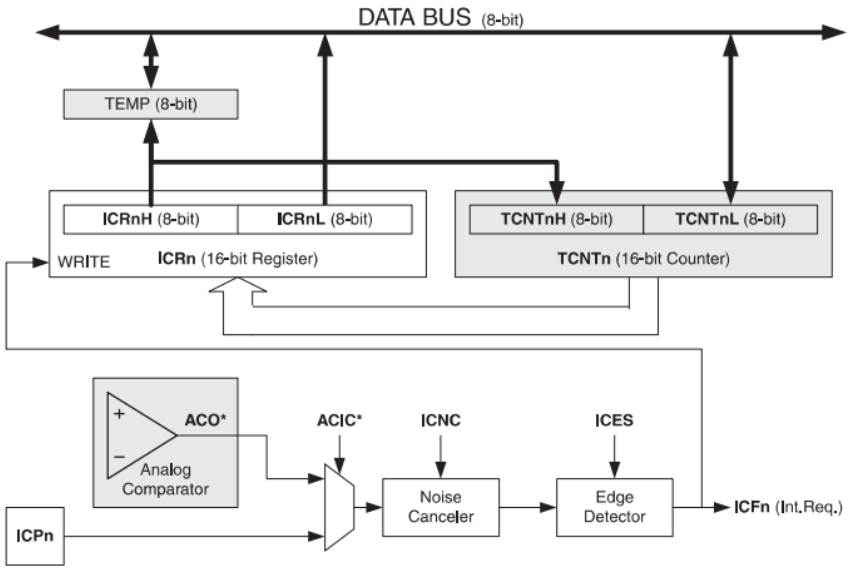


Figura 2.17. Input Capture Unit Block Diagram

Quando avviene un cambiamento del livello logico (un evento) sul pin ICPn (oppure sul Analog Compare Output nel caso del Timer1) viene innescata una cattura e il valore del TCNTn viene scritto in ICRn e viene settato l'**Input Capture Flag (ICFn)** nello stesso istante. Se abilitato (TICIEn = 1), l'input capture flag genera una interrupt. Tale flag viene azzerato dopo l'interrupt routine oppure via software scrivendo uno logico nella posizione del relativo bit.
Il registro ICRn può essere scritto solo quando usiamo Waveform Generation mode che utilizza questo registro per definire il valore TOP del counter. In questo caso i bit della Waveform Generation mode (WGMn3:0) devono essere settati prima di scrivere sul registro ICRn. Quando scriviamo sul registro ICRn, bisogna scrivere prima su ICRnH e poi ICRnL. Quando bisogna leggere tale registro dopo che è stata innescata una interrupt a causa di evento, nella ISR questo registro va letto il prima possibile, prima che venga modificato a causa di un'altro evento.

Bit	7	6	5	4	3	2	1	0	
(0x87)	ICRn [15:8]								ICR1H
(0x86)	ICRn [7:0]								ICR1L
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 2.18. ICRn

2.4.3 Output Compare Units

Il comparatore a 16 bit confronta continuamente il registro TCNTn con l'Output Compare Register (OCRnx). Se il registro TCNTn e il registro OCRnx sono uguali viene segnalato il confronto. Se abilitato (OICEnx = 1), l'Output Compare Flag genera un'interrupt. Il OCFnx Flag è automaticamente azzerato dopo che l'ISR viene eseguita oppure via software scrivendo uno logico nella posizione del relativo bit. Il Waveform Generetor utilizza il segnale del confronto per generare un output in

accordo con la Waveform Generation mode e il Compare Output mode. Si può notare che il valore TOP permette di definire un periodo per l'onda generata attraverso il Waveform Generator. Il funzionamento dell'Output Compare Unit si può descrivere con il seguente diagramma a blocchi:

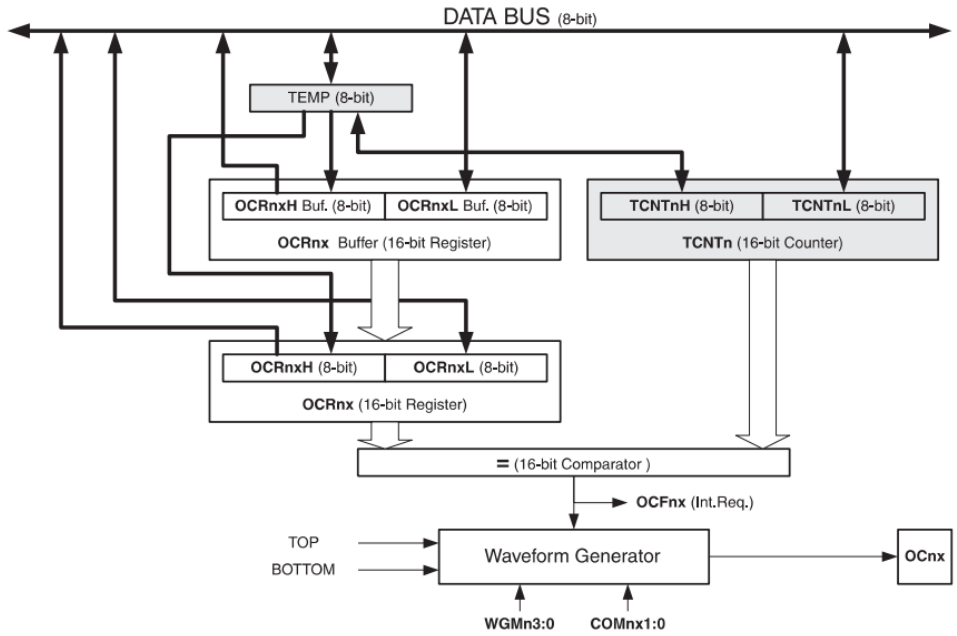


Figura 2.19. Output Compare Unit Block Diagram

Il registro OCRnx è doppiamente bufferizzato quando utilizziamo una delle dodici PWM modes. Per la Normal mode e la Clear Timer on Compare (CTC) il doppio buffer è disabilitato. Quando il doppio buffer è abilitato la CPU ha accesso al OCRnx Buffer Register, quando invece il doppio buffer è disabilitato, la CPU ha acesso diretto ala OCRnx. Il valore contenuto in OCRnx può essere modificato solo attraverso una scrittura, non può essere modificato automaticamente. Inoltre, è buona regola scrivere prima i bit più significativi (OCRnxH) e poi quelli meno significativi (OCRnxL).

Bit	7	6	5	4	3	2	1	0	
(0x89)	OCRnx [15:8]								OCRnx H
(0x88)	OCRnx [7:0]								OCRnx L
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 2.20. OCRnx

Se il Waveform Generation mode è in non-PWM, l'output del confronto può essere forzato scrivendo 1 nel bit **Force Output Compare (FOCnx)**. Forzando il comparatore non verrà settato il OCFnx Flag, però il pin OCnx sarà aggiornato come se fosse avvenuta una comparazione reale.

2.4.4 Compare Match Output Unit

I **Compare Output mode (COMnx1:0) bit** hanno due funzioni:

1. vengono usate dal Waveform Generator per definire l'**Output Compare (OCnx) state** per il prossimo confronto;

2. vengono usati per controllare l'output della sorgente dei pin OCnx.

Il funzionamento del Compare Match Output Unit è descritto nel seguente diagramma a blocchi:

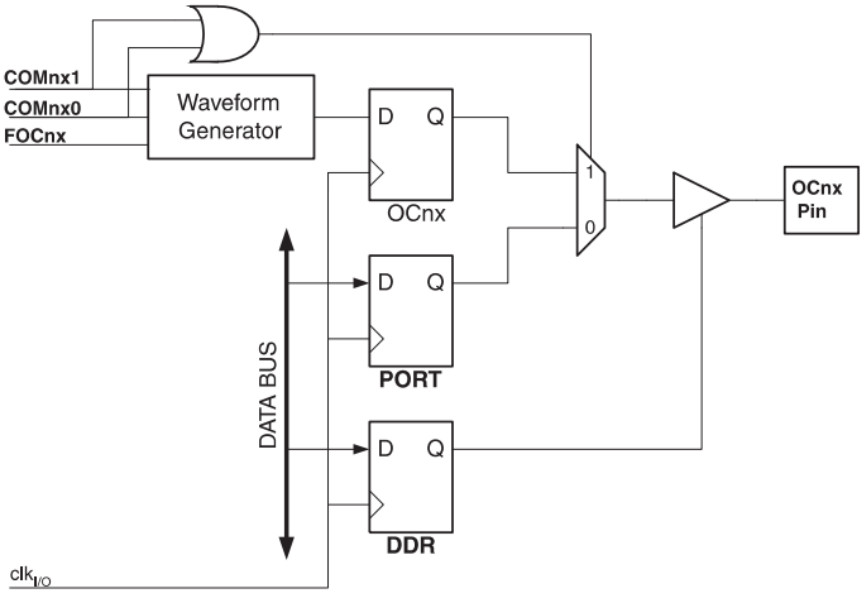


Figura 2.21. Compare Match Output Unit Scheme

Quando ci riferiamo allo OCnx state, intendiamo il registro OCnx e non il pin OCnx. Se il sistema è riavviato il registro OCnx viene azzerato.
Le funzionalità generatli della porta I/O sono soprascritte attraverso Output Compare (OCnx) dal Waveform Generator se uno dei bit COMnx1:0 è settato a 1. Tuttavia, la direzione del OCnx pin (input o output) rimane controllata dal Data Direction Register (DDR) per la porta pin. Il Data Direction Register bit per OCnx pin (DDR_OCnx) va settato come output prima che OCnx value sia visibile sul pin. Lo schema del Output Compare pin logic permette di inizializzare lo stato OCnx prima che l'output sia abilitato.
Il Waveform Generator usa i bit COMnx1:0 in maniera differente in base alla modalità usata, in particolare per ogni modalità possiamo avere i seguenti effetti:

COMnA1 COMnB1 COMnC1	COMnA0 COMnB0 COMnC0	Description
0	0	Normal port operation, OCnA/OCnB/OCnC disconnected
0	1	Toggle OCnA/OCnB/OCnC on compare match
1	0	Clear OCnA/OCnB/OCnC on compare match (set output to low level)
1	1	Set OCnA/OCnB/OCnC on compare match (set output to high level)

Figura 2.22. Compare Output Mode Non PWM

COMnA1 COMnB1 COMnC1	COMnA0 COMnB0 COMnC0	Description
0	0	Normal port operation, OCnA/OCnB/OCnC disconnected
0	1	WGM13:0 = 14 or 15: Toggle OC1A on Compare Match, OC1B and OC1C disconnected (normal port operation). For all other WGM1 settings, normal port operation, OC1A/OC1B/OC1C disconnected
1	0	Clear OCnA/OCnB/OCnC on compare match, set OCnA/OCnB/OCnC at BOTTOM (non-inverting mode)
1	1	Set OCnA/OCnB/OCnC on compare match, clear OCnA/OCnB/OCnC at BOTTOM (inverting mode)

Figura 2.23. Compare Output Mode Fast PWM

COMnA1 COMnB1 COMnC1	COMnA0 COMnB0 COMnC0	Description
0	0	Normal port operation, OCnA/OCnB/OCnC disconnected
0	1	WGM13:0 =9 or 11: Toggle OC1A on Compare Match, OC1B and OC1C disconnected (normal port operation). For all other WGM1 settings, normal port operation, OC1A/OC1B/OC1C disconnected
1	0	Clear OCnA/OCnB/OCnC on compare match when up-counting Set OCnA/OCnB/OCnC on compare match when downcounting
1	1	Set OCnA/OCnB/OCnC on compare match when up-counting Clear OCnA/OCnB/OCnC on compare match when downcounting

Figura 2.24. Compare Output Mode Phase and Frequency Correct PWM

Il cambiamento dello stato dei COMnx1:0 bit ha effetto dopo il primo confronto. Per i non-PWM l'azione può essere forzata per avere un effetto immediato usando i bit FOCnx.

2.4.5 Modalità di operazione

Le modalità di operazione sono definite dal Waveform Generation mode (WGMn3:0) e dal Compare Output mode (COM1:0). Il Waveform Generation mode influisce sulla sequenza di conteggio, mentre il Compare Output mode controlla se l'output della PWM generata deve essere invertente o non invertente.
Le modalità di operazione sono:

- **Normal mode**: in questa modalità il conteggio avviene unicamente in una sola direzione, ovvero viene incrementato. Il counter quando raggiunge il valore MAX (0xFFFF) viene riavviato dal valore BOTTOM (0x0000). Normalemento

il TOVn Flag viene settato nello stesso ciclo in cui avviene il raggiungimento del valore massimo ma in questo caso il TOVn Flag ha il compito di essere come il 17-essimo bit, con l'eccezione che viene solo settato e non viene azzerato. Tuttavia, si può combinare con l'overflow interrupt che azzera automaticamente tale registro al termine dell'ISR. L'Output Compare unit può essere utilizzata per generare interrupt in un dato momento. Usare l'Output Compare unit per generare forme d'onda in questa modalità non è raccomandato, perché occuperebbe troppo tempo alla CPU.

- **Clear Timer on Compare Match (CTC) mode**: in questa modalità il counter è azzerato quanto il counter value (TCNTn) raggunge i valori in OCRnA oppure in ICRn, in base alla modalità scelta. I valori OCRnA o ICRn definiscono il valore più alto per il counter. Possiamo mostrare il timing diagram per la modalità CTC e come evolvono i valori dei vari registri:

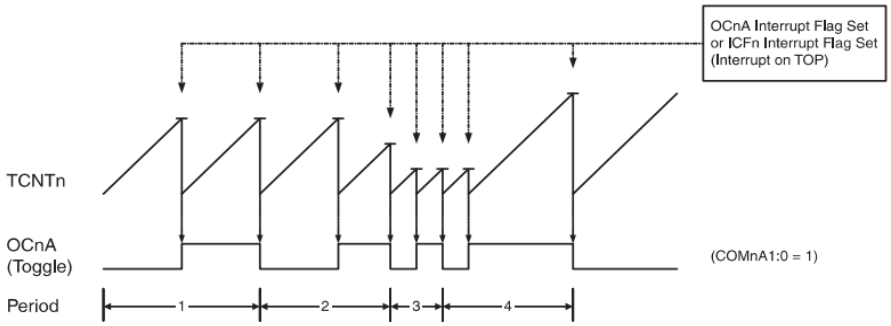


Figura 2.25. CTC Mode Timing Diagram

Un'interrupt può essere generata attraverso OCFnA oppure ICFn ogni volta che il counter value raggiunge il valore TOP. Se l'interrupt è abilitato si può usare l'interrupt routine per aggionare il valore TOP.
Per generare un onda in output si può settare l'OCnA output ad alternare il suo livello logico ad ogni confronto settanto il Compare Output mode bit a toggle mode (COMnA1:0=1). Il OCnA valo non sarà visibile sulla porta pin finchè il data direction per il pin non è settato in output (DDR_OCnA=1). La frequenza dell'onda è definita attraverso questa equazione:

$$f_{OCnA} = \frac{f_{clk\_I/O}}{2 * N * (1 + OCRnA)} \tag{2.1}$$

Dove N è la variabile che rappresenta il fattore di prescaler.

- **Fast PWM Mode**: questa modalità permette di generare un onda PWM ad alta frequenza. La Fast PWM differisce dalle altre opzioni di PWM dal fatto che è una single-slop operation (operazione su un solo pendio). Il counter parte da BOTTOM e quando arriva a TOP riparte di nuovo da BOTTOM.In non - inverting Compare Output mode, l'Output Compare (OCnx) è azzerato quando avviene il congronto tra TCNTn OCRnx ed è settato a BOTTOM. Nel caso invertente avviene l'opposto. Il valore più alto assunto dal counter può essere

fissato a 8 bit, 9 bit o 10 bit oppure definito tramite il registro ICRn oppure OCRnA (possono assumere un valore minimo di 0x0003). Nel fast PWM mode il counter è incrementato fino a che il suo valore non raggiunge il valore TOP. Il counter è azzerato ciclo di clock successivo dopo il raggiungimento del valore massimo. Possiamo mostrare il timing diagram per la modalità Fast PWM e come evolvono i valori dei registri:

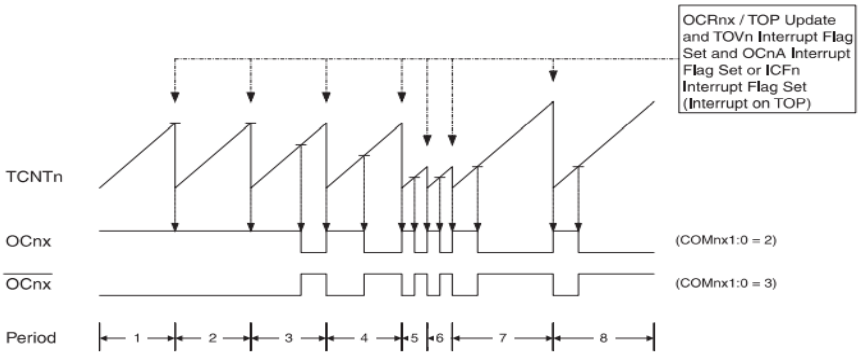


Figura 2.26. Fast PWM Mode Timing Diagram

Nel fast PWM mode, il compaare unit pemette di generare onde PWM sui pin OCnx. Settando adeguatamente i bit COMnx1:0 possiamo generare onda in modalità invertete o non invertente. Il valore attuale OCnx può diventare visibile sulla porta pin se la data direction per la porta pin è settata in output (DDR_OCnx).
La frequenza dell'onda è definita attraverso questa equazione:

$$f_{OCnxPWM} = \frac{f_{clk\_I/O}}{N * (1 + OCRnA)} \tag{2.2}$$

Dove N è la variabile che rappresenta il fattore di prescaler.

Mode	WGMn3	WGMn2 (CTCn)	WGMn1 (PWMn1)	WGMn0 (PWMn0)	Timer/Counter Mode of Operation	TOP	Update of OCRnx at	TOVn Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCRnA	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	BOTTOM	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	BOTTOM	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	BOTTOM	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICRn	BOTTOM	BOTTOM
9	1	0	0	1	PWM,Phase and Frequency Correct	OCRnA	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICRn	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCRnA	TOP	BOTTOM
12	1	1	0	0	CTC	ICRn	Immediate	MAX
13	1	1	0	1	(Reserved)	–	–	–
14	1	1	1	0	Fast PWM	ICRn	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCRnA	BOTTOM	TOP

Figura 2.27. Waveform Generation Mode Bit Description

2.5 Motore DC con Encoder

Il motore DC con Encoder GM25-370 è motore da 12V formato da una bobina in rame con elevata conduttività, da una spazzola in carbone ingranaggi in metallo e piastrelle magnetiche ad alta resistenza. Il motore ospita anche un encoder, ovvero un dispositivo capace di misurare la rotazione del rotore. Il motore viene alimentato da una tensione da 12V e in base all'intensità di corrente e di tensione il motore può variare velocità.

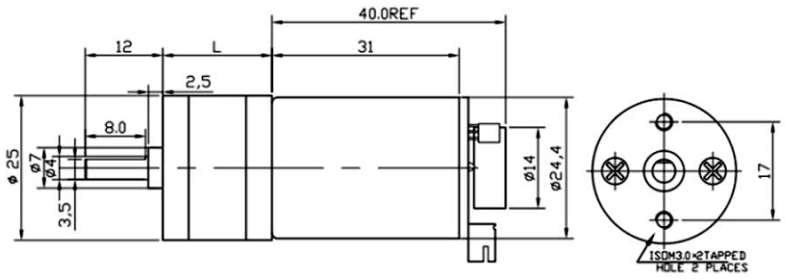


Figura 2.28. Motore DC

2.6 Ponte H

Il ponte H è un circuito elettronico che può funzionare nei quattro quadranti del piano corrente-tensione sul carico, in questo modo permette al motore di ruotare in due direzioni. In particolare il modello L298H permette anche l'inserimento di un onda quadra capace di regolare la velocità del motore. Il circuito logico del L298H equivalente è il seguente:

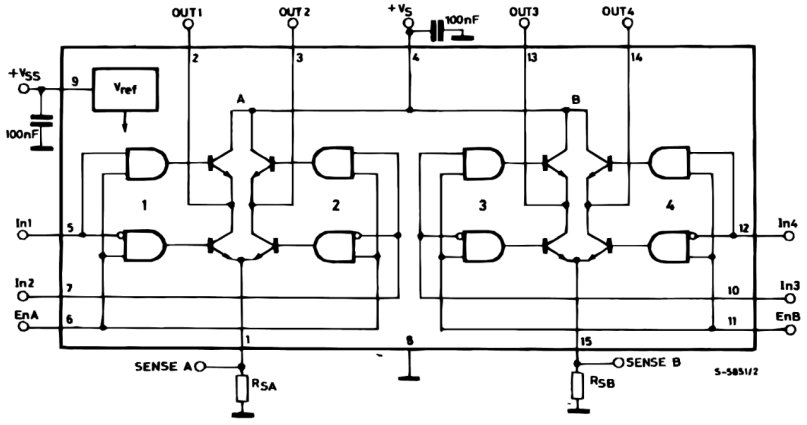


Figura 2.29. L298H

Capitolo 3

Descrizione del sistema

Il progetto, come visto nel capitolo precedente, è formato da diversi strumenti elettronici che devono essere composti nel seguente modo:

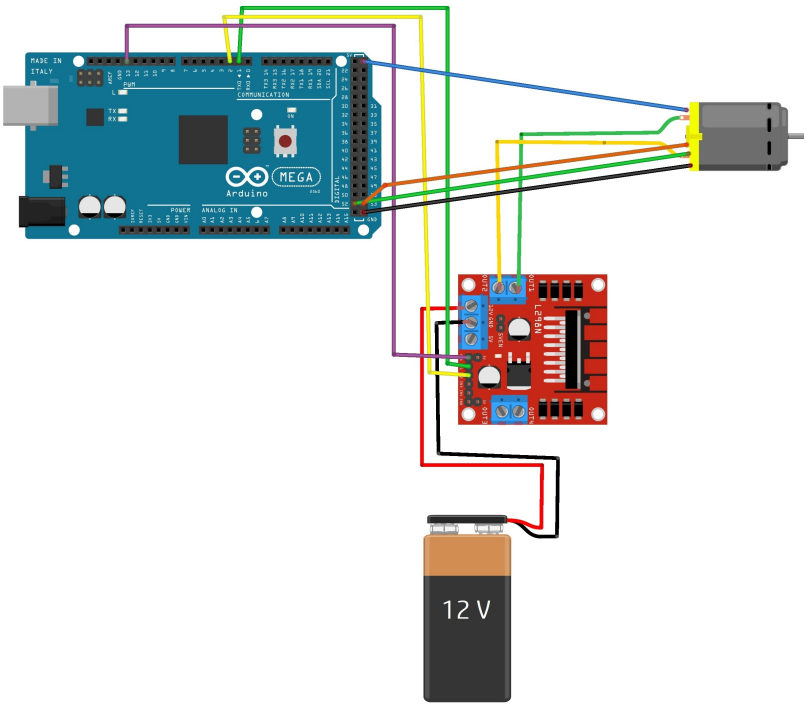


Figura 3.1. Schema elettronico del controllore

Per la realizzazione del controllore vengono realizzate alcune librerie ausiliare. In seguito ci occuperemo a descrivere il funzionamento di queste queste librerie e il ruolo che assumono nella realizzazione del controllore.

3.1 Digital I/O

La libreria **Digital I/O** ha lo scopo di fornire la direzione di rotazione del motore sfruttando le porte pin I/O dell'AVR. Vengono realizzate due funzioni:

- **DigIO_REGE_setDirection**: ha il compito di settare la direzione di un pin (input o output);
- **DigIO_REGE_setValue**: ha il compito di scrivere un valore su tale porta.

```
1  void DigIO_init(void) {}
2
3  void DigIO_REGE_setDirection(uint8_t bit, uint8_t dir) {
4      uint8_t mask = 0x01 << bit;
5      if (dir)
6          DDRE |= mask;
7      else
8          DDRE &= ~mask;
9  }
10
11 void DigIO_REGE_setValue(uint8_t bit, uint8_t dir) {
12     uint8_t mask = 0x01 << bit;
13     if (dir)
14         PORTE |= mask;
15     else
16         PORTE &= ~mask;
17 }
18
```

Listing 3.1. Digita I/O

3.2 Encoder

La libreria **Encoder** ha lo scopo di realizzare un lettore dell'encoder del motore. In particolare per realizzare tale lettore facciamo uso delle External Interrupt, le quali ci permettono ogni volta che avviene una commutazione sui pin di generare una interrupt ed quindi eseguire una interrupt routine. Per creare le External Interrupt facciamo uso delle porte pin 52 e 53 appartenenti al gruppo dei pin PCI0. Per poter realizzare il lettore bisogna prima definire alcune strutture globali:

- **position_status**: questa variabile è un contatore che ha lo scopo di memorizzare la posizione corrente del rotore del motore.
- **transition_table**: questo array ha lo scopo di descrivere le transizioni possibili di uno Shaft Encoder, in accordo con la seguente macchina a stati:

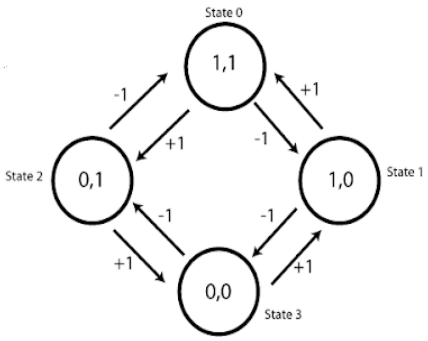


Figura 3.2. Shaft Encoder Bit Transition

Per poter usare il lettore dell'encoder vengono implementate le seguenti funzioni:

- **encoder_init**: questa funzione ha il compito di settare i pin 52 e 53 in lettura e abilitare le External Interrupt su tali pin.
- **encoder_read**: questa funzione restituisce il valore contenuto all'interno contatore che indica la posizione del rotore.

```
#define PIN_MASK 0x03     //Usiamo gli ultimi due bit della porta B (52-53)

volatile uint8_t previous_pins;
volatile uint8_t current_pins;

volatile uint16_t position_status;


static const int8_t transition_table[] = {
   0,     //00->00
   1,     //00->01
  -1,     //00->10
   0,     //00->11
  -1,     //01->00
   0,     //01->01
   0,     //01->10
   1,     //01->11
   1,     //10->00
   0,     //10->01
   0,     //10->10
  -1,     //10->11
   0,     //11->00
  -1,     //11->01
   1,     //11->10
   0      //11->11


};


void encoder_init(void){
  cli();          //Disabilitiamo le interrupt

  DDRB &= ~PIN_MASK;    //Settiamo la PIN_MASK come input per poter leggere l'encoder
  PORTB |= PIN_MASK;    //Abilitimo i pull up resistors (poiche sono porte settate in input)

  PCICR |= (1<< PCIE0); //Abilitiamo Pin Change Interrupt 0, ogni cambiamento su PCINT7:0
            //causa interrupt ed eseguie la routine del PCI0 Interrupt Vector
  PCMSK0 |= PIN_MASK;   //I pin PCINT7:0 sono abilitati individualmente dal registro PCMSK0
            //in particolare abilitiamo:
            //- PCINT0 = pin 53 Arduino
            //- PCINT1 = pin 52 Arduino
```

```
43
44
45    previous_pins = PIN_MASK; //Settaggio dei pin precedenti ai pin
        attuali
46    current_pins = PIN_MASK;    //Lettura dei pin attuali
47
48    position_status = 0;
49
50    sei();          //Abilitiamo le interrupt
51
52  }
53
54
55  uint16_t encoder_read(void){
56    return position_status; //Restituiamo la posizione del encoder
57  }
58
59
60  ISR(PCINT0_vect){
61    previous_pins = current_pins;     //Imposto i pin precedenti i pin
        attuali ai pin precedenti
62    current_pins = PINB;    //Leggo i pin e gli imposto come attuali;
63
64    //Generazione dell'indice della tabella
65    uint8_t TT_index = ( previous_pins & PIN_MASK ) << 2 | (
        current_pins & PIN_MASK );
66
67    //Incremento dello stato corrente del encoder
68    position_status += transition_table[TT_index];
69
70  }
```

Listing 3.2. Encoder

3.3 Timer

La libreria Timer sfrutta il timer 5 dell'AVR per generare interrupt a intervalli regolari, permettendo in questo modo di avere una esecuzione del programma più controllata in termini di tempo. Tale libreria è composta solamente da una sola funzione **timer5_init**, che una volta lanciata ogni **timer_duration_ms** ms viene generata una interrupt e gestita tramite una interrupt routine.

```
1  void timer5_init(int timer_duration_ms){
2
3    // Cofiguraimo il time con modalita:
4    // - CTC
5    // - Prescaler = 1024
6    TCCR5A = 0;
7    TCCR5B = (1 << WGM52) | (1 << CS50) | (1 << CS52);
8
9    //Settiamo il valore OCR = (16MHz/1024)* time_duration_ms
10   uint16_t ocrval=(uint16_t)(15.62*timer_duration_ms);
11   OCR5A = ocrval;
12
```

```
13   cli();              //Cancelliamo gli interrupt
14
15   TIMSK5 |= (1<<OCIE5A);    //Abilita timer interrupt
16
17   sei();              //Abilita tutti gli interrupt
18 }
```

Listing 3.3. Timer

3.4 PWM Signal

La libreria PWM Signal ha il compito di generare un onda PWM sfruttando il timer 1. In particolare, il timer viene messo in modalità fast PWM (WGM50 = 1, WGM52 = 1) non invertente (COM5C1 = 1) e viene scelta una sorgente interna per il timer senza prescaler (CS50 = 1) e settiamo la porta pin 13 in output per poter leggere da esso l'onda PWM. Questa libreria è composta da due funzioni:

- **pwm_init**: questa funzione ha lo scopo di settare tutti i parametri necessari per il timer 1 al fine di generare l'onda PWM.
- **pwm_set_intensity**: questa funzione permette di settare l'intensità dell'onda PWM generata.

```
1  // Mettiamo le varie modalita:
2  // - fast PWM attraverso (WGM50 = 1, WGM52 = 1)
3  // - non invertente (COM5C1 = 1) : ovvero che l'impulso avviene all'
       inizio del period
4  // - nessun prescaler (CS50 = 1).
5  #define TCCRA_MASK (1<<WGM10)|(1<<COM1C0)|(1<<COM1C1)
6  #define TCCRB_MASK (1<<WGM12)|(1<<CS10)
7
8  void pwm_init(void){
9
10   TCCR1A = TCCRA_MASK;    //Impostiamo i registri di TCCR A
11   TCCR1B = TCCRB_MASK;    //Impostiamo i registri di TCCR B
12
13   //Puliamo tutti i bit alti per il compere timer
14   OCR1AH = 0;
15   OCR1BH = 0;
16   OCR1CH = 0;
17   OCR1CL = 1;
18
19   //Settiamo il PIN 7 PORTB in output
20   const uint8_t mask = (1<<7);
21   DDRB |= mask;
22
23 }
24
25 void pwm_set_intensity(uint8_t intensity){
26
27   //Settiamo l'intensita
28   OCR1CL = 255 - intensity;
29 }
```

Listing 3.4. PWM Signal

3.5 Motor

La libreria Motor implementa un controllore PID, che dipende da tutte le librerie sopra descritte, per controllare motori DC attraverso un onda PWM. All'interno di questa libreria è definita una struttura **Motor** che rappresenta il motore reale, nella quale sono state definite alcune variabili di stato:

- **angular_position**: contatore che indica la posizione corrente del rotore del motore;
- **angular_velocity**: variabile che indica la velocità corrente del rotore del motore;

Nella struttura dati viene salvata anche la variabile di controllo che deve essere applicata al motore, tale variabile è rappresentata da:

- **current_pwm**: variabile che indica l'intensità corrente di onda PWM da applicare.

Per controllare il motore, assumiamo che questo sistema sia **model-free**, ovvero non abbiamo un modello matematico capace di descriverci il comportamento del motore. Per tale ragione utilizziamo un controllore in controreazione noto come controllore PID.
Il **controllore PID** è un controllore in controreazione che prende come ingresso l'errore definito come la differenza tra la grandezza di riferimento, velocità desiderata nel nostro caso, e l'uscita del processo. La variabile di uscita di questo controllore è la variabile di controllo del processo, nel nostro caso l'intensità dell'onda PWM, ed è data dalla somma di tre contributi:

- **Proposizionale**: l'errore viene moltiplicato per una costante moltiplicativa.
- **Integrale**: l'integrale dell'errore viene moltiplicato per per una costante moltiplicativa.
- **Derivativo**: la derivata dell'errore viene moltiplicato per una costante moltiplicativa.

Il diagramma a blocchi di tale sistema è il seguente:

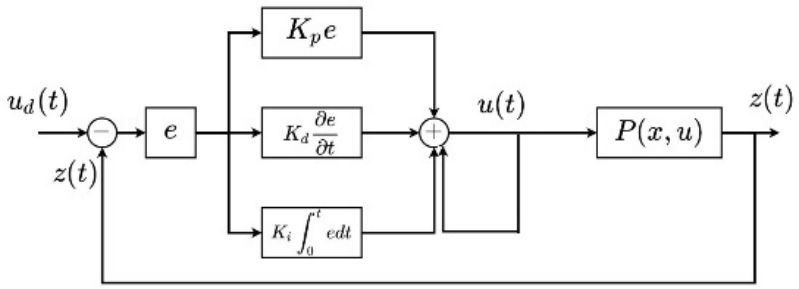


Figura 3.3. Diagramma Controllore

Per realizzare il controllore utilizziamo l'AVR che è un sistema digitale che lavora a tempo discreto. Per tale ragione il controllore PID sarà discretizzato approssimando la derivata attraverso il rapporto incrementale e l'integrale attraverso una sommatoria.
La struttura che descrive il motore all'interno dell'AVR è la seguente:

```
typedef struct{

  // Variabili di stato
  int16_t angular_position;
  int16_t angular_velocity;
  int16_t desired_velocity;
  float error;
  float error_acc;

  // Variabili di controllo
  uint8_t dira, dirb;
  int16_t current_pwm;

  // Variabile scelta tipo controllore
  ctrtype_t type_controller;
  float Kp;
  float Ki;
  float Kd;
  float max_error;
  float max_error_integral;

} Motor;
```

Listing 3.5. Motor Struct

Per poter controllare il motore nella libreria Motor le funzioni principali sono:

- **Motor_init**: questa funzione ha il compito di settare tutti i pin, timer, registri e interrupt necessari al funzionamento del motore, in oltre riceve come parametri le tre costanti moltiplicative che definiscono il controllore PID.
- **set_desired_velocity**: questa funzione setta la velocità desiderata con la quale il motore deve girare.
- **spin_once**: questa è una funzione particolare che deve essere eseguita all'interno di una interrupt routine in maniera controllata, poiché esegue tutte le istruzioni che sono alla base della logica del controllore PID.

```
#define LOOP_TIMING 100
typedef enum {OPEN_LOOP, CLOSE_LOOP} ctrtype_t;

float clamp(float input_val, float range_max){
  if(input_val > range_max)
    return  range_max;
  else if(input_val < -range_max)
    return -range_max;
  else
    return input_val;
}
```

```
12
13 // Definizione delle funzioni riguardanti il motore:
14 Motor* Motor_init(float K_p, float K_i, float K_d){
15
16   //Inizzializzazione di tools indispensabili per il motore
17   pwm_init();     // Controllo generatore d'onda PWM
18   encoder_init();   // Lettura dello stato corrente
19   timer5_init(LOOP_TIMING); // Controllo di eventi con intervalli di
       tempo regolari
20
21   // Allocazione dell'oggetto Motor
22   Motor* mtr = (Motor*) malloc(sizeof(Motor));
23
24   // Inizializzazione dei parametri del controllore PID
25   mtr->Kp = K_p;
26   mtr->Ki = K_i;
27   mtr->Kd = K_d;
28   mtr->max_error = 20.f;
29   mtr->max_error_integral = 50.f;
30
31   // Inizializzazione di parametri
32   mtr->angular_velocity = 0;
33   mtr->angular_position = 0;
34   mtr->desired_velocity = 0;
35   mtr->error = 0;
36   mtr->dira = 4;
37   mtr->dirb = 5;
38
39   // Inizializzazione dell'intensita a zero
40   mtr->current_pwm = 0;
41   mtr->error_acc = 0;
42
43   //Setto in ouput i registri per la direzione del motore
44   DigIO_REGE_setDirection(mtr->dira,1);
45   DigIO_REGE_setDirection(mtr->dirb,1);
46
47
48   return mtr;
49
50 }
51
52
53 void Motor_detach(Motor* mtr){
54   //Rilascio lo spazio dedicato al motore;
55   free(mtr);
56 }
57
58
59 void set_type_controller(Motor* mtr, ctrtype_t type_controller){
60   //Setto il tipo di controller
61   mtr->type_controller = type_controller;
62 }
63
64
65
66 void set_desired_velocity(Motor* mtr, uint16_t desired_velocity){
67   //Imposto la velocita desiderata
```

```
68    mtr->desired_velocity = desired_velocity;
69
70  }
71
72
73  void spin_once(Motor* mtr){
74    uint16_t curr_pos = encoder_read();           // Lettura posizione
       corrente
75    uint16_t prev_pos = mtr->angular_position;    // Lettura posizione
       precedente
76
77    //Calcoliamo la velocita misurata
78    int16_t curr_vel = curr_pos - prev_pos;
79    int16_t des_vel = mtr->desired_velocity;
80
81
82    mtr->angular_position = curr_pos;       // Setto posizione attuale
83    mtr->angular_velocity = curr_vel;         // Setto velocita attuale
84
85    switch(mtr->type_controller){
86
87      case CLOSE_LOOP:;
88        // Caso : Controllore ad anello chiuso
89
90        // Calcolo dell'errore e lo limitiamo in un intervallo per fare
       un ingresso a rampa
91        // per evitare che il controllore sia instabile all'inizio.
92        float prev_err = mtr->error;
93        float curr_err = curr_vel - des_vel;
94          curr_err = clamp(curr_err, mtr->max_error);
95          mtr->error = curr_err;
96
97        //Calcolo dell'ingresso integrale, con un range massimo per
       anti wind up
98          mtr->error_acc = clamp(mtr->error_acc + mtr->Ki * curr_err, mtr
       ->max_error_integral);
99        float u_i = mtr->error_acc;
100
101       //Calcoliamo ingresso derivato4
102       float u_d = mtr->Kd * (curr_err - prev_err);
103
104       //Calcoliamo ingresso proporzionale
105       float u_p = mtr->Kp * curr_err;
106
107       // Limitiamo l' onda quadra tra +- 255
108       mtr->current_pwm =(int16_t) clamp(mtr->current_pwm + u_i + u_d
       + u_p, 255.); //Settiamo l'intensita della pwm corrente
109
110       break;
111
112
113     case OPEN_LOOP:
114       // Caso : Controllore ad anello aperto
115       mtr->current_pwm = (int16_t) des_vel;
116
117       break;
118   }
```

```
119
120   //  Controlo comand per il controllo dell'hardware
121   if(mtr->current_pwm < 0){
122     DigIO_REGE_setValue(mtr->dira, 1);
123     DigIO_REGE_setValue(mtr->dirb, 0);
124   }
125   else{
126     DigIO_REGE_setValue(mtr->dira, 0);
127     DigIO_REGE_setValue(mtr->dirb, 1);
128   }
129
130   pwm_set_intensity(abs(mtr->current_pwm));
131 }
```

Listing 3.6. Motor

Nota: La funzione ausiliaria **clamp** è creata con lo scopo di limitare l'errore, poiché senza tale funzione l'errore in partenza sarebbe molto elevato rendendo il sistema instabile. Per tale ragione creiamo un limitatore in modo tale che il sistema riceva come ingresso una rampa.

Capitolo 4

Esperimenti e casi d'uso

Dopo aver spiegato il funzionamento del controllore e le varie librerie, è giunto il momento di unire i pezzi e mostrare come deve essere implementato il programma che deve essere eseguito sull'AVR.

4.1 Main Program

Per eseguire il programma principale, per prima cosa bisogna trovare i parametri del controllore PID. Essendo che il motore è un processo model-free, tali parametri sono stati trovati in maniera empirica e assumono questi valori:

```
1 //Parametri PID
2 #define K_P .4f
3 #define K_I 0.0f
4 #define K_D .1f
```

Listing 4.1. Costanti PID

Nel programma principale bisogna definire un riferimento globale all'oggetto motore, in modo che è accessibile da tutte le funzioni e interrupt routine che hanno bisogno di accedere a tale oggetto:

```
1 Motor* mtr;
```

Listing 4.2. Oggetto Motor

Una volta definite le variabili globali, si può mostrare la struttura della **main** del controllore. Possiamo dividere il main in due parti:

- **Setup (rows 2:11)**: in questa parte viene creato l'oggetto motore, settati tutti i registri necessari e la tipologia del controllore (CLOSE_LOOP oppure OPEN_LOOP) ed infine attivati gli interrupt.
- **Loop (rows 12:35)**: in questa parte abbiamo una lettura di eventuale velocità desiderata da settare, e viene comunicato costantemente il proprio stato.

```
1 int main(void){
2   //Settiamo lo slave
3   UART_init();
4   mtr = Motor_init(K_P, K_I, K_D);
```

```
  set_type_controller(mtr, CLOSE_LOOP);
  Slave_Addr_init(0x01, 1);                    //inizializza TWI
  sei();                                       //abilita le
    interruzioni globali


  uint8_t current_state[VELOCITY_LEN]= "P:0, CV:0, DV:0";
  uint8_t desired_velocity[VELOCITY_LEN]= "0";

  while(1){
    //Ricevi velocita
    TWI_Slave_Receive_Data();           //ricevo un comando

    if (!strcmp(Receive_Buffer, SET_command)){
      TWI_Slave_Receive_Data();          //ricevo desired velocity
      if (*Receive_Buffer != 0) strcpy(desired_velocity,
    Receive_Buffer);   //inscerisco in desired velocity
    }
    else if(!strcmp(Receive_Buffer, APPLY_command)){
      // Conversione e setta velocita
      int vel = atoi(desired_velocity);
      set_desired_velocity(mtr, vel);
    }
    else if(!strcmp(Receive_Buffer, SAMPLE_command)){
      //Lettura stato motore
      sprintf(current_state, "P:%d, CV:%d, DV:%d", mtr->
    angular_position, mtr->angular_velocity , mtr->desired_velocity);
    }
    else if (!strcmp(Receive_Buffer, GET_command)){
      TWI_Slave_Transmit_Data(current_state, VELOCITY_LEN); //invio
    velocita
    }
    else{
      UART_putString("Non ho ricevuto un comando valido\n");
    }
  }
  return 0;
}
```

Listing 4.3. Oggetto Motor

La logica del controllore PID, come accenato, deve essere inserita all'interno di una interrupt service routine in modo tale che abbia una esecuzione controllata in termini di tempo. Sfruttiamo le interrupt generate dal timer 5 per questo scopo e inseriamo la funzione **spin_once** all'interrno dell'ISR.

```
ISR(TIMER5_COMPA_vect) {
  spin_once(mtr);
}
```

Listing 4.4. spin once

Capitolo 5

Conclusione

Si è cercato di mostrare come poter realizzare un controllore PID per il controllo di un servo motore. Sebbene tali aspetti sono stati affrontati in maniera più semplice, ciò non toglie che tali elementi possono essere utilizzati per realizzare dei sistemi più articolati. Partendo da questa base si può realizzare un sistema formato da **giunti**, che sono appunto rappresentati dai servo motori, e da **collegamenti**,che legano due giunti.

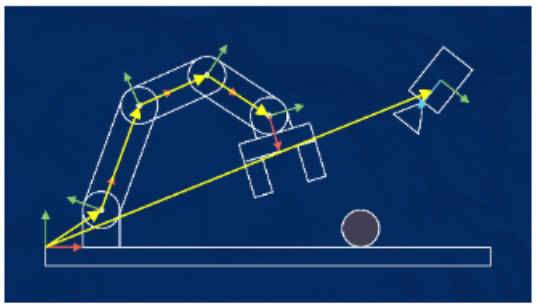


Figura 5.1. Sistema di giunti e collegamenti

Lo scopo di questo progetto è, anche, quello di mostrare l'applicazione dell'informatica in settori differenti dallo sviluppo software e l'Interdisciplinarità della materia.

Bibliografia

[1] atmel 2549 8-bit avr microcontroller atmega640 1280 1281 2560 2561 datasheet

[2] Dispensa Sistemi di Calcolo di Camil Demetrescu

[3] Silberschatz, Galvin, Gagne: Sistemi Operativi. Concetti ed Esempi

[4] Video Lezioni e slide prof.ssor Giorgio Grisetti

[5] L298 H Bridge Datasheet