SAPIENZA
UNIVERSITÀ DI ROMA

Implementazione del protocollo di comunicazione I^2C su piattaforma AVR per il controllo di un sistema multi-motore

Facoltà di Ingegneria dell'informazione, informatica e statistica

Corso di Laurea in Ingegneria Informatica ed Automatica

Candidato

Sara Attiani
Matricola 1893420

Relatore

Prof. Giorgio Grisetti

Anno Accademico 2021/2022

Implementazione del protocollo di comunicazione I^2C su piattaforma 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: attiani.1893420@studenti.uniroma1.it

"Cos'è la grandezza? Risponderò che è la capacità di vivere secondo i tre valori fondamentali di John Galt: ragione, scopo, autostima." -Ayn Rand

Indice

Capitolo 1

Introduzione

L'oggetto di questa trattazione interessa uno degli aspetti fondamentali del funzionamento di un calcolatore: come avviene lo scambio di informazioni tra le varie componenti di un sistema elettronico digitale.
Più in dettaglio, verrà mostrato il funzionamento e lo sviluppo di un protocollo di comunicazione per lo scambio di informazioni tra microcontrollori AVR su piattaforma Arduino, finalizzato al controllo simultaneo di motori in corrente continua.

Per cominciare, è opportuno definire cosa si intende quando ci si riferisce ad un "protocollo di comunicazione": esso è un insieme di regole che consente a due o più entità di un sistema di trasmettere informazioni, che vengono veicolate attraverso la variazione di un parametro fisico.
Il protocollo ha il compito di definire la sintassi, la semantica, la sincronizzazione della comunicazione e le possibili modalità di controllo e ripristino degli errori.

Vi sono pochi protocolli di comunicazione che dominano la scena nel settore dello sviluppo di microcontrollori e, più in generale, all'interno dell'architettura di un calcolatore, quali:

- I^2C (Inter Integrated Circuit): protocollo a bassa complessità, half-duplex, per una comunicazione sincrona tra molteplici entità in collegamento;
- SPI (Serial Peripheral Interface): protocollo a complessità crescente con il numero di entità in collegamento, full-duplex, per comunicazione sincrona.

A seconda dei requisiti del sistema, uno di questi protocolli è preferibile rispetto all'altro.

Un'altra modalità di comunicazione ampiamente utilizzata è quella che passa attraverso lo UART, un circuito che consente la ricezione/trasmissione seriale asincrona di dati tra due dispositivi.

Nell'ambito del progetto che si intende sviluppare verrà utilizzato il protocollo I^2C per la trasmissione di informazioni di controllo da un dispositivo "master" centralizzato a dei dispositivi "slave" in ascolto sulla linea; mentre l'input fornito dal computer client al dispositivo "master" verrà trasmesso via UART.

Maggiori dettagli verranno forniti:

- Nel Capitolo 2, nella quale verranno esposti gli strumenti necessari allo sviluppo del sistema proposto;
- Nel Capitolo 3, nella quale verrà descritta l'implementazione del sistema;
- Nel Capitolo 4, nella quale verranno mostrati esperimenti sul sistema e casi d'uso;

Infine, nel Capitolo 5 verranno tratte le conclusioni finali su quanto discusso.

Allo scopo di fornire un riferimento per ciò che verrà esposto in seguito, viene mostrata una panoramica generale sul sistema finale che verrà realizzato:

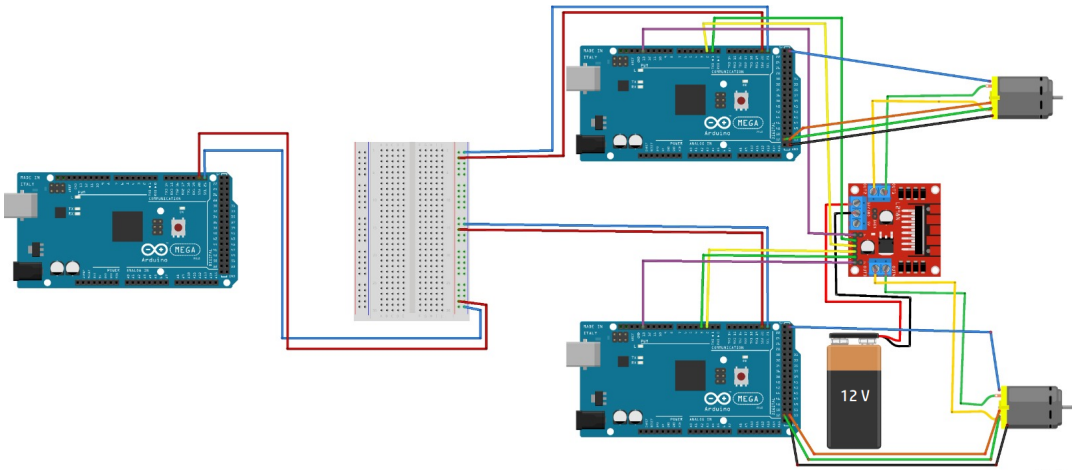

Figura 1.1. schema del progetto

La logica per il controllo della velocità dei motori dc non verrà affrontata nella presente trattazione. È possibile prendere visione del codice, comprensivo di tutti i moduli necessari per la realizzazione del sistema finale, presso il seguente repository git: https://github.com/GeoDimi99/AVR-Multi-Motor-Control.

Capitolo 2

Gli strumenti utilizzati

2.1 Il protocollo I^2C

Il protocollo I^2C costituisce di fatto uno standard mondiale, attualmente utilizzato da più di 50 rilevanti compagnie all'interno di circuiti integrati. È stato progettato all'inizio degli anni '80 dalla Philips Semiconductors (NXP Semiconductors dal 2006) per consentire una facile comunicazione tra i componenti che risiedono sulla stessa scheda, attraverso un semplice bus a due fili bidirezionale.

Tutti i dispositivi compatibili con il bus I^2C incorporano all'interno del chip un'interfaccia che gli consente di comunicare direttamente attraverso il bus.

Lo sviluppo di questo protocollo ha consentito di risolvere molti problemi che si era soliti incontrare durante la progettazione di circuiti per il controllo digitale.

2.1.1 Caratteristiche del protocollo

Presentiamo, innanzitutto, le caratteristiche fondamentali del protocollo:

- Sono necessarie solo due linee: una linea seriale per la trasmissione dei dati (SDA) ed una linea seriale per il clock del sistema (SCL);
- Ogni dispositivo connesso al bus è indirizzabile in maniera univoca a livello software;
- Il dispositivo che controlla la trasmissione può inviare dati sulla linea (controller-transmitter) o richiedere il trasferimento di dati da parte di un altro dispositivo (controller-receiver);
- È possibile avere diverse entità interfacciate sulla linea nel ruolo di controller, e per questo implementa meccanismi di rilevazione delle collisioni e dei conflitti di arbitraggio per prevenire la corruzione dei dati se due o più controller tentano simultaneamente di iniziare una trasmissione;
- È un protocollo seriale, orientato alla trasmissione di byte, bidirezionale;
- Presenta diverse modalità operative, contraddistinte dal bit-rate utilizzato (vedi Tabella 2.1);
- Possiede un sistema di filtraggio all'interno del chip per preservare l'integrità dell'informazione trasmessa sulla linea dei dati;

- Il numero di circuiti integrati che possono essere connessi sulla linea è limitato solamente dalla capacità della stessa.

Modalità	Bit rate	Bus
Standard-mode (Sm)	Fino a 100 kbit/s	Bidirezionale
Fast-mode (Fm)	Da 100 kbit/s fino a 400 kbit/s	Bidirezionale
Fast-mode Plus (Fm+)	Da 400 kbit/s fino ad 1 Mbit/s	Bidirezionale
High speed-mode (Hs)	Da 1 Mbit/s fino a 3.4 Mbit/s	Bidirezionale
Ultra fast-mode (UFm)	Fino a 5Mbit/s	Unidirezionale

Tabella 2.1. bit-rate sul bus

La Figura 2.1 mostra un esempio applicativo del bus I²C:

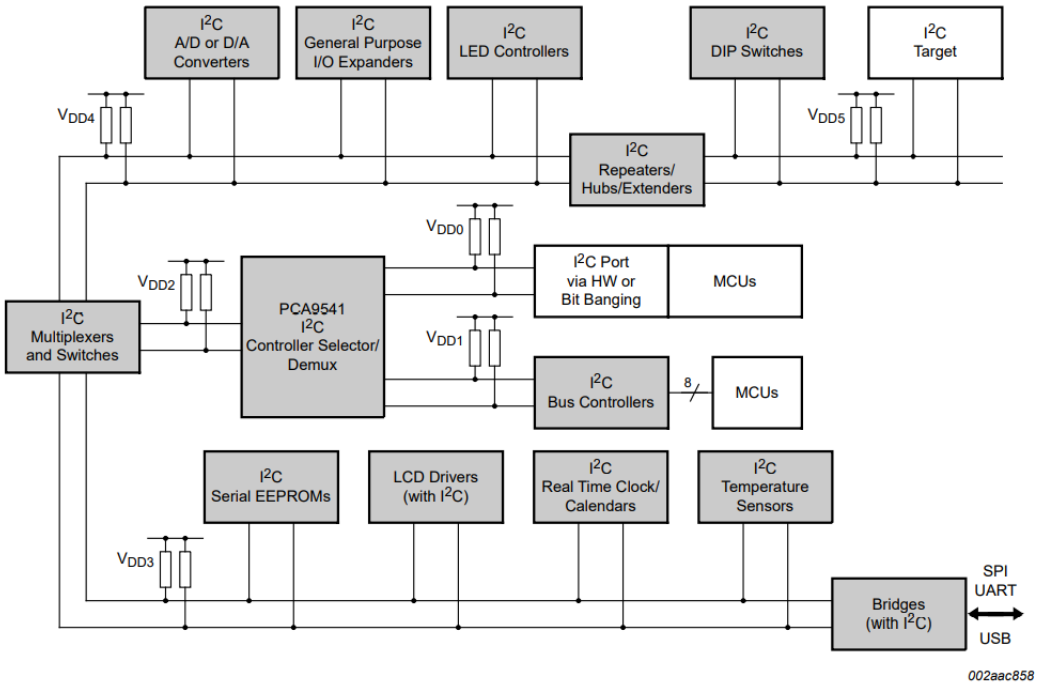

Figura 2.1. Applicazione del bus I²C

2.1.2 Terminologia utilizzata

Allo scopo di agevolare la spiegazione del protocollo, è conveniente definire a priori alcuni termini fondamentali:

- **Controller/Master**: dispositivo che assume, all'interno di una trasmissione da effettuarsi, il ruolo di controllore, avendo il compito di iniziare e concludere la trasmissione sulla linea, indirizzandola verso un dispositivo target;

- **Target/Slave**: dispositivo che assume un ruolo passivo all'interno di una trasmissione, avendo il compito di rispondere ai comandi del master;
- **Transmiter**: il dispositivo che invia i dati sul bus, non necessariamente coincidente con il Master (si parla di Master-Transmitter quando è lo stesso master ad inviare i dati sulla linea, viceversa si parla di Slave Transmitter);
- **Receiver**: il dispositivo che effettua la ricezione dei dati sulla linea (si parla di Master Receiver nel caso esso sia il master, viceversa si parla di Slave Receiver);
- **Start condition**: evento generato dall'entità Master che determina l'inizio di una trasmissione sulla linea, consentendo al dispositivo di acquisire il controllo esclusivo della stessa;
- **Stop condition**: evento generato dall'entità Master che determina la fine di una trasmissione sulla linea, consentendo al dispositivo di rilasciarne il controllo;
- **Repeated start condition**: condizione di Start che avviene a seguito di un'altra effettuata in precedenza e non seguita da una condizione di Stop, al fine di mantenere il controllo della linea da parte del Master;
- **Indirizzamento**: l'utilizzo di uno specifico indirizzo finalizzato a dichiarare in maniera univoca il target della trasmissione;
- **Trasmissione in broadcast**: trasmissione che ha per target tutti gli Slave collegati sulla linea, grazie all'impiego dell'indirizzo speciale per il broadcast, riconosciuto da ogni Slave.

2.1.3 Specifica del protocollo

Due linee, la linea dei dati SDA e la linea del clock SCL, consentono di realizzare le trasmissioni tra i dispositivi connessi al bus. Come si è detto, ogni dispositivo è identificato da un indirizzo univoco e può operare in modalità Trasmitter o Receiver a seconda delle esigenze contingenti.

Il protocollo consente la presenza di più dispositivi nel ruolo di Master affacciati sulla linea, come mostrato nella seguente configurazione:

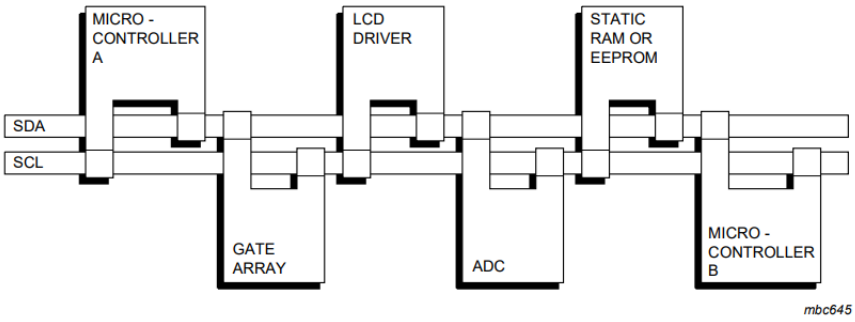

Figura 2.2. Esempio di configurazione di un bus I²C con due microcontrollori

La possibilità di connettere più di un dispositivo Master sul bus comporta il rischio che più di un dispositivo tenti di acquisire il controllo della linea per effettuare una trasmissione.

Linee SCL ed SDA

Per evitare la confusone che può derivare da un simile evento, è stato sviluppato un meccanismo di arbitraggio della linea che fa affidamento su una logica di cablaggio di tipo 'wired-AND', in cui ciascuna linea è mantenuta ad un livello logico alto quando il bus è libero.

Dunque, le linee SCL ed SDA sono entrambe linee bidirezionali, collegate ad una tensione di alimentazione positiva tramite una sorgente di corrente o un resistore pull-up. Gli stadi di uscita dei dispositivi collegati al bus devono avere un 'open drain' (per le porte di tipo CMOS) oppure un 'open-collector' (per porte di tipo TTL), che gli consenta di abbassare la tensione sulla linea. Tale sistema permette di realizzare la funzione AND cablata: la linea rimane in pull-up solo se nessuno dei dispositivi connessi presenta un percorso verso massa.

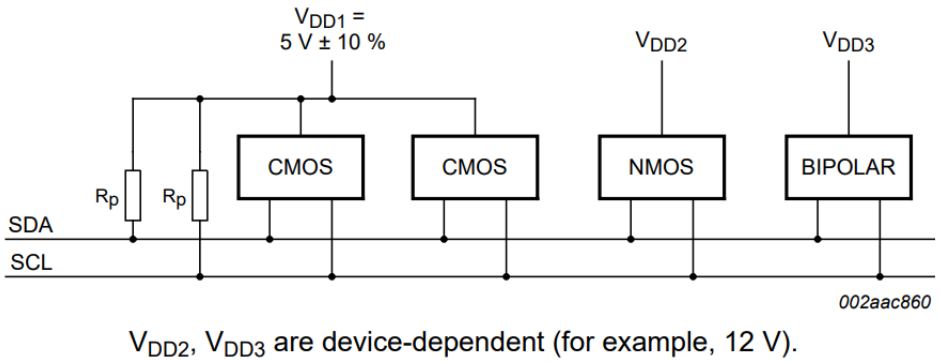

Figura 2.3. Dispositivi con diversi voltaggi interfacciati sullo stesso bus

Livelli logici

A causa dell'eterogeneità nelle tecnologie dei dispositivi che si possono connettere al bus (CMOS, NMOS, bipolari), i livelli logici '0' (LOW) e '1' (HIGH) non sono fissi, ma dipendono dal valore della tensione di alimentazione V_{DD}. Il livello logico basso V_{IL} è inteso come il 30% di V_{DD} (V_{IL}= $0.3V_{DD}$), mentre il livello logico alto corrisponde al 70% di V_{DD} (V_{IH}=$0.7V_{DD}$).

Validità dei dati

Il dato trasmesso sulla linea SDA deve rimanere stabile durante il periodo in cui il segnale sulla linea SCL è alto. Variazioni sulla linea dei dati devono verificarsi solo nell'intervallo di tempo in cui il segnale è basso. Ogni impulso del clock corrisponde al trasferimento di un bit sulla linea.

La Figura 2.4 mostra il corretto andamento dei segnali sulle linee SDA ed SCL durante un trasferimento:

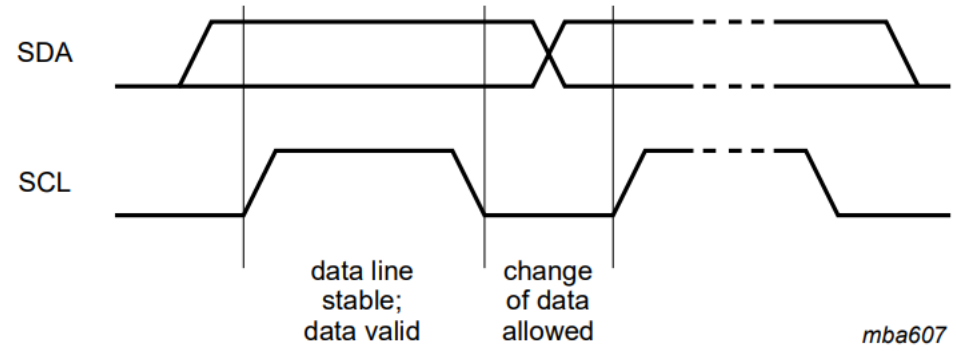

Figura 2.4. Trasferimento di bit su bus I²C

Condizioni di START e di STOP

Tutte le trasmissioni iniziano con una condizione di Start (S) e terminano con una condizione di Stop (P).

- Si verifica una condizione di **START** quando avviene un passaggio del segnale da alto (HIGH) a basso (LOW) sulla linea dei dati, in corrispondenza del segnale di clock alto (HIGH);
- Si verifica una condizione di **STOP** quando avviene un passaggio del segnale da basso (LOW) ad alto (HIGH) sulla linea dei dati, in corrispondenza del segnale di clock alto (HIGH);

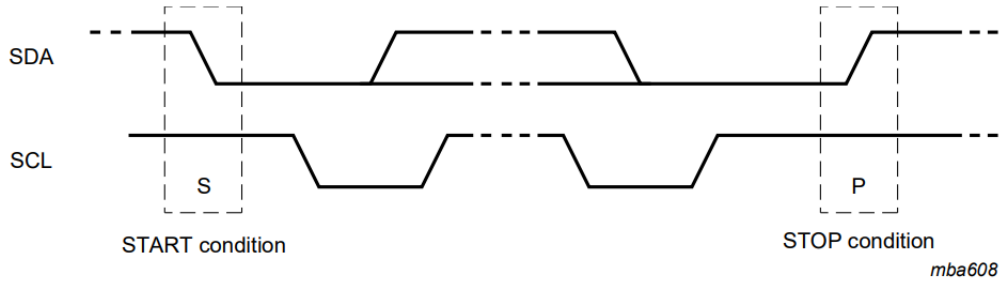

Figura 2.5. Condizioni di Start e Stop

Le condizioni di Start e di Stop devono sempre essere generate da un controller. Il bus viene considerato 'occupato' dopo che è stata inviata una condizione di Start, mentre viene considerato di nuovo libero in seguito ad una condizione di Stop.

Un dispositivo può mantenere il controllo sul bus inviando uno Start ripetuto (Repeated-Start) (Sr), rimandando così l'invio della condizione di Stop ad un momento successivo. Al di là della differenza concettuale che sussiste tra una condizione di Start ed una condizione di Repeated Start, non vi è alcuna differenza nel modo in cui queste vengono realizzate e nell'effetto che sortiscono.

Il rilevamento delle condizioni di Start e di Stop da parte dei dispositivi interfacciati sul bus risulta semplice se questi possiedono una specifica interfaccia hardware dedicata. Al contrario, i microcontrollori che non possiedono tale interfaccia hanno

necessità di campionare almeno due volte il canale SDA, nell'intervallo in cui il segnale in SCL è alto, per rilevare una eventuale transizione di tensione.

Formato dei byte

Ogni dato trasferito sulla linea deve avere la dimensione di un byte, sebbene non vi sia nessuna restrizione circa numero di byte che possono essere inviati durante un trasferimento. Ad ogni byte segue immediatamente un bit di Acknowledge.
I bit vengono trasmessi a partire dal più significativo (MSB).

Se il dispositivo target non può ricevere o trasmettere altre informazioni prima di aver eseguito qualche altro tipo di compito, ad esempio occuparsi di gestire un'interruzione, questo ha la possibilità di mantenere la linea SCL abbassata in modo da forzare il dispositivo master a ritardare l'invio/ricezione di ulteriori dati. Il trasferimento può riprendere nel momento in cui il dispositivo slave rilascia la linea del clock. Questo procedimento prende il nome di "clock stretching", e consente di adattare la velocità di una trasmissione alle esigenze specifiche di ciascun dispositivo.

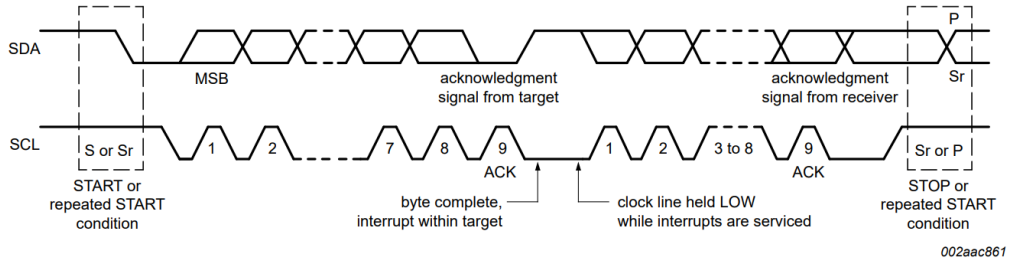

Figura 2.6. Trasferimento dati sul bus I²C

Acknowledge (ACK) e Not-Acknowledge (NACK)

L'invio del bit di Acknowledge (ACK) sussegue ogni byte trasmesso sulla linea. Esso consente all'entità ricevente di segnalare al trasmittente che la ricezione del byte è avvenuta con successo e, di conseguenza, un altro byte può essere trasmesso sulla linea.

La trasmissione di un ACK avviene nel seguente modo: il dispositivo trasmittente rilascia la linea SDA durante la pulsazione del clock che segue l'invio dell'ultimo bit del dato, in modo da consentire al ricevente di abbassare il segnale sulla linea, che deve rimanere basso (LOW) durante il periodo il cui la linea SCL è alta (HIGH).

Al contrario, se la linea non viene mantenuta bassa dal receiver durante in no-no impulso di clock, si ottiene la trasmissione di un Not-Acknowledge bit (NACK), che segnala una condizione di errore nel trasferimento.

Vi sono cinque scenari possibili che conducono alla trasmissione di un NACK:

1. Non è presente nessuno slave in ascolto sulla linea con l'indirizzo utilizzato come target;
2. Il dispositivo indirizzato non è pronto ad iniziare una comunicazione con il controller;

3. Durante il trasferimento, il ricevente ottiene dati o comandi che non comprende;
4. Durante il trasferimento, ricevente non è più in grado di ricevere nessun altro dato;
5. Un dispositivo in modalità Master-Receiver deve segnalare la fine di un trasferimento al dispositivo trasmettente.

Sincronizzazione dei clock

Come è già stato evidenziato, due master possono tentare l'inizio di una trasmissione contemporaneamente, motivo per cui deve esistere un metodo per decidere quale dispositivo ottiene il controllo del bus. Ciò avviene grazie alla sincronizzazione dei clock e al meccanismo di arbitraggio.

Il primo è reso possibile dalle connessioni 'wired AND' sulla linea di clock SCL: questa viene mantenuta abbassata (LOW) per l'intervallo di tempo più elevato in cui un controller mantiene il segnale che si interfaccia sulla linea basso.

Gli altri controller, se presentano un segnale di clock alto (HIGH) in corrispondenza di tale intervallo, rimangono in stato di attesa (HIGH-wait state) fintanto che la linea SCL viene mantenuta abbassata.

Quando tutti i controller rilasciano la linea SCL, il segnale viene ripristinato ad un livello alto (HIGH) e si manterrà tale fintanto che un controller, per primo, non abbasserà nuovamente la linea.

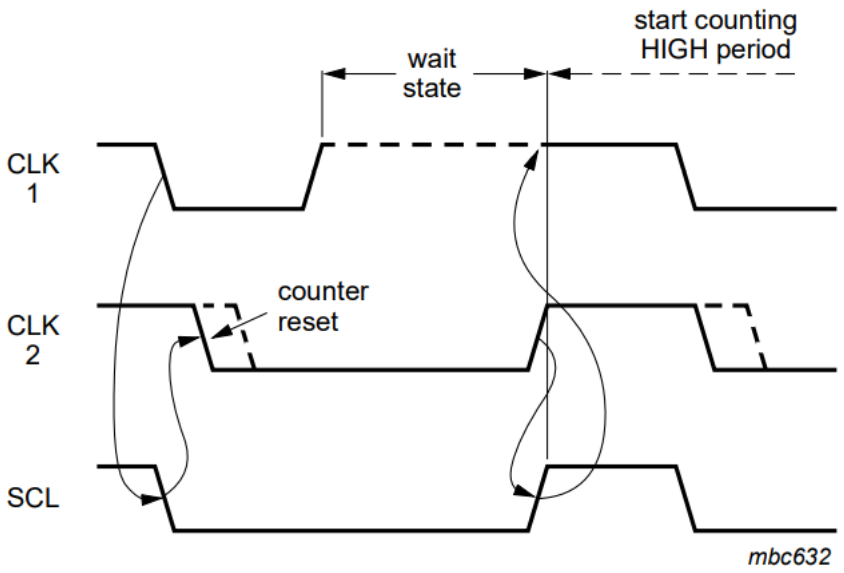

Figura 2.7. Sincronizzazione del clock

In definitiva, la linea SCL viene mantenuta abbassata dal controller con il più elevato intervallo di clock basso (LOW), mentre l'intervallo in cui la linea rimane alzata è determinato dal controller con il più ristretto intervallo di clock alto (HIGH).

Arbitraggio

L'arbitraggio sulla linea dei dati consente di stabilire quale dispositivo mantiene il controllo del bus, nel caso in cui diversi controller tentino contemporaneamente di portare a termine una trasmissione. In maniera speculare al meccanismo di sincronizzazione del clock, l'arbitraggio è reso possibile dalle connessioni 'wired AND' sulla linea SDA.

Due controller, infatti, possono generare una condizione di Start nello stesso momento, il che risulta in una condizione legale per entrambi nell'iniziare una trasmissione. Il meccanismo di arbitraggio procede 'bit-by-bit': durante la trasmissione di ogni bit, ciascun controller verifica, in un momento in cui la linea SCL è alta (HIGH), che il dato trasmesso sulla linea sia congruente con quello che esso stesso ha inviato. Quando accade che un controller osserva un dato sulla linea SDA diverso da quello atteso, il dispositivo apprende di aver perso il controllo del bus e rinuncia alla trasmissione dei dati. Questo accade, nella pratica, se il controller aveva tentato di inviare un '1' (HIGH) ma legge uno '0' (LOW) sulla linea. Almeno un controller riuscirà a portare a termine la trasmissione con successo.

A questo punto risultano pienamente apprezzabili i vantaggi della connessione di tipo 'wired AND': questa consente di preservare l'integrità dei dati trasmessi con successo sulla linea, tanto che due controller possono completare contemporaneamente un intero trasferimento, fintanto che i dati trasmessi sono identici.

La Figura 2.8 mostra il meccanismo di arbitraggio in azione per due controller che tentano un trasferimento:

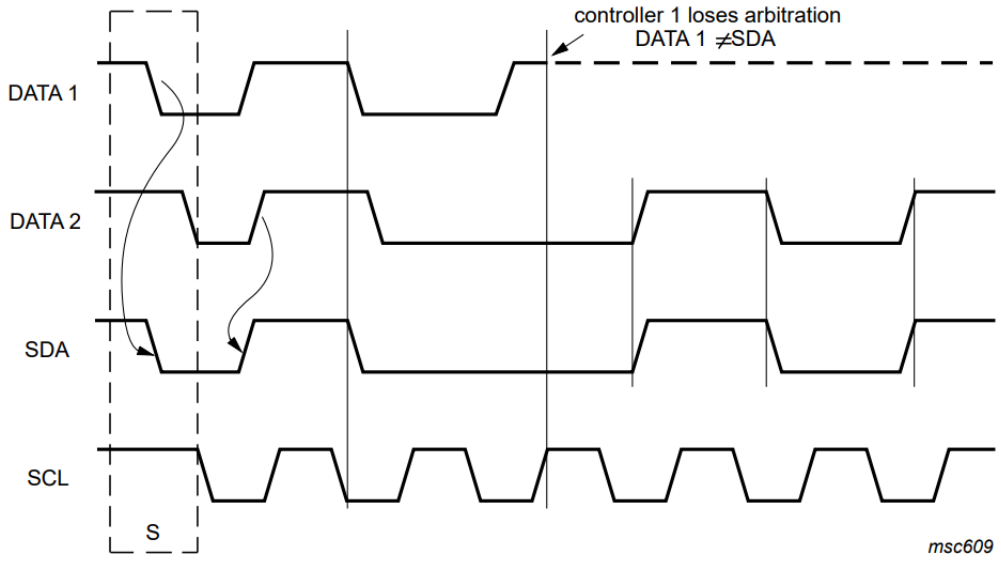

Figura 2.8. Arbitraggio tra due entità controller

Dalla procedura di arbitraggio descritta si evince che non esiste un controller centrale, o con maggiore priorità sul trasferimento.

Si verifica una condizione di indefinizione nel momento in cui un controller invia una condizione di Repeated-Start o di Stop mentre un altro controller sta effettuando il trasferimento di dati. Dunque, il comportamento del sistema risulta indefinito nei seguenti casi:

1. Il controller 1 invia un Repeated-Start ed il controller 2 invia un bit di dato;
2. Il controller 1 invia uno Stop ed il controller 2 invia un bit di dato;
3. Il controller 1 invia un Repeated-Start ed il controller 2 invia uno Stop.

Indirizzamento e R/W bit

Un trasferimento completo di dati segue lo schema riportato in Figura 2.10:

1. A seguito di una condizione di Start, si verifica il trasferimento di un 'address byte', composto -per i primi 7 bit- dall'indirizzo del target della trasmissione e -per l'ultimo bit- da un flag di direzione (R/W bit), che viene impostato a '0' se il controller desidera trasmettere (Write) dati sulla linea, mentre è impostato a '1' se desidera ricevere (Read) dati sulla linea;
2. Dopo la trasmissione dell'indirizzo e del R/W bit, se si riceve un ACK da parte del dispositivo target, può iniziare il trasferimento dei data byte sulla linea, ciascuno dei quali viene susseguito da un ACK bit se il target è disponibile a ricevere un altro dato;
3. Terminata la trasmissione, il controller può mantenere il possesso del bus inviando una condizione di Repeated Start, oppure rilasciare il controllo inviando una condizione di Stop.

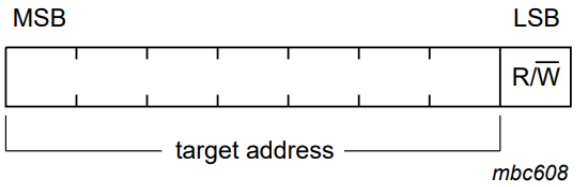

Figura 2.9. Il primo byte dopo una condizione di Start

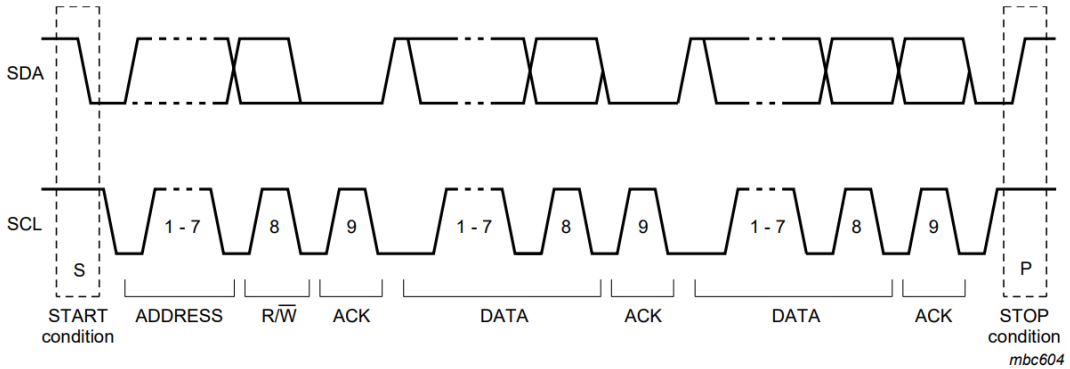

Figura 2.10. Trasferimento dati completo

Indirizzo broadcast

È possibile effettuare una trasmissione che abbia come target tutti i dispositivi in ascolto sul bus. Ogni dispositivo può rispondere positivamente (ACK) alla chiamata del controller, oppure negativamente (NACK). Il controller, in presenza di un riscontro positivo sulla linea, effettua la trasmissione, ma non è in grado di sapere quanti dispositivi si sono correttamente predisposti alla ricezione, se ve ne sono molteplici.

Concludiamo la specifica del protocollo con alcune note che si ritengono rilevanti, lasciando al lettore il compito di approfondire aspetti più complessi del protocollo quali l'indirizzamento a 10 bit (che consente evidentemente a più dispositivi di interfacciarsi sul bus), la specifica degli indirizzi riservati e le modalità di funzionamento che superano quella Standard, che abbiamo finora presentato. Le informazioni tralasciate all'interno della presente trattazione non sono rilevanti per il sistema che ci si prefigge di sviluppare.

Note finali:

- I dispositivi compatibili con I²C devono immediatamente resettare la loro logica sul bus nel momento in cui ricevono una condizione di Start o di Repeated-Start, in modo tale da predisporsi a ricevere l'indirizzo target della comunicazione;
- Una condizione di Start seguita immediatamente da una di Stop (messaggio vuoto) è illegale per il protocollo, nonostante molti dispositivi sono stati sviluppati per lavorare correttamente anche in presenza di tale condizione;
- È bene notare che un indirizzamento a 7 bit consente a non più di 127 dispositivi (considerato l'indirizzo broadcast) di interfacciarsi sul bus.

2.1.4 Esempi di utilizzo del protocollo

Come si è detto, il protocollo I²C è di fatto uno standard mondiale. Grazie alla sua versatilità, viene utilizzato per varie architetture di controllo, tra le quali si annoverano:

- Il 'System Management Bus' (SMBus): letteralmente il "bus per il controllo di sistema", utilizzato per la comunicazione a bassa velocità tra le periferiche presenti sulla scheda madre;
- Il 'Power Management Bus' (PMBus): variante del SMBus, in cui il target è costituito specificatamente dai chip relativi all'energia, come i sottosistemi per la gestione delle batterie ricaricabili nei portatili;
- L' 'Intelligent Platform Management Interface' (IPMI): è un insieme di specifiche -guidato da Intel- sull'interfaccia del calcolatore che fornisce funzionalità di gestione e monitoraggio indipendentemente dalla CPU , dal firmware (BIOS o UEFI) e dal sistema operativo;
- Il 'Display Data Channel' (DDC): canale per la comunicazione digitale tra il display del computer e la scheda grafica;
- L' 'Advanced Telecom Computing Architecture' (ATCA): l' open-standard più utilizzato per le infrastrutture di telecomunicazioni globali.

2.2 L'interfaccia TWI

I microcontrollori Atmel (AVR) forniscono l'interfaccia TWI (2-wire interface) per consentire l'implementazione del protocollo di comunicazione I^2C descritto in precedenza. Attraverso tale interfaccia è possibile interconnettere un dispositivo AVR ad un altro, realizzando un sistema distribuito come quello che verrà mostrato all'interno di questa trattazione.

2.2.1 Overview del modulo TWI

Il modulo TWI comprende diversi sotto-moduli, come mostrato in Figura 2.11.Tutti i registri evidenziati con una linea di maggiore spessore sono accessibili attraverso il data bus AVR.

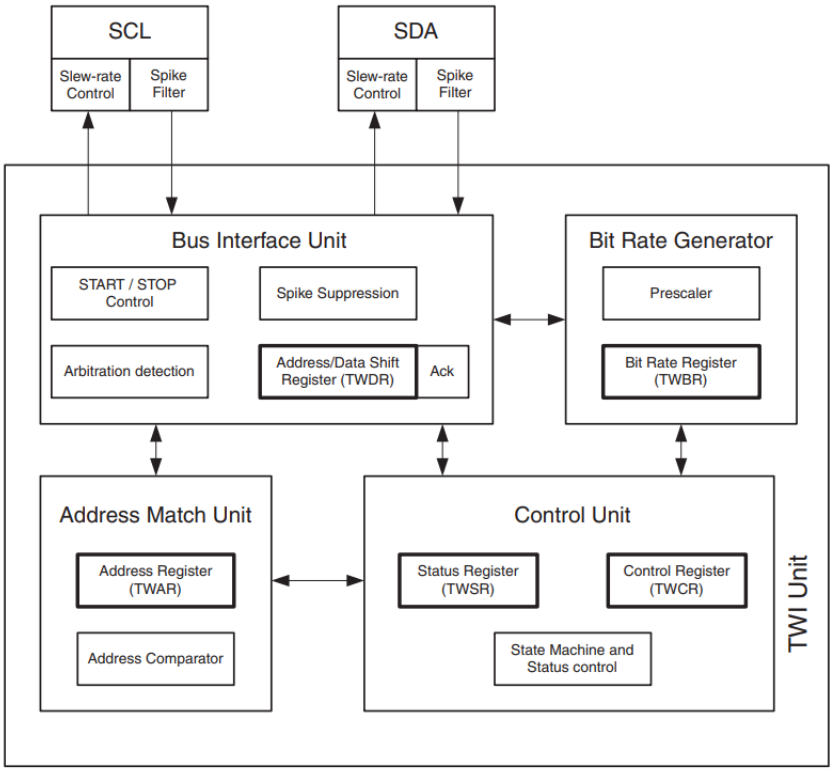

Figura 2.11. Overview del modulo TWI

I pin SCL ed SDA

I pin SCL ed SDA interfacciano il modulo TWI con il resto microcontrollore AVR (MCU). I driver in output contengono un limitatore di velocità in modo da conformarsi alla specifica del modulo TWI. D'altro canto, l'input verso il sistema presenta un'unità che effettua il filtraggio dei picchi inferiori ai 50 ns. I pull-up interni possono essere abilitati settando i bit della porta del dispositivo corrispondenti ai pin SCL ed SDA.

Bit Rate Generator

Questa unità controlla il periodo del segnale di clock (SCL) quando si opera in modalità 'Master'. Per impostare la frequenza del segnale SCL, occorre settare il 'Bit Rate Register' (TWBR) ed i bit del Prescaler nello 'Status Register' (TWSR). Le operazioni di un dispositivo operante come 'Slave' non dipendono dal bit-rate o dalle impostazioni del Prescaler, ma la frequenza di clock della CPU del dispositivo deve essere almeno sedici volte maggiore della frequenza di SCL.

La frequenza del segnale SCL viene generata secondo la seguente equazione:

$$SCLfrequency = \frac{CPU\ clock\ frequency}{16 + 2 \cdot (TWBR) \cdot 4^{TWPS}} \tag{2.1}$$

ove

- TWBR: valore del registro 'TWI Bit Rate Register';
- TWPS: valore dei bit di prescaling nel registro di stato 'TWI Status Register'.

Bus Interface Unit

All'interno della 'Bus Interface Unit' è presente uno shift register per contenere il dato o l'indirizzo (TWDR) trasmesso/ricevuto, un controller per la generazione ed il rilevamento delle condizioni di START/STOP ed un hardware per tenere traccia dell'arbitraggio del sistema. In aggiunta al registro TWDR ad 8-bit, è presente un registro contenente il bit di (N)ACK trasmesso o ricevuto. Quest'ultimo non è direttamente accessibile dal software applicativo, ma può essere settato manipolando il registro di controllo 'TWI Control Register' (TWCR) e determinato attraverso il valore del registro di stato TWSR.

Address Match Unit

Questa unità controlla se l'indirizzo ricevuto corrisponde l'indirizzo a 7 bit presente nel registro 'TWI Address Register' (TWAR). Se è stato abilitato il riconoscimento dell'indirizzo broadcast, asserendo il bit 'TWI General Call Recognition Enable' (TWGCE), anch'esso verrà riconosciuto dal dispositivo. Se si verifica un match dell'indirizzo, l'unità di controllo viene informata, consentendo di eseguire una corretta azione di risposta.

L'Address Match Unit è in grado di effettuare la comparazione degli indirizzi anche quando il microcontrollore AVR è in 'sleep mode'. Questo verrà eventualmente "risvegliato" se si verifica un match address. Se un'altra interruzione (ad esempio INT0) si verifica durante l'address match e risveglia l'MCU, il modulo TWI interrompe l'operazione e ritorna in stato di riposo.

L'Unità di Controllo

L'unità di controllo ('Control Unit') monitora il bus TWI e genera risposte sulla base dell'impostazione del registro di controllo TWCR. Quando si verifica un evento sul bus TWI che richiede l'attenzione dell'applicazione, si asserisce il 'TWI Interrupt Flag' (TWINT). Al prossimo ciclo di clock, il registro di stato TWSR verrà aggiornato con un codice di stato che identifica l'evento verificatosi. Il registro TWSR contiene informazioni di stato rilevanti solo nel momento in cui si asserisce il

flag TWINT. In ogni altro momento, il registro TWSR contiene un codice di stato speciale che indica che non ci sono informazioni rilevanti disponibili. Finché il flag TWINT rimane settato, il segnale SCL è mantenuto basso (LOW). Questo consente al software applicativo di gestire l'interruzione verificatasi prima di riprendere la trasmissione.

Il flag TWINT viene settato nei seguenti contesti:

- In seguito alla trasmissione di una condizione di START/REPEATED START;
- In seguito alla trasmissione di un indirizzo seguito dal bit R/W (SLA + R/W);
- In seguito alla trasmissione di un dato;
- In seguito alla perdita di arbitraggio sul bus;
- In seguito al riconoscimento di un indirizzo ricevuto;
- In seguito alla ricezione di un dato;
- In seguito alla ricezione di una condizione di STOP/REPEATED START come slave;
- In seguito al verificarsi di un errore dovuto ad una condizione di START/STOP illegale.

2.2.2 Descrizione dei registri

TWBR – TWI Bit Rate Register

Bit	7	6	5	4	3	2	1	0	
(0xB8)	TWBR7	TWBR6	TWBR5	TWBR4	TWBR3	TWBR2	TWBR1	TWBR0	TWBR
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 2.12. Registro TWBR

Il registro TWBR seleziona il fattore di divisione per il generatore del bit-rate. Il generatore del bit rate è un divisore di frequenza per il segnale SCL.

TWCR – TWI Control Register

Bit	7	6	5	4	3	2	1	0	
(0xBC)	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	–	TWIE	TWCR
Read/Write	R/W	R/W	R/W	R/W	R	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 2.13. Registro TWCR

Il registro TWCR controlla le operazioni del modulo TWI. Viene utilizzato per abilitare il modulo TWI, per inviare una condizione di START/STOP sul bus, per generare (N)ACK e per fermare il bus mentre il dato da trasmettere si trova nel registro TWDR. Inoltre, indica il verificarsi di una collisione se si tenta di inserire un dato nel registro TWDR quando il registro non è accessibile.

Si descrivono in dettaglio i bit che compongono il registro TWCR:

- **Bit 7 – TWINT: TWI Interrupt Flag**

  Il bit TWINT viene settato dall'hardware quando il modulo TWI ha terminato il suo lavoro ed aspetta una risposta da parte del software applicativo. Se il bit delle interruzioni in SREG ed il bit TWIE in TWCR sono settati, l'MCU salterà nell'Interrupt Vector del modulo TWI. Mentre il bit TWINT è settato, il segnale di clock SCL viene mantenuto basso (LOW). Il flag TWINT deve essere ripristinato a livello software scrivendovi al suo interno un '1' logico, poiché questo non viene ripulito dall'hardware quando viene eseguito l'interrupt handler. Il ripristino del flag TWINT comporta la ripresa delle operazioni del modulo TWI.

- **Bit 6 – TWEA: TWI Enable Acknowledge Bit**

  Il bit TWEA controlla la generazione dell'impulso per l'Acknowledge. Se impostato ad '1', viene generato un impulso per l'ACK quando si verifica la condizione opportuna. Viceversa, impostando il bit TWEA a '0', un dispositivo può essere virtualmente disconnesso dal bus per un certo periodo di tempo.

- **Bit 5 – TWSTA: TWI START Condition Bit**

  Il bit TWSTA viene impostato ad '1' da parte dell'applicazione quando si desidera assumere il ruolo di Master sul bus. L'hardware TWI controlla se il bus è disponibile e genera una condizione di START in caso affermativo. Al contrario, se il bus risulta occupato da un'altra trasmissione, l'hardware aspetta fino a che non viene rilevata una condizione di STOP ed in seguito genera una START. Il bit TWSTA deve essere ripulito dal software dopo che la condizione di START è stata inviata.

- **Bit 4 – TWSTO: TWI STOP Condition Bit**

  Il bit TWSTO genera una condizione di STOP sul bus TWI se impostato ad '1'. Quando la condizione di STOP è stata inviata sul bus, TWSTO viene ripulito in automatico.

- **Bit 3 – TWWC: TWI Write Collision Flag**

  Il bit TWWC viene settato nel momento in cui si tenta di scrivere nel registro per il dato TWDR quando TWINT non è settato. TWWC viene ripristinato scrivendo nel registro TWDR quando TWINT è settato.

- **Bit 2 – TWEN: TWI Enable Bit**

  Il bit TWEN abilita le operazioni TWI ed attiva l'interfaccia. Quando TWEN è impostato ad '1', il modulo TWI prende il controllo sui pin I/O connessi ad SCL ed SDA.

- **Bit 1 – Res: Reserved Bit**

  Questo bit è riservato e viene sempre letto con valore '0'.

- **Bit 0 – TWIE: TWI Interrupt Enable**

  Il bit TWIE, se impostato ad '1', abilita le interruzioni TWI se il bit 'I' del registro SREG e il bit TWINT sono settati.

TWSR – TWI Status Register

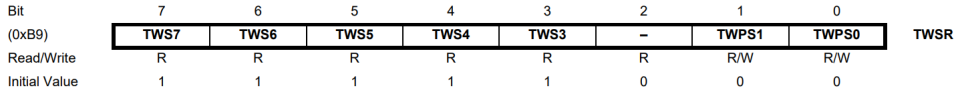

Figura 2.14. Registro TWSR

Il registro TWSR contiene informazioni sullo stato del sistema ed i bit del Prescaler.

Si descrivono in dettaglio i bit che compongono il registro TWSR:

- **Bits 7:3 – TWS: TWI Status**

  Questi cinque bit rispecchiano lo stato del modulo TWI e del bus seriale. I codici di stato vengono descritti in seguito.

- **Bit 2 – Res: Reserved Bit**

  Questo bit è riservato e viene sempre letto con valore '0'.

- **Bits 1:0 – TWPS: TWI Prescaler Bits**

  I primi due bit meno significativi del registro TWSR ed impostano un valore di prescaling diverso a seconda della combinazione dei due, come mostrato nella Tabella 2.2.

TWPS1	TWPS0	Prescaler value
0	0	1
0	1	4
1	0	16
1	1	64

Tabella 2.2. TWI Bit Rate Prescaler

TWDR – TWI Data Register

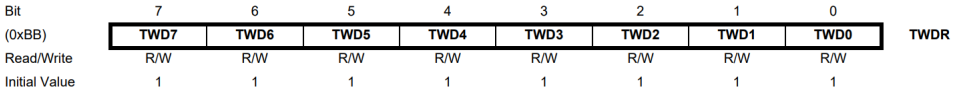

Figura 2.15. Registro TWDR

Il registro TWDR contiene il prossimo dato da trasmettere se in 'Transmit mode'. Viceversa, contiene l'ultimo byte ricevuto se in 'Receive mode'.

TWAR – TWI (Slave) Address Register

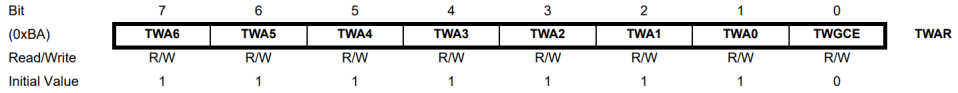

Figura 2.16. Registro TWAR

Il registro TWAR memorizza l'indirizzo del dispositivo Slave nei suoi 7 bit più significativi. Il bit meno significativo, se impostato ad '1', abilita il riconoscimento dell'indirizzo broadcast.

TWAMR – TWI (Slave) Address Mask Register

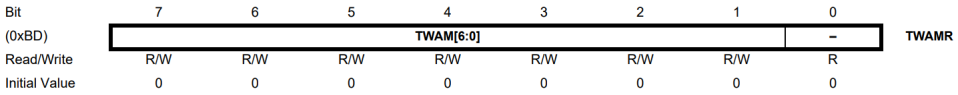

Figura 2.17. Registro TWAMR

I 7 bit più significativi del registro TWAMR possono essere settati in maniera tale da mascherare, dunque disabilitare, il bit corrispondente nel registro TWAR, facendo in modo che la logica di comparazione tra l'indirizzo ricevuto e l'indirizzo presente in TWAR ignori i bit mascherati.

2.2.3 Utilizzo dell'interfaccia TWI

Il modulo AVR TWI è orientato alla trasmissione di byte e gestito attraverso interruzioni. In tal modo, il software applicativo è libero di eseguire altre operazioni durante un trasferimento. Occorre notare che il bit TWIE nel registro di controllo TWCR, insieme al bit per le interruzioni globali 'Global Interrupt Enable' nel registro SREG, consentono all'applicazione di decidere se l'asserzione del flag TWINT deve generare oppure no una richiesta di gestione dell'interruzione. Se il bit TWIE è impostato a '0' l'applicazione deve controllare in polling lo stato del flag TWINT per rilevare l'occorrenza di un evento sul bus.

Quando il bit TWINT è settato, il modulo TWI ha concluso un'operazione ed è in attesa di una risposta da parte dell'applicazione. In questo caso, come è già stato evidenziato, il registro di stato TWSR indica lo stato corrente del sistema TWI. L'applicazione software può decide il comportamento del modulo TWI durante il prossimo ciclo di clock manipolando il registro di controllo TWCR ed il registro contenente il data byte TWDR.

La Figura 2.18 mostra un esempio di come l'applicazione può interfacciarsi con l'hardware del modulo TWI. Nell'esempio riportato, un dispositivo Master trasmette un singolo byte di dato ad un dispositivo Slave.

1. L'applicazione imposta il registro TWCR per iniziare la trasmissione;
2. TWINT viene settato, il codice di stato in TWSR indica che una condizione di START è stata inviata;

3. L'applicazione controlla il codice di stato per verificare che una condizione di START sia stata inviata, carica il byte SLA+W in TWDR, imposta opportunamente il registro TWCR assicurandosi che TWINT sia impostato ad '1' e che TWSTA sia impostato a '0';

4. TWINT viene settato, il codice di stato indica che un byte di tipo SLA+W è stata inviato, ed un ACK è stato ricevuto;

5. L'applicazione controlla il codice di stato per verificare che l'indirizzo ed il bit W siano stati inviati ed un ACK sia stato ricevuto, carica il dato in TWDR, imposta opportunamente il registro TWCR assicurandosi che TWINT sia impostato ad '1';

6. TWINT viene settato, il codice di stato indica che un data byte è stato inviato, ed un ACK è stato ricevuto;

7. L'applicazione controlla il codice di stato per verificare che il dato sia stato inviato ed un ACK ricevuto, imposta opportunamente il registro TWCR per inviare una condizione di STOP, assicurandosi che TWINT sia impostato ad '1'.

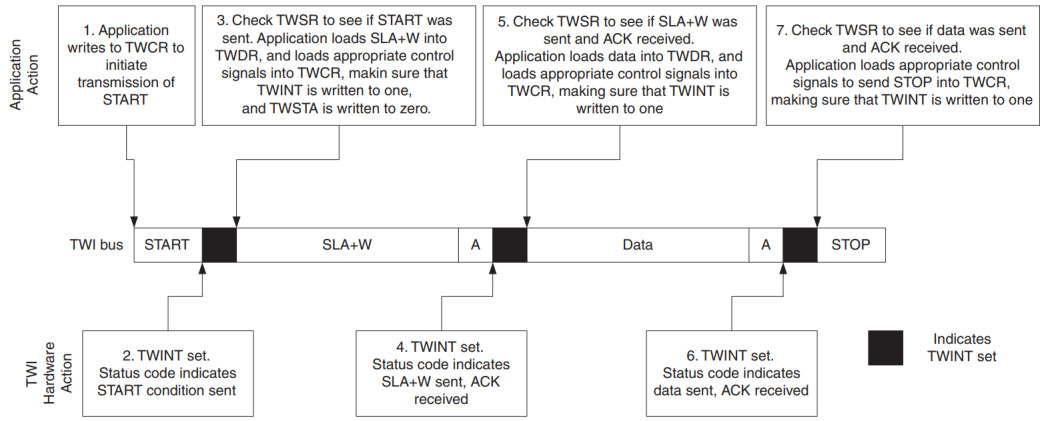

Figura 2.18. Esempio di una tipica trasmissione TWI

Attraverso questo semplice esempio è possibile estrapolare dei principi generali che interessano tutte le trasmissioni TWI:

- Quando il modulo TWI ha finito un'operazione ed aspetta una risposta da parte dell'applicazione, il flag TWINT è settato e la linea SCL rimane abbassata fino a quando il flag non viene ripulito;

- Quando il flag TWI è settato, l'applicazione deve aggiornare i registri TWI in maniera consistente con l'azione desiderata durante il successivo ciclo di clock;

- Dopo che i registri sono stati aggiornati ed altri task in sospeso sono stati eseguiti, viene impostato il registro di stato TWCR così da generare un nuovo evento.

2.2.4 Modalità di trasmissione

Il modulo TWI può operare in una delle seguenti quattro modalità:

- Master-Transmitter;
- Master-Receiver;
- Slave-Receiver;
- Slave-Transmitter.

È evidente che ad un dispositivo in modalità Master-Transmitter dovrà corrispondere uno in modalità Slave-Receiver, così come un dispositivo in modalità Master-Receiver sarà accoppiato con uno in modalità Slave-Transmitter.

Vengono in seguito riportate, dal datasheet della Atmel ATmega640/V-1280/V-1281/V-2560/V-2561/V, le tabelle che per ciascuna delle quattro modalità presentate riportano i codici di stato che è possibile incontrare durante lo svolgimento della trasmissione, e le azioni di gestione opportune.

Status Code (TWSR) Prescaler Bits are 0	Status of the 2-wire Serial Bus and 2-wire Serial Interface Hardware	Application Software Response					Next Action Taken by TWI Hardware
		To/from TWDR	To TWCR				
			STA	STO	TWINT	TWEA	
0x08	A START condition has been transmitted	Load SLA+W	0	0	1	X	SLA+W will be transmitted; ACK or NOT ACK will be received
0x10	A repeated START condition has been transmitted	Load SLA+W or	0	0	1	X	SLA+W will be transmitted; ACK or NOT ACK will be received
		Load SLA+R	0	0	1	X	SLA+R will be transmitted; Logic will switch to Master Receiver mode
0x18	SLA+W has been transmitted; ACK has been received	Load data byte or	0	0	1	X	Data byte will be transmitted and ACK or NOT ACK will be received
		No TWDR action or	1	0	1	X	Repeated START will be transmitted
		No TWDR action or	0	1	1	X	STOP condition will be transmitted and TWSTO Flag will be reset
		No TWDR action	1	1	1	X	STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset
0x20	SLA+W has been transmitted; NOT ACK has been received	Load data byte or	0	0	1	X	Data byte will be transmitted and ACK or NOT ACK will be received
		No TWDR action or	1	0	1	X	Repeated START will be transmitted
		No TWDR action or	0	1	1	X	STOP condition will be transmitted and TWSTO Flag will be reset
		No TWDR action	1	1	1	X	STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset
0x28	Data byte has been transmitted; ACK has been received	Load data byte or	0	0	1	X	Data byte will be transmitted and ACK or NOT ACK will be received
		No TWDR action or	1	0	1	X	Repeated START will be transmitted
		No TWDR action or	0	1	1	X	STOP condition will be transmitted and TWSTO Flag will be reset
		No TWDR action	1	1	1	X	STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset
0x30	Data byte has been transmitted; NOT ACK has been received	Load data byte or	0	0	1	X	Data byte will be transmitted and ACK or NOT ACK will be received
		No TWDR action or	1	0	1	X	Repeated START will be transmitted
		No TWDR action or	0	1	1	X	STOP condition will be transmitted and TWSTO Flag will be reset
		No TWDR action	1	1	1	X	STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset
0x38	Arbitration lost in SLA+W or data bytes	No TWDR action or	0	0	1	X	2-wire Serial Bus will be released and not addressed Slave mode entered
		No TWDR action	1	0	1	X	A START condition will be transmitted when the bus becomes free

Figura 2.19. Codici di stato in modalità Master-Transmitter

Status Code (TWSR) Prescaler Bits are 0	Status of the 2-wire Serial Bus and 2-wire Serial Interface Hardware	Application Software Response					Next Action Taken by TWI Hardware
		To/from TWDR	To TWCR				
			STA	STO	TWINT	TWEA	
0x08	A START condition has been transmitted	Load SLA+R	0	0	1	X	SLA+R will be transmitted ACK or NOT ACK will be received
0x10	A repeated START condition has been transmitted	Load SLA+R or	0	0	1	X	SLA+R will be transmitted ACK or NOT ACK will be received
		Load SLA+W	0	0	1	X	SLA+W will be transmitted Logic will switch to Master Transmitter mode
0x38	Arbitration lost in SLA+R or NOT ACK bit	No TWDR action or	0	0	1	X	2-wire Serial Bus will be released and not addressed Slave mode will be entered
		No TWDR action	1	0	1	X	A START condition will be transmitted when the bus becomes free
0x40	SLA+R has been transmitted; ACK has been received	No TWDR action or	0	0	1	0	Data byte will be received and NOT ACK will be returned
		No TWDR action	0	0	1	1	Data byte will be received and ACK will be returned
0x48	SLA+R has been transmitted; NOT ACK has been received	No TWDR action or	1	0	1	X	Repeated START will be transmitted
		No TWDR action or	0	1	1	X	STOP condition will be transmitted and TWSTO Flag will be reset
		No TWDR action	1	1	1	X	STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset
0x50	Data byte has been received; ACK has been returned	Read data byte or	0	0	1	0	Data byte will be received and NOT ACK will be returned
		Read data byte	0	0	1	1	Data byte will be received and ACK will be returned
0x58	Data byte has been received; NOT ACK has been returned	Read data byte or	1	0	1	X	Repeated START will be transmitted
		Read data byte or	0	1	1	X	STOP condition will be transmitted and TWSTO Flag will be reset
		Read data byte	1	1	1	X	STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset

Figura 2.20. Codici di stato in modalità Master-Receiver

Status Code (TWSR) Prescaler Bits are 0	Status of the 2-wire Serial Bus and 2-wire Serial Interface Hardware	Application Software Response					Next Action Taken by TWI Hardware
		To/from TWDR	To TWCR				
			STA	STO	TWINT	TWEA	
0x60	Own SLA+W has been received; ACK has been returned	No TWDR action or	X	0	1	0	Data byte will be received and NOT ACK will be returned
		No TWDR action	X	0	1	1	Data byte will be received and ACK will be returned
0x68	Arbitration lost in SLA+R/W as Master; own SLA+W has been received; ACK has been returned	No TWDR action or	X	0	1	0	Data byte will be received and NOT ACK will be returned
		No TWDR action	X	0	1	1	Data byte will be received and ACK will be returned
0x70	General call address has been received; ACK has been returned	No TWDR action or	X	0	1	0	Data byte will be received and NOT ACK will be returned
		No TWDR action	X	0	1	1	Data byte will be received and ACK will be returned
0x78	Arbitration lost in SLA+R/W as Master; General call address has been received; ACK has been returned	No TWDR action or	X	0	1	0	Data byte will be received and NOT ACK will be returned
		No TWDR action	X	0	1	1	Data byte will be received and ACK will be returned
0x80	Previously addressed with own SLA+W; data has been received; ACK has been returned	Read data byte or	X	0	1	0	Data byte will be received and NOT ACK will be returned
		Read data byte	X	0	1	1	Data byte will be received and ACK will be returned
0x88	Previously addressed with own SLA+W; data has been received; NOT ACK has been returned	Read data byte or	0	0	1	0	Switched to the not addressed Slave mode; no recognition of own SLA or GCA
		Read data byte or	0	0	1	1	Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1"
		Read data byte or	1	0	1	0	Switched to the not addressed Slave mode; no recognition of own SLA or GCA; a START condition will be transmitted when the bus becomes free
		Read data byte	1	0	1	1	Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1"; a START condition will be transmitted when the bus becomes free
0x90	Previously addressed with general call; data has been received; ACK has been returned	Read data byte or	X	0	1	0	Data byte will be received and NOT ACK will be returned
		Read data byte	X	0	1	1	Data byte will be received and ACK will be returned
0x98	Previously addressed with general call; data has been received; NOT ACK has been returned	Read data byte or	0	0	1	0	Switched to the not addressed Slave mode; no recognition of own SLA or GCA
		Read data byte or	0	0	1	1	Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1"
		Read data byte or	1	0	1	0	Switched to the not addressed Slave mode; no recognition of own SLA or GCA; a START condition will be transmitted when the bus becomes free
		Read data byte	1	0	1	1	Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1"; a START condition will be transmitted when the bus becomes free
0xA0	A STOP condition or repeated START condition has been received while still addressed as Slave	No action	0	0	1	0	Switched to the not addressed Slave mode; no recognition of own SLA or GCA
			0	0	1	1	Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1"
			1	0	1	0	Switched to the not addressed Slave mode; no recognition of own SLA or GCA; a START condition will be transmitted when the bus becomes free
			1	0	1	1	Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1"; a START condition will be transmitted when the bus becomes free

Figura 2.21. Codici di stato in modalità Slave-Receiver

Status Code (TWSR) Prescaler Bits are 0	Status of the 2-wire Serial Bus and 2-wire Serial Interface Hardware	Application Software Response					Next Action Taken by TWI Hardware
		To/from TWDR	To TWCR				
			STA	STO	TWINT	TWEA	
0xA8	Own SLA+R has been received; ACK has been returned	Load data byte or	X	0	1	0	Last data byte will be transmitted and NOT ACK should be received
		Load data byte	X	0	1	1	Data byte will be transmitted and ACK should be received
0xB0	Arbitration lost in SLA+R/W as Master; own SLA+R has been received; ACK has been returned	Load data byte or	X	0	1	0	Last data byte will be transmitted and NOT ACK should be received
		Load data byte	X	0	1	1	Data byte will be transmitted and ACK should be received
0xB8	Data byte in TWDR has been transmitted; ACK has been received	Load data byte or	X	0	1	0	Last data byte will be transmitted and NOT ACK should be received
		Load data byte	X	0	1	1	Data byte will be transmitted and ACK should be received
0xC0	Data byte in TWDR has been transmitted; NOT ACK has been received	No TWDR action or	0	0	1	0	Switched to the not addressed Slave mode; no recognition of own SLA or GCA
		No TWDR action or	0	0	1	1	Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1"
		No TWDR action or	1	0	1	0	Switched to the not addressed Slave mode; no recognition of own SLA or GCA; a START condition will be transmitted when the bus becomes free
		No TWDR action	1	0	1	1	Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1"; a START condition will be transmitted when the bus becomes free
0xC8	Last data byte in TWDR has been transmitted (TWEA = "0"); ACK has been received	No TWDR action or	0	0	1	0	Switched to the not addressed Slave mode; no recognition of own SLA or GCA
		No TWDR action or	0	0	1	1	Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1"
		No TWDR action or	1	0	1	0	Switched to the not addressed Slave mode; no recognition of own SLA or GCA; a START condition will be transmitted when the bus becomes free
		No TWDR action	1	0	1	1	Switched to the not addressed Slave mode; own SLA will be recognized; GCA will be recognized if TWGCE = "1"; a START condition will be transmitted when the bus becomes free

Figura 2.22. Codici di stato in modalità Slave-Transmitter

2.3 USART

Sebbene la comunicazione attraverso USART sia stata impiegata nel progetto in esame per realizzare lo scambio di informazioni tra il dispositivo Master del sistema ed il PC client, per amor di brevità non si intende discutere in questa sede tale aspetto dell'applicazione in maniera approfondita.

È opportuno, ad ogni modo, fornire almeno un quadro d'insieme su cosa essa sia e quali sono le sue caratteristiche fondamentali.

2.3.1 Il dispositivo UART

Il termine UART (Universal Asynchronous Receiver and Transmitter) si riferisce ad uno specifico circuito sviluppato per consentire una comunicazione seriale asincrona tra due dispositivi. Dunque, lo UART non costituisce propriamente un protocollo di comunicazione come I^2C o SPI, ma è essenzialmente un modulo hardware in un microcontrollore o circuito integrato autonomo.

Lo UART consente di realizzare in modo agevole un flusso di informazioni tra un PC client ed un microcontrollore AVR. I dati vengono trasmessi tramite UART in maniera asincrona, ovvero in assenza di un segnale di clock che sincronizzi l'uscita dei bit dall'UART trasmismittente ed il campionamento dei bit da parte dell'UART ricevente.

2.3.2 Il dispositivo USART

Lo USART (Universal Synchronous and Asynchronous Receiver and Transmitter) costituisce un'estensione del circuito presentato, consentendo di sviluppare una comunicazione sincrona. I dispositivi ATmega640/1280/2560 presentano quattro

circuiti USART. Per lo sviluppo del sistema proposto, è stato utilizzato USART0 ed è stata implementata una gestione basata totalmente su interruzioni.

In ultimo, si vuole far presente che lo UART/USART consente di realizzare una comunicazione "punto-punto" tra un solo dispositivo Master ed un solo Slave. Da ciò, è possibile apprezzare le diversità dei sistemi di comunicazione e la preferibilità di uno o di un altro a seconda dei requisiti specifici di un sistema.

2.4 GTK

GTK è un toolkit di widget, ovvero una una raccolta di librerie contenente un insieme di elementi di controllo grafici (chiamati widget) utilizzati per costruire l' interfaccia utente grafica (GUI) dei programmi. L'interfaccia di programmazione GTK è orientata agli oggetti (Object Orientated) e basata sugli eventi (Event Driven). Le applicazioni GNOME utilizzano il toolkit GTK.

All'interno del progetto proposto, si è fatto uso di GTK per implementare una finestra interattiva che agevoli l'utente nel controllo del sistema multi-motore.

Ancora una volta, questo aspetto esula dall'argomento che ci si è proposto di esaminare, motivo per il quale si ritiene ragionevole fermarsi a questo livello di presentazione e non indugiare oltre nei dettagli dello sviluppo dell'interfaccia.

Si desidera unicamente far presente che lo sviluppo della suddetta interfaccia ha consentito di unire la logica di input (dal PC client verso il Master) con quella dell' output (dal Master verso il PC client) all'interno di un unico processo. Questo ha comportato la necessità di implementare un sistema multi thread, unitamente ai dovuti di meccanismi di gestione dei conflitti tra thread concorrenti.

Capitolo 3

Il sistema sviluppato

Nel precedente capitolo sono stati forniti gli strumenti necessari per comprendere il sistema che ci si è prefissati di realizzare. Nel presente, si intende finalmente mostrare e commentare l'implementazione del sistema.

Occorre dunque tornare alla Figura 1.1 e fornire alcune delucidazioni di carattere generale sul funzionamento desiderato per il sistema, prima di addentrarsi nell'analisi del software.

3.1 Descrizione del sistema

Come si è detto, il sistema di comunicazione che occorre sviluppare è finalizzato al controllo di molteplici motori in corrente continua.

La comunicazione tra gli elementi del sistema viene guidata interamente dal dispositivo che assume il ruolo di Master, configurato per inviare quattro comandi diversi ai dispositivi Slave in ascolto:

- Comando 'SAMPLE': è il comando attraverso il quale il dispositivo Master effettua una richiesta (in broadcast) della velocità corrente del motore rivolta a tutti gli Slave in ascolto;
- Comando 'GET': è il comando attraverso il quale il Master richiede (in unicast) ad uno Slave di inviare la velocità corrente precedentemente campionata;
- Comando 'SET': è il comando attraverso il quale il Master invia (in unicast) la velocità che desidera applicare successivamente ad uno Slave;
- Comando 'APPLY': è il comando attraverso il quale il Master (in broadcast) impone in maniera simultanea a tutti gli Slave in ascolto di applicare la velocità desiderata.

Ad ogni comando inviato di tipo GET seguirà l'invio della velocità da parte dello slave, motivo per cui il Master dopo il comando GET dovrà passare temporaneamente in modalità Master-Receiver. Inoltre, ad ogni comando SET inviato deve seguire l'invio della velocità che si desidera applicare.

È bene far presente che il dispositivo Master non ha nessun potere decisionale per quanto riguarda la velocità da applicare agli Slave. Esso acquisisce (quando inviate) le velocità in input dall'utente attraverso lo USART e presenta in output le velocità correnti ricevute dagli Slave sempre attraverso il canale USART.

3.2 Implementazione

Di seguito vengono presentate le parti di codice ritenute di maggiore interesse per l'applicazione, commentate opportunamente laddove necessario.

3.2.1 TWI_lib.h

All'interno del file header TWI_lib.h vengono innanzitutto dichiarate le macro relative alla frequenza del modulo, allo stato effettivo del sistema (prelevando dal terzo al settimo bit del registro TWSR) e alla lunghezza dei buffer. A seguire, i buffer per la trasmissione e ricezione dei dati, con le relative variabili per la loro corretta gestione. Viene inoltre dichiarata la struttura dati TWI_info_struct, che contiene le informazioni rilevanti per la gestione del modulo TWI.

```
1  //Frequenza e bit di stato TWI//
2  #define TWI_FREQ 100000
3  #define TWI_STATUS  (TWSR & 0xF8)
4
5  //Buffer globali per la trasmissione e ricezione//
6  #define TRANSMIT_BUFLEN 30
7  #define RECEIVE_BUFLEN 30
8  uint8_t Transmit_Buffer[TRANSMIT_BUFLEN];
9  volatile uint8_t Receive_Buffer[RECEIVE_BUFLEN];
10 volatile int TB_Index;
11 volatile int RB_Index;
12 int transmit_len;
13 int receive_len;
14
15 //Modalita operative TWI//
16 typedef enum {
17   Ready,
18   Initializing,
19   Repeated_Start,
20   Master_Transmitter,
21   Master_Receiver,
22   Slave_Transmitter,
23   Slave_Receiver
24 } TWI_mode;
25
26 //Struttura dati info TWI//
27 typedef struct TWI_info_struct{
28   volatile TWI_mode mode;
29   uint8_t error_code;
30   uint8_t repeated_start;
31 }TWI_info_struct;
32
33 TWI_info_struct TWI_info;
```

Listing 3.1. Strutture dati TWI

In seguito, vengono definite delle macro utili per la modifica del registro di controllo TWCR, come quelle per la generazione delle condizioni di START/STOP, l'invio del bit (N)ACK e così via. In ogni caso, l'impostazione del del registro TWCR è stata effettuata in accordo con le specifiche riportate nel datasheet.

```
1  //Macro di controllo//
2
3  #define TWI_Set_Address() (TWCR=((1<<TWEA)|(1<<TWEN)|(1<<TWIE))
4
```

```
5 #define TWI_Send_Start() (TWCR=(1<<TWINT)|(1<<TWSTA)|(1<<TWEN)|(1<<
      TWIE))
6
7 #define TWI_Send_Stop() (TWCR=(1<<TWINT)|(1<<TWSTO)|(1<<TWEN)|(1<<
      TWIE))
8
9 #define TWI_Send_Transmit() (TWCR=(1<<TWINT)|(1<<TWEN)|(1<<TWIE))
10
11 #define TWI_Send_ACK() (TWCR=(1<<TWINT)|(1<<TWEA)|(1<<TWEN)|(1<<TWIE)
      )
12
13 #define TWI_Send_NACK() (TWCR=(1<<TWINT)|(1<<TWEN)|(1<<TWIE))
```

Listing 3.2. Macro di controllo

Vengono inoltre definite le macro per i codici di stato, con riferimento alle tabelle in Figura 2.19, Figura 2.20, Figura 2.21, Figura 2.22. Sono stati definiti anche dei codici di stato generici, non riportati in precedenza per sinteticità.

```
1 //GENERIC STATUS CODES//
2 #define TWI_SUCCESS                 0xFF
3 #define TWI_NO_RELEVANT_INFO        0xF8
4 #define TWI_ILLEGAL_START_STOP      0x00
5
6 //MASTER TRASMITTER AND MASTER RECEIVER COMMON//
7 #define START_TRANSMITTED           0x08
8 #define REP_START_TRANSMITTED       0x10
9 #define ARBITRATION_LOST            0x38
10
11 //MASTER TRANSMITTER//
12 #define SLAW_TR_ACK_RV              0x18
13 #define SLAW_TR_NACK_RV             0X20
14 #define M_DATA_TR_ACK_RV            0x28
15 #define M_DATA_TR_NACK_RV           0x30
16
17 //MASTER RECEIVER//
18 #define SLAR_TR_ACK_RV              0x40
19 #define SLAR_TR_NACK_RV             0x48
20 #define DATA_RV_ACK_TR              0x50
21 #define DATA_RV_NACK_TR             0x58
22
23 //SLAVE RECEIVER//
24 #define SLAW_RV_ACK_TR              0x60
25 #define BRDW_RV_ACK_TR              0x70
26 #define DATA_SLA_RV_ACK_TR          0x80
27 #define DATA_SLA_RV_NACK_TR         0x88
28 #define DATA_BRD_RV_ACK_TR          0x90
29 #define DATA_BRD_RV_NACK_TR         0x98
30 #define START_STOP_FOR_SLAVE        0xA0
31 #define ARBITRATION_LOST_SR_ADDR    0x68
32 #define ARBITRATION_LOST_SR_BRD     0x78
33
34 //SLAVE TRANSMITTER//
35 #define SLAR_RV_ACK_TR              0xA8
36 #define S_DATA_TR_ACK_RV            0xB8
37 #define S_DATA_TR_NACK_RV           0xC0
38 #define ARBITRATION_LOST_ST         0xB0
39 #define S_LDATA_TR_ACK_RV           0xC8
```

Listing 3.3. Macro per lo stato

Il codice relativo all'intestazione delle funzioni mostrate in seguito non è stato riportato.

3.2.2 TWI.c

All'interno del file TWI.c vengono definite le funzioni fondamentali per effettuare una comunicazione in ciascuna delle quattro modalità operative, oltre al gestore delle interruzioni TWI.

Inizializzazione

Le seguenti funzioni hanno lo scopo di inizializzare correttamente il modulo TWI e le variabili del software applicativo. La funzione TWI_Init():

- Imposta i valori iniziali nella struct TWI_info;
- Imposta a '0' gli indici dei buffer per la trasmissione/ricezione;
- Setta il bit rate del protocollo;
- Abilita il modulo TWI.

La funzione Slave_Addr_init() aggiunge a TWI_Init() l'impostazione del registro TWAR per l'indirizzo, consentendo di abilitare o disabilitare la comunicazione in broadcast.

Si evince che la funzione TWI_Init() verrà invocata dal dispositivo Master per inizializzare il sistema, mentre la funzione Slave_Addr_init() verrà invocata con lo stesso scopo dai dispositivi Slave, che necessitano l'attribuzione di un indirizzo.

```
1  #include <avr/io.h>
2  #include <avr/interrupt.h>
3  #include <stdio.h>
4  #include "TWI_lib.h"
5  #include "util/delay.h"
6  #include "util/atomic.h"
7
8  void TWI_Init(){
9    TWI_info.mode = Ready;
10   TWI_info.error_code = TWI_SUCCESS;
11   TWI_info.repeated_start = 0;
12   TWSR = 0;                                    //no prescaling
13   TWBR = ((F_CPU / TWI_FREQ) - 16) / 2;        //setta il bit rate
14   TWCR = (1 << TWIE) | (1 << TWEN);            //abilita TWI
15   TB_Index=0;
16   RB_Index=0;
17 }
18
19 void Slave_Addr_init(uint8_t SL_addr, uint8_t brd){
20   TWI_Init();
21   if (brd){
22     TWAR= (SL_addr << 1) | 0x01;           //LSB=1  broadcast attivo
23   }
24   else {
25     TWAR= (SL_addr << 1) | 0x00;
26   }
27 }
```

Listing 3.4. Inizializzazione

Trasmissione e Ricezione

La funzione TWI_Transmit_Data() predispone l'invio dei dati attraverso il bus per un dispositivo in modalità Master Transmitter.

Nello specifico:

1. Attende in busy waiting finché TWI non entra in modalità 'Ready';
2. Inizializza la trasmissione, copiando i dati da inviare nel buffer di trasmissione, dichiarando la dimensione degli stessi e ripristinando a '0' l'indice di trasmissione;
3. Inizia la trasmissione senza condizione di Start se questa sta avendo luogo a seguito di una Repeated Start, al contrario invia la condizione di Start;
4. Attende la fine della trasmissione, verificando in busy waiting che il modulo TWI ritorni in modalità 'Ready'.

La funzione is_TWI_ready() verifica semplicemente che la modalità dicharata nella struttura TWI_info sia 'Ready', pertanto non è stata riportata.

```
uint8_t TWI_Transmit_Data(void *const TR_data, uint8_t data_len,
    uint8_t repeated_start)
{
  if (data_len <= TRANSMIT_BUFLEN){
    while (!is_TWI_ready()) {
      _delay_us(1);
    }
    TWI_info.mode = Initializing;
    TWI_info.repeated_start = repeated_start;
    uint8_t *data = (uint8_t *)TR_data;
    for (int i = 0; i < data_len; i++){
      Transmit_Buffer[i] = data[i];
    }
    transmit_len = data_len;
    TB_Index = 0;
    if (TWI_info.mode == Repeated_Start){
      TWDR = Transmit_Buffer[TB_Index++];
      TWI_Send_Transmit();
    }
    else{
      TWI_Send_Start();
    }
    while (!is_TWI_ready()){
      _delay_us(1);
    }
  }
  else{
    return 1;
  }
  return 0;
}
```

Listing 3.5. Master Transmitter

Si desidera a questo punto far notare che lo scopo di questa e delle seguenti funzioni è quello di predisporre i dati per la trasmissione e generare il primo cambio di stato nel modulo TWI (utilizzando le macro di controllo definite in TWI_lib.h). Questa iniziale azione di controllo funge, infatti, da "trigger" per le successive, producendo

un "effetto cascata" che fa avanzare la trasmissione fino al suo termine.

Le altre funzioni del protocollo per la trasmissione/ricezione seguono la stessa logica della precedente, motivo per il quale vengono mostrate senza indugiare in ulteriori commenti, se non quelli presenti nel codice.

```
uint8_t TWI_Read_Data(uint8_t SL_addr, uint8_t bytes_to_read, uint8_t
    repeated_start)
{
  if (bytes_to_read < RECEIVE_BUFLEN){
    while (!is_TWI_ready()) {  //attesa modulo Ready
      _delay_us(1);
    }
    RB_Index = 0;
    receive_len = bytes_to_read; //dimensione lettura
    uint8_t TR_data[1];
    TR_data[0] = (SL_addr << 1) | 0x01;  //SLA+ Read
    TWI_Transmit_Data(TR_data, 1, repeated_start); //Trasmiss. SLA+R
  }
  else return 1;
  return 0;
}
```

Listing 3.6. Master Receiver

```
uint8_t TWI_Slave_Transmit_Data(void *const TR_data, uint8_t data_len
    ){
  if (data_len <= TRANSMIT_BUFLEN){
    while (!is_TWI_ready()) {           //attesa modulo Ready
      _delay_us(1);
    }
    TWI_info.mode = Initializing;       //inizializzazione
    TB_Index=0;
    transmit_len=data_len;
    TWI_Set_Address();                  //ascolto sul bus
    uint8_t *data = (uint8_t *)TR_data;
    for (int i =0; i< data_len; i++){
      Transmit_Buffer[i]= data[i];    //predisposizione dati
    }
    while (!is_TWI_ready()) {           //attesa fine trasmissione
      _delay_us(1);
    }
  }
  else return 1;
  return 0;
}
```

Listing 3.7. Slave Transmitter

```
uint8_t TWI_Slave_Receive_Data(){
  while (!is_TWI_ready()) {          //attesa modulo Ready
    _delay_us(1);
  }
  TWI_info.mode=Initializing;        //inizializzazione
  RB_Index=0;
  TWI_Set_Address();                 //ascolto sul bus
  while (!is_TWI_ready()) {
    _delay_us(1);                   //attesa fine ricezione
  }
  return 0;
}
```

Listing 3.8. Slave Receiver

Nota: ognuna delle funzioni presentate restituisce '0' in caso di avvenuto tentativo di trasmissione/ricezione, mentre un '1' viene restituito se si tenta di inviare/ricevere una quantità di dati maggiore della capacità del buffer.

Gestore delle interruzioni

Viene presentato, infine, il codice per la gestione delle interruzioni TWI sulla base dello stato corrente del sistema. È possibile a questo punto apprezzare l' "effetto valanga" di cui si è parlato in precedenza: in seguito alla gestione di un evento verificatosi nel sistema si agisce sul registro di controllo TWCR per generare un ulteriore evento (trasmissione di un dato, invio di un ACK...), facendo così in modo che la trasmissione prosegua passando di volta in volta attraverso l'ISR (Interrupt Service Routine), fino alla sua conclusione.

La gestione dei codici di stato segue quella riportata nelle tabelle in Figura 2.19, Figura 2.20, Figura 2.21, Figura 2.22.

```
ISR (TWI_vect){

  switch (TWI_STATUS){

    case REP_START_TRANSMITTED:
      TWI_info.mode=Repeated_Start;

    case START_TRANSMITTED:
      TWDR=Transmit_Buffer[TB_Index++];
      TWI_Send_Transmit();
      break;

    case SLAW_TR_ACK_RV:
      TWI_info.mode = Master_Transmitter;

    case M_DATA_TR_ACK_RV:

      if (TB_Index < transmit_len){
        TWDR=Transmit_Buffer[TB_Index++];
        TWI_Send_Transmit();
      }
      else if (TWI_info.repeated_start){
        TWI_Send_Start();
      }
      else{
        TWI_Send_Stop();
        TWI_info.error_code=TWI_SUCCESS;
        TWI_info.mode=Ready;
      }
      break;

    case SLAW_TR_NACK_RV:

    case M_DATA_TR_NACK_RV:

    case SLAR_TR_NACK_RV:

    case ARBITRATION_LOST:

      if (TWI_info.mode == Initializing){
        TB_Index--;
        TWI_Send_Start();
        break;
      }
```

```
45
46        TWI_info.error_code = TWI_STATUS;
47
48        if (TWI_info.repeated_start){
49          TWI_Send_Start();
50        }
51        else{
52          TWI_Send_Stop();
53          TWI_info.mode=Ready;
54        }
55        break;
56
57      case SLAR_TR_ACK_RV:
58        TWI_info.mode=Master_Receiver;
59        TWI_info.error_code = TWI_NO_RELEVANT_INFO;
60        if (RB_Index < receive_len -1){
61          TWI_Send_ACK();
62        }
63        else{
64          TWI_Send_NACK();
65        }
66        break;
67
68      case DATA_RV_ACK_TR:
69        Receive_Buffer[RB_Index++]=TWDR;
70        TWI_info.error_code = TWI_NO_RELEVANT_INFO;
71        if (RB_Index < receive_len -1){
72          TWI_Send_ACK();
73        }
74        else{
75          TWI_Send_NACK();
76        }
77        break;
78
79      case DATA_RV_NACK_TR:
80        Receive_Buffer[RB_Index++] = TWDR;
81        TWI_info.error_code = TWI_SUCCESS;
82        if (TWI_info.repeated_start){
83          TWI_Send_Start();
84        }
85        else{
86          TWI_Send_Stop();
87          TWI_info.mode = Ready;
88        }
89        break;
90
91      case ARBITRATION_LOST_SR_ADDR:
92
93      case ARBITRATION_LOST_SR_BRD:
94
95      case BRDW_RV_ACK_TR:
96
97      case SLAW_RV_ACK_TR:
98        TWI_info.mode=Slave_Receiver;
99        TWI_info.error_code = TWI_NO_RELEVANT_INFO;
100       if (RB_Index < RECEIVE_BUFLEN -1){
101         TWI_Send_ACK();
102       }
103       else{
104         TWI_Send_NACK();
105       }
106       break;
107
108     case DATA_BRD_RV_ACK_TR:
```

```
109
110     case DATA_SLA_RV_ACK_TR:
111       Receive_Buffer[RB_Index++] = TWDR;
112       TWI_info.error_code = TWI_NO_RELEVANT_INFO;
113       if (RB_Index < RECEIVE_BUFLEN-1){
114         TWI_Send_ACK();
115       }
116       else{
117         TWI_Send_NACK();
118       }
119       break;
120
121     case DATA_BRD_RV_NACK_TR:
122
123     case DATA_SLA_RV_NACK_TR:
124       Receive_Buffer[RB_Index++] = TWDR;
125       TWI_info.error_code = TWI_SUCCESS;
126       if (TWI_info.repeated_start){}
127       else{
128         TWI_info.mode = Ready;
129       }
130       break;
131
132     case S_DATA_TR_NACK_RV:
133
134     case S_LDATA_TR_ACK_RV:
135
136     case START_STOP_FOR_SLAVE:
137       TWI_info.error_code = TWI_SUCCESS;
138       TWI_Send_NACK();
139       if (TWI_info.repeated_start){}
140       else{
141         TWI_info.mode = Ready;
142       }
143       break;
144
145     case ARBITRATION_LOST_ST:
146
147     case SLAR_RV_ACK_TR:
148       TWI_info.mode=Slave_Transmitter;
149       TWI_Set_Address();
150
151     case S_DATA_TR_ACK_RV:
152       TWDR= Transmit_Buffer[TB_Index++];
153       TWI_info.error_code = TWI_NO_RELEVANT_INFO;
154       if (TB_Index < transmit_len){
155         TWI_Send_ACK();
156       }
157       else{
158         TWI_Send_NACK();
159       }
160       break;
161
162     case TWI_NO_RELEVANT_INFO:
163       break;
164
165     case TWI_ILLEGAL_START_STOP:
166       TWI_Send_Stop();
167       TWI_info.mode=Ready;
168       break;
169   }
170 }
```

Listing 3.9. ISR TWI

3.3 Note finali

- Tutte le funzioni implementate per la trasmissione/ricezione del modulo TWI sono bloccanti;
- Il valore di ritorno pari a '0' di una funzione per la trasmissione/ricezione non implica il successo della trasmissione, che è invece verificabile al termine della trasmissione attraverso il campo TWI_info.error_code, che deve risultare uguale a TWI_SUCCESS.

Capitolo 4

Esperimenti e casi d'uso

4.1 Esempio di una run di sistema

A questo punto non ci rimane che entrare nel vivo del progetto, utilizzando il modulo implementato per realizzare la comunicazione tra i dispositivi AVR finalizzata al controllo dei motori dc.

4.1.1 Dispositivo Master

A seguire viene mostrato esempio di gestione del sistema a due motori da parte del dispositivo Master.

Nel main loop vengono eseguiti i seguenti passaggi:

1. Si acquisiscono in input dal PC client tramite USART le velocità desiderate dall'utente dei sistema per i due motori, se queste sono state inserite (ricezione non bloccante);
2. Si inviano in successione i comandi SET a ciascuno dei due dispositivi Client, ciascuno immediatamente seguito dalla velocità desiderata;
3. Si invia il comando APPLY in broadcast per applicare simultaneamente le velocità desiderate ai motori;
4. Si invia il comando SAMPLE in broadcast per richiedere ai dispositivi di rilevare la velocità corrente del motore;
5. Si inviano in successione i comandi SET a ciascuno dei due dispositivo Client, ciascuno immediatamente seguito dalla funzione TWI_Read_Data() per ricevere la velocità corrente del motore;
6. Si manda in output le velocità ricevute.

```
while(1){

    UART_putString("Stato corrente motore 1: ");
    UART_putString(current_state_1);
    UART_putString("\n");

    UART_putString("Stato corrente motore 2: ");
    UART_putString(current_state_2);
    UART_putString("\n");
```

```
10
11    if(UART_getString(desired_velocity_1) > 1 && UART_getString(
      desired_velocity_2)>1){
12
13        ///// COMUNICAZIONE CON IL PRIMO SLAVE /////
14
15        set[0]= (0x01 << 1) | 0x00;
16
17        TWI_Transmit_Data(set, 5, 0);
18
19        send_velocity[0]= (0x01 << 1) | 0x00;
20
21        strcpy(send_velocity + 1, desired_velocity_1);
22
23        TWI_Transmit_Data(send_velocity, VELOCITY_LEN+1, 0);
24
25        ///// COMUNICAZIONE CON IL SECONDO SLAVE /////
26
27        set[0]= (0x02 << 1) | 0x00;
28
29        TWI_Transmit_Data(set, 5, 0);
30
31        send_velocity[0]= (0x02 << 1) | 0x00;
32
33        strcpy(send_velocity + 1, desired_velocity_2);
34
35        TWI_Transmit_Data(send_velocity, VELOCITY_LEN+1, 0);
36
37        ///// COMANDO APPLY IN BROADCAST /////
38
39        TWI_Transmit_Data(apply, 7, 0);
40
41    }
42
43    //// COMANDO SAMPLE IN BROADCAST /////
44
45    TWI_Transmit_Data(sample, 8, 0);
46
47    ///// CONTROLLO VELOCITA PRIMO MOTORE /////
48
49    get[0]= (0x01 << 1) | 0x00;
50
51    TWI_Transmit_Data(get, 5, 0);
52
53    TWI_Read_Data(0x01, VELOCITY_LEN, 0);
54
55    strcpy(current_state_1, Receive_Buffer);
56
57    ///// CONTROLLO VELOCITA SECONDO MOTORE /////
58
59    get[0]= (0x02 << 1) | 0x00;
60
61    TWI_Transmit_Data(get, 5, 0);
62
63    TWI_Read_Data(0x02, VELOCITY_LEN, 0);
64
65    strcpy(current_state_2, Receive_Buffer);
66
67    UART_putString("\n");
68
69    _delay_ms(1000);
70  }
```

Listing 4.1. Master.c main loop

Per brevità, è stato riportato solo il main loop presente nel file Master.c.

4.1.2 Dispositivi Slave

Viceversa, il comportamento di un generico Slave del sistema, all'interno del main loop, sarà il seguente:

1. Ricevi il comando da parte del Master;
2. Agisci in base al comando:
   - Se si è ricevuto un comando SAMPLE: acquisisci la velocità corrente del motore;
   - Se si è ricevuto un comando GET: trasmetti la velocità;
   - Se si è ricevuto un comando SET: memorizza la velocità desiderata per il motore;
   - Se si è ricevuto un comando APPLY: applica la velocità desiderata al motore.

```
while(1){
    TWI_Slave_Receive_Data();

    if (!strcmp(Receive_Buffer, SET_command)){
      TWI_Slave_Receive_Data();
      if (*Receive_Buffer != 0) strcpy(desired_velocity,
   Receive_Buffer);
    }
    else if(!strcmp(Receive_Buffer, APPLY_command)){
      int vel = atoi(desired_velocity);
      set_desired_velocity(mtr, vel);
    }
    else if(!strcmp(Receive_Buffer, SAMPLE_command)){
      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);
    }
    else{
      UART_putString("Non ho ricevuto un comando valido\n");
    }
  }
```

Listing 4.2. Slave.c main loop

Per brevità, è stato riportato solo il main loop presente nel file Slave.c.

Nota: quella che è stata presentata è una semplice applicazione del protocollo di comunicazione implementato. A titolo di esempio, il codice main del Master è stato gestito staticamente, conoscendo la presenza di due dispositivi Slave in ascolto sul bus.Ad ogni modo, non risulta difficile implementare il codice del Master in maniera dinamica, facendo in modo che possa adattarsi ad un numero variabile di Slave nel sistema.

Per finire, viene mostrato l'output ottenuto dall'esecuzione del sistema, a seguito di alcune richieste di modifica della velocità dei motori:

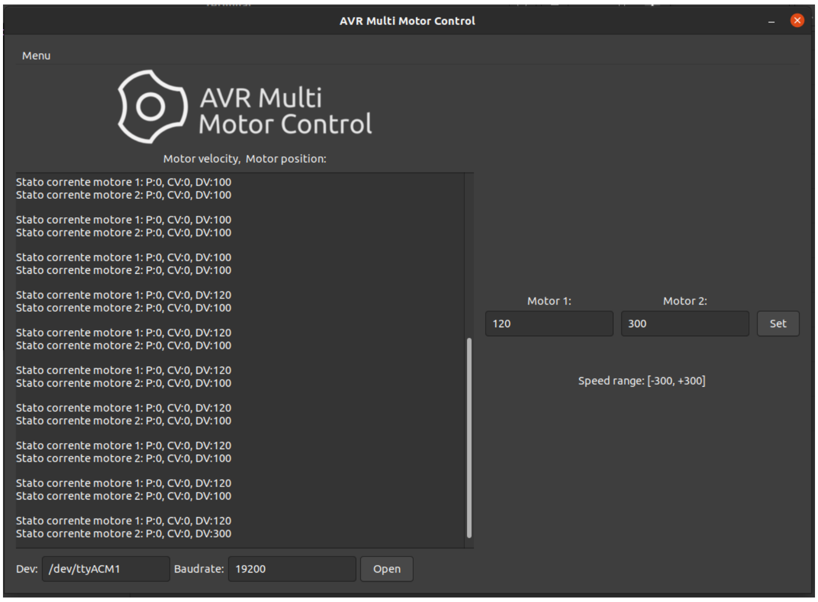

Figura 4.1. Output dell'applicazione

ed il collegamento fisico tra i dispositivi, secondo lo schema riportato in Figura 1.1.

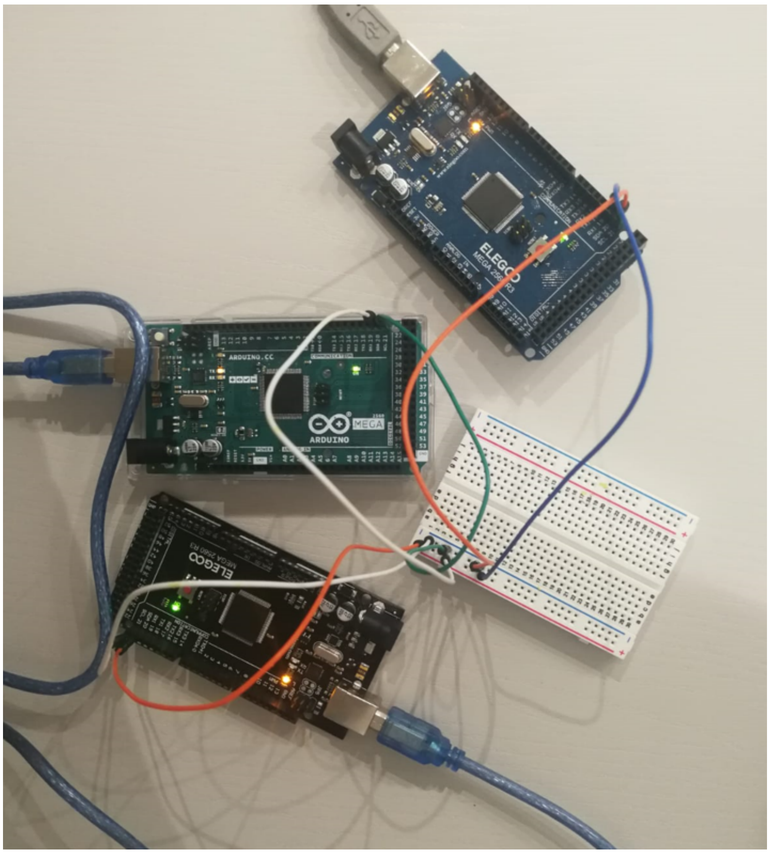

Figura 4.2. Collegamento tra i dispositivi

Capitolo 5

Conclusioni

Siamo dunque giunti alla fine della presente trattazione. L'autrice si augura di essere riuscita ad esporre in maniera chiara, se pur sintetica, i principali aspetti riguardanti il protocollo di comunicazione I^2C.

L'applicazione presentata costituisce la base per la realizzazione di progetti di complessità superiore, come può essere un piccolo dispositivo mobile telecomandato.

Il progetto attuale si configura come un primissimo approccio, da parte dell'autrice, al mondo dello sviluppo di applicazioni su microcontrollori. Un esercizio di fondamentale importanza, che stimola l' incremento di competenze tecniche ed analitiche.

Si ricorda nuovamente che è possibile prendere visione del codice completo dell'applicazione presso il seguente repository git: https://github.com/GeoDimi99/AVR-Multi-Motor-Control.

Bibliografia

[1] Communication protocol, https://en.wikipedia.org/wiki/Communication_protocol

[2] UART, I2C and SPI, https://www.seeedstudio.com/blog/2019/09/25/uart-vs-i2c-vs-spi-communication-protocols-and-uses/

[3] I2C infos, https://www.i2c-bus.org/

[4] I2C-bus specification and user manual, https://www.nxp.com/docs/en/user-guide/UM10204.pdf

[5] System Management Bus, https://it.wikipedia.org/wiki/System_Management_Bus

[6] Intelligent Platform Management Interface, https://en.wikipedia.org/wiki/Intelligent_Platform_Management_Interface

[7] Display Data Channel, https://en.wikipedia.org/wiki/Display_Data_Channel

[8] Atmel ATmega640/V-1280/V-1281/V-2560/V-2561/V, https://ww1.microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf

[9] UART, https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

[10] GTK, https://www.gtk.org/