Protocollo di comunicazione I^2C

Autore: Samuele Crivellaro – Versione 20/03/2017

Quest'opera è soggetta alla licenza Creative Commons Attribuzione - Non commerciale - Non opere derivate reperibile qui.

Sommario

1. Cos'è un protocollo di comunicazione?

Prima di iniziare a descrivere nel dettaglio il protocollo di comunicazione I^2C è utile precisare cosa sia, in generale, un protocollo di comunicazione. Detto in parole semplici, un protocollo di comunicazione consiste in una serie di regole che permette la comunicazione tra dispositivi. Per meglio capire questo punto possiamo pensare, più semplicemente, alle regole che governano una qualsiasi conversazione tra esseri umani: ogni qual volta iniziamo a parlare con una persona ci atteniamo implicitamente ad una serie di norme. Sicuramente dobbiamo parlare utilizzando la stessa lingua (a meno di non voler utilizzare un linguaggio gestuale, come si fa quando si va all'estero e non si conosce nemmeno l'inglese…).

Normalmente, poi, ci si attiene alla regola di buon senso di non parlare contemporaneamente; un'altra buona norma è quella di chiedere all'interlocutore di ripetere l'ultima frase qualora, per esempio, un qualche rumore avesse impedito la corretta comprensione. Ovviamente, in una conversazione tra esseri umani, tutte queste regole non sono rigidamente codificate, e quindi potremmo dire che tra uomini si usa un protocollo di comunicazione "approssimativo". Risulta però chiaro che un protocollo di questo tipo non potrebbe funzionare nella comunicazione tra dispositivi elettronici (per lo meno finché l'intelligenza artificiale non raggiungerà quella umana…).

Proviamo a costruire, a titolo di esempio, un semplice protocollo di comunicazione che permetta una comunicazione unidirezionale tra due dispositivi[1]. Immaginiamo che un sensore debba comunicare la luminosità da esso rilevata ad un microcontrollore; per fare questo deve inviare un numero compreso tra 0 e 255. Per esempio la sequenza inviata potrebbe essere la seguente:

```
32, 33, 33, 34, 36, 31, 28, 15, 33, 34, 35
```

Immaginiamo ora che il sensore sia in grado di rilevare, oltre alla luminosità, anche la temperatura ambientale; possiamo a questo punto proporre che vengano inviati i due valori in sequenza:

```
32, 88, 33, 87, 34, 88, 36, 89, 32, 91, 29
```

Già qui iniziamo a intravedere un problema: come possiamo essere sicuri che il microcontrollore, che riceve i dati, riesca ad interpretarli correttamente? Non si corre il rischio che il primo numero venga interpretato come temperatura, il secondo come luminosità e così via? Perché il tutto funzioni nella maniera corretta si potrebbe allora introdurre un qualche "indicatore di inizio sequenza". In effetti questo "indicatore" quasi sempre esiste nei protocolli di comunicazione e prende il nome di **sequenza di start**. Nel nostro caso, potremmo per esempio concordare la regola che ogni sequenza inizi con un numero ben preciso, per esempio 255 (ovviamente dovremmo limitare i valori rappresentativi delle grandezze misurate tra 0 e 254). La sequenza precedente diverrebbe allora:

```
255, 32, 88, 33, 87, 34, 88, 36, 89, 32, 91, 29
```

In questo modo siamo sicuri che, dopo lo "start" (il numero 255), i numeri di posizione dispari rappresentano valori di luminosità, e quelli di posizione pari valori di temperatura.

Aggiungiamo ancora un grado di complessità al nostro protocollo: cosa accadrebbe se il nostro sensore volesse inviare talvolta due soli valori (luminosità e temperatura) ed altre volte anche un terzo valore (per esempio l'umidità)? In tal caso la strategia illustrata poco fa non potrebbe per ovvie ragioni funzionare. Possiamo allora suggerire una ulteriore modifica del nostro protocollo che permetta di superare questo problema: far seguire ogni "start" da un numero indicante il numero di informazioni inviate. Se il sensore deve inviare le sole informazioni di luminosità e temperatura, farà seguire un '2' allo "start", se vuole invece inviare anche il valore di umidità invierà un '3' dopo lo "start". La sequenza sarebbe allora qualcosa di questo tipo:

```
255, 2, 32, 88, 255, 3, 33, 89, 122, 255, 3, 34, 91, 123
```

È chiaro che le regole iniziano a diventare notevolmente più complesse non appena la comunicazione richieda qualcosa di più complicato. Il principio, ad ogni modo, è quello che abbiamo appena cercato di descrivere.

Notiamo, prima di passare oltre, che l'ultima sequenza può idealmente essere suddivisa in tre pacchetti (in inglese *frame* oppure *packet*):

```
255, 2, 32, 88
255, 3, 33, 89, 122
255, 3, 34, 91, 123
```

ciascuno dei quali presenta, dopo lo "start", dei dati utili (in inglese *payload*, carico utile) e delle informazioni che potremmo definire "di contorno" (nel nostro esempio il numero di dati inviati dal sensore), in inglese talvolta indicati come *metadata*. Precisiamo comunque che i dati che abbiamo qui

[1] Cfr. Faludi R., *Building Wireless Sensor Networks*, 2011, O'Reilly, pp. 113 ss.

definito “di contorno” sono tutt’altro che accessori; senza di essi, infatti, la comunicazione non potrebbe avvenire.

Concludiamo il paragrafo indicando alcuni esempi di protocolli a tutti noti (anche a quelli che non sanno cos’è un protocollo):

- USB (Universal Serial Bus): regola le comunicazioni tra un PC e le sue periferiche;
- http (HyperText Transfer Protocol): usato per la trasmissione di informazioni sul Web;
- TCP/IP: insieme di protocolli per le reti di computer (tra cui http ed Ethernet)
- Ethernet: protocollo di comunicazione all’interno di reti locali. Esso, oltre a definire uno standard dal punto di vista hardware (tipo di cavi, tipi di connettori, ecc.) definisce anche la struttura di un pacchetto di dati (detto frame); in esso (cfr. Figura 1) sono riconoscibili, in particolare, il preambolo (una sequenza di start avente anche funzione di sincronizzazione), gli indirizzi MAC del destinatario e del mittente, il payload (carico utile, ovvero i dati effettivi della trasmissione).

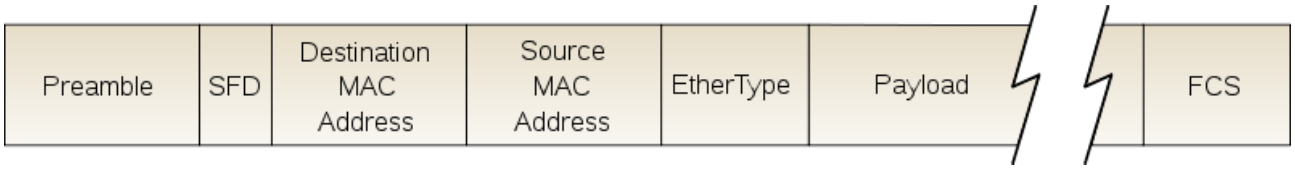


Figura 1. Struttura del pacchetto nel protocollo Ethernet

2. Caratteristiche generali del protocollo I2C

Il presente lavoro mostra come lavora il protocollo di comunicazione I^{2}C. Verrà discussa solamente la modalità a singolo master (la modalità in cui, cioè, è presente un singolo dispositivo che controlla il bus) poiché è la più comunemente utilizzata nei piccoli sistemi.

Il protocollo I^{2}C (pronuncia I-quadro-C, in Inglese I-squared-C) è stato creato dalla Philips Semiconductors nel 1982; la sigla, comunemente indicata anche con con I2C, sta per Inter-Integrated Circuit. Il protocollo permette la comunicazione di dati tra due o più dispositivi I^{2}C utilizzando un **bus**[2] a due fili, più uno per il riferimento comune di tensione (Figura 2). In tale protocollo le informazioni sono inviate serialmente usando una linea per i dati (SDA: Serial Data line) ed una per il Clock (SCL: Serial Clock line). Deve inoltre essere presente una terza linea: la massa, comune a tutti i dispositivi.

[2] In elettronica e informatica, il **bus** è un canale di comunicazione che permette a periferiche e componenti di un sistema elettronico di "dialogare" tra loro scambiandosi informazioni o dati di sistema attraverso la trasmissione di segnali. Diversamente dalle connessioni punto-punto un solo bus può collegare tra loro più dispositivi. [http://it.wikipedia.org/wiki/Bus_(informatica)]

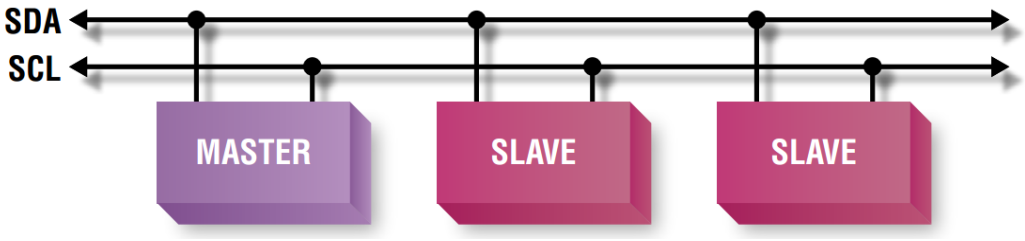


Figura 2. Bus I2C cui sono connessi un dispositivo master e due dispositivi slave

In generale ci sono 4 distinti modi di operare:

- un master trasmette – controlla il clock e invia dati agli slave
- un master riceve – controlla il clock e riceve dati dallo slave
- uno slave trasmette – il dispositivo non controlla il clock e invia dati al master
- uno slave riceve – il dispositivo non controlla il clock e riceve dati dal master.

Il dispositivo **master** è semplicemente il dispositivo che controlla il bus in un certo istante; tale dispositivo controlla il segnale di Clock e genera i segnali di START e di STOP. I dispositivi **slave** semplicemente "ascoltano" il bus ricevendo dati dal master o inviandone qualora questo ne faccia loro richiesta.

La modalità **multi-master** è un modo più complesso per l'uso del protocollo I^2C che permette di avere più dispositivi che controllano il bus in tempi differenti; in altre parole sono presenti più dispositivi che possono fungere da master. Tale modalità di funzionamento richiede la sincronizzazione dei segnali di Clock e coinvolge il concetto di **arbitraggio del bus**[3]. La modalità multi-master non sarà trattata nel presente lavoro, poiché l'uso più comune del protocollo I^2C consiste nell'usare un unico master che controlla dispositivi periferici come, per esempio, memorie, ADC (Analog to Digital Converter), DAC (Digital to Analog Converter), RTC (**Real Time Clock**[4]), controllo di display (per esempio nei telefoni cellulari).

3. Caratteristiche fisiche

L'interfaccia I^2C usa due linee bidirezionali; ciò significa che qualsiasi dispositivo potrebbe guidare entrambe le linee. In un sistema a master singolo il dispositivo master guida la linea di clock per la maggior parte del tempo; il master controlla il clock ma gli slave possono influenzarlo chiedendo di diminuirne la frequenza.

A differenza di altri bus, ai quali i dispositivi si connettono con uscite di tipo tri-state, i due conduttori del bus I^2C devono essere controllati come linee di uscita di tipo **open-collector/open-drain** e devono essere mantenute a livello alto usando un resistore di pull-up per ciascuna (Figura 3). Ogni dispositivo connesso alla linea che forzi il livello basso fa sì che tutti gli altri dispositivi vedano il livello basso; per il livello logico

[3] L'**arbitraggio** è la funzione che gestisce il possesso del bus per evitare ambiguità quando più master richiedono contemporaneamente il suo utilizzo; l'arbitro decide a quale dispositivo concederlo per primo [http://it.wikipedia.org/wiki/Arbitraggio_del_bus]

[4] Un ***real-time clock*** (**RTC**), orologio in tempo reale, è un dispositivo con funzione di orologio, solitamente costituito da un processore a circuito integrato specializzato per questa funzione, il quale conteggia il tempo reale (anno, mese, giorno, ore, minuti e secondi) anche quando l'utilizzatore viene spento. Viene usato in molti tipi di computer ed è presente in tutti i moderni PC. L'RTC è presente anche in molti sistemi embedded (sistemi elettronici appositamente progettati per una certa applicazione, per esempio elettrodomestici) nonché viene utilizzato anche in circuiti elettronici dove è necessario avere un preciso riferimento temporale. [http://it.wikipedia.org/wiki/Sistema_embedded]

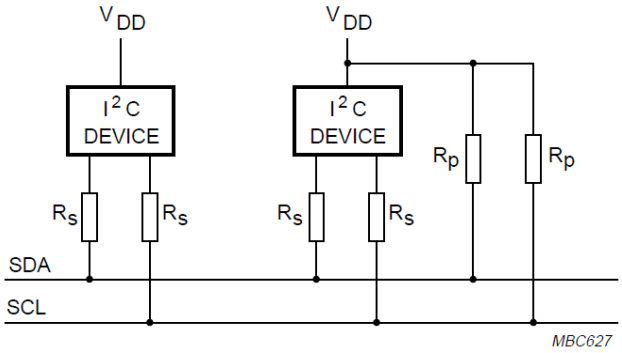


Figura 3. Connessioni elettriche ad un bus I^2C

alto tutti i dispositivi devono smettere di "tirare giù" la linea. Per uno sguardo più approfondito al funzionamento del bus I^2C si può fare riferimento all'appendice A.

I valori della **resistenza di pull-up** dipendono da vari fattori, quali la tensione di alimentazione, la capacità del bus[5] ed il numero di dispositivi connessi. In prima approssimazione possiamo dire che valori intorno ai 2 kΩ vanno bene per la maggior parte delle applicazioni; valori inferiori potrebbero dare origine a correnti troppo elevate per i dispositivi (specialmente quando la tensione di alimentazione è a 5 V), valori maggiori potrebbero dare problemi se la capacità del bus è elevata, anche se in generale il valore della resistenza di pull-up non è così critico.

È possibile usare anche dei resistori R_S (per esempio da 300 Ω) per la protezione contro gli spike di tensione[6].

La tensione V_{DD} è tipicamente – ma non necessariamente – di 3,3 oppure 5 V. Esiste anche la possibilità di usare tensioni di alimentazione diverse per i dispositivi connessi al bus. In questo caso se tutti i dispositivi sono "tolleranti" rispetto alla tensione di alimentazione disponibile possono essere collegati direttamente al bus; altrimenti si dovranno inserire dei circuiti adattatori di livello tra i nodi che hanno tensioni di alimentazione diverse. In appendice C si può trovare un approfondimento in tal senso.

Nella *modalità standard* la frequenza del clock può variare da 0 a 100 kHz. A partire dalla versione del 1992 del protocollo viene introdotta la modalità *fast-mode* che permette di raggiungere frequenze fino a 400 kHz. Nel 1998 viene introdotta la modalità *high speed mode* che permette di raggiungere la frequenza di 3,4 MHz.

Un dispositivo slave lento potrebbe avere la necessità di fermare il bus mentre sta raccogliendo dei dati o servendo un'interruzione. Per farlo deve forzare a livello basso la linea di clock SCL forzando così il master nello stato di attesa. Il master deve attendere finché la linea SCL è rilasciata dallo slave "lento".

4. Sequenza di trasferimento dati

Una sequenza elementare di lettura o scrittura di dati tra master e slave segue il seguente ordine (Figura 4):

1. Invio del bit di START (S) da parte del master
2. Invio dell'indirizzo dello slave (ADDR)7 ad opera del master

[5] Si veda l'appendice B per un approfondimento su tale concetto.

[6] Con **spike** ci si riferisce ad una rapida variazione di tensione, più precisamente ad un picco di tensione di breve durata (tipicamente da pochi millisecondi a 100 e oltre ms).

[7] Ciascun dispositivo slave connesso al bus deve essere univocamente individuato da un indirizzo. È possibile usare indirizzi a sette oppure a 10 bit. Usando l'indirizzamento a 7 bit (il più comune), esistono 2^7=128 indirizzi; di questi, però, 16 sono riservati, quindi solamente 112 indirizzi sono effettivamente disponibili.

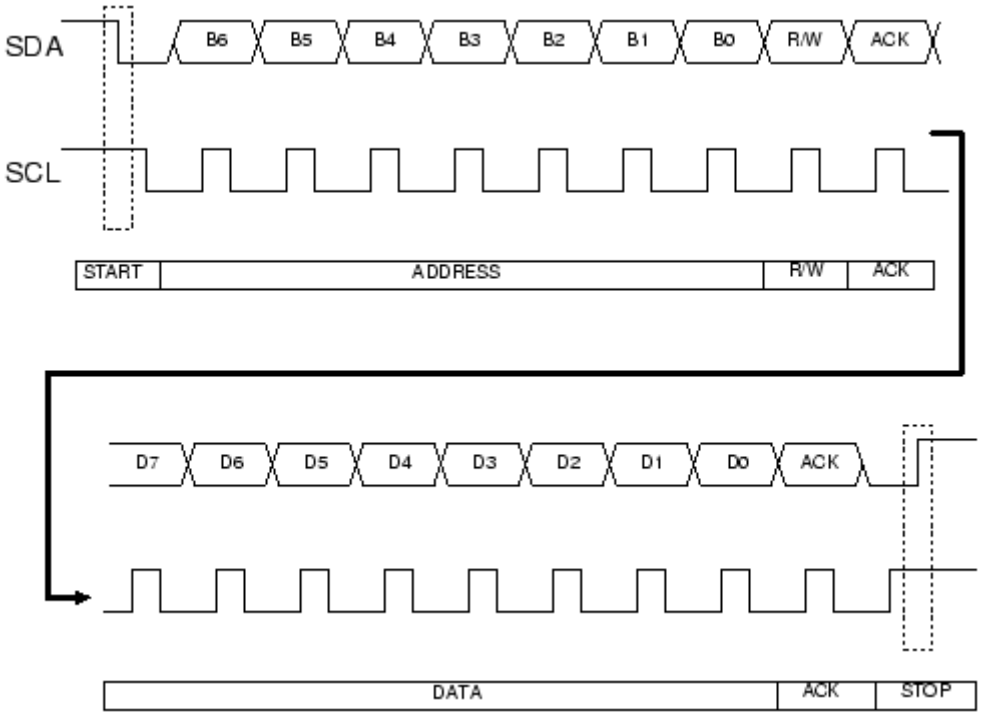


Figura 4. Tipica sequenza di trasferimento dati con il protocollo I^2C

3. Invio del bit di Read (R) o di Write (W), che valgono rispettivamente 1 e 0 (sempre ad opera del master)
4. Attesa/invio del bit di Acknowledge (ACK)
5. Invio/ricezione del byte dei dati (DATA)
6. Attesa/invio del bit di Acknowledge (ACK)
7. Invio del bit di STOP (P) da parte del master

I passi 5 e 6 possono essere ripetuti così da leggere o scrivere più byte.

Prima di passare ad analizzare più nel dettaglio le due tipiche sequenze per il trasferimento dal master allo slave e dallo slave al master, analizziamo nel dettaglio l'invio dei bit di start e di stop, l'invio e la ricezione dei bit dell'indirizzo o dei dati, l'invio e la ricezione dell'Acknowledge.

4.1 Bit di Start e bit di Stop

Una comunicazione avviene all'interno di due sequenze di segnali che il master invierà sul bus I^2C; queste sono il comando di START e di STOP. La regola fondamentale da tenere sempre ben presente è che nella trasmissione dei dati, il segnale SDA può cambiare stato soltanto mentre il segnale SCL è basso, ed il suo valore viene poi letto durante il fronte di salita o di discesa del segnale SCL. Se il segnale SDA cambia valore mentre SCL è alto, significa che il master non sta trasmettendo o ricevendo un dato, ma sta iniziando o terminando la comunicazione, attraverso appunto i comandi di start e di stop (Figura 5).

Segnale di start. Prima di inviare il segnale di start, lo stato del bus I^2C si trova nella condizione in cui entrambi i segnali SDA ed SCL sono al livello logico alto. Per inviare il comando di start, il master pone a livello basso il segnale SDA mentre SCL è a livello logico alto, in questo modo segnala agli altri dispositivi che sta per iniziare una comunicazione.

Segnale si stop. Prima di inviare il segnale di stop, sono stati trasferiti dei dati per cui lo stato del bus I^2C si trova nella condizione in cui entrambi i segnali SDA ed SCL sono al livello logico basso. Per inviare il

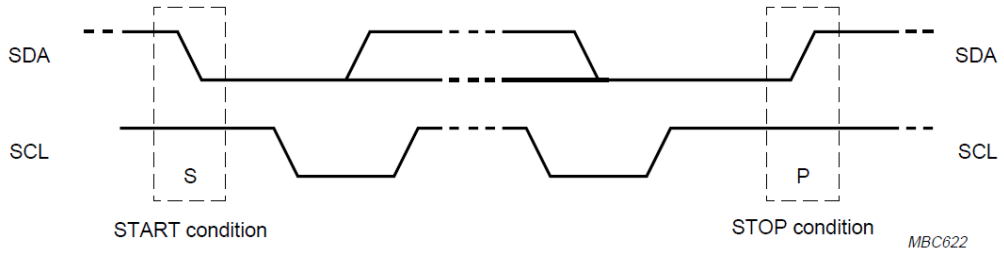


Figura 5. Sequenze di START e di STOP

comando di stop, il master pone prima a livello alto il segnale SCL e poi il segnale SDA. In questo modo segnala allo slave che la comunicazione è terminata.

4.2 Invio e ricezione di un bit

Siamo nella condizione in cui il master ha inviato la sequenza di START (oppure sono stati inviati altri bit) per cui entrambe le linee sono a livello logico basso (Figura 6).

Invio di un bit da parte del master. Poichè il segnale SDA può cambiare valore esclusivamente quando il segnale SCL è basso, il master in questo momento imposta il valore che deve trasmettere su SDA. Dopodichè manda alto il segnale SCL ed infine lo riporta basso.

Ricezione di un bit da uno slave. In questo caso è lo slave che impone lo stato del segnale SDA, quindi il master deve rilasciare questo bus e poi leggerne il valore. Quando il master pone a livello basso il segnale SCL lo slave stabilizza SDA al valore che vuole comunicare; il master, quindi, pone alto SCL, legge SDA e ripone a livello basso SCL.

4.3 Segnale di Acknowledge

In una comunicazione seriale è necessario comunicare la corretta ricezione dei dati; questo compito viene assolto tramite l'invio di un bit di riconoscimento (acknowledge) dal dispositivo che ha correttamente ricevuto il byte. Immaginiamo che il master abbia appena inviato un byte ad uno slave (Figura 7); per essere certo che il dato sia stato ricevuto correttamente aspetta dallo slave un bit di acknowledge: rilascia la linea SDA (che perciò si porterà a livello alto) ed inizia a leggerne lo stato; lo slave trasmette il bit ACK esattamente come quando trasmette un bit di dati, quindi durante un ciclo di SCL generato dal master. Se la trasmissione è avvenuta correttamente e lo slave vuole segnalare ACK, manterrà a livello basso SDA, in caso contrario lo slave manterrà a livello alto SDA. Se il master non riceve ACK, potrà terminare la comunicazione con un comando di STOP o iniziarne un'altra con una nuova sequenza di START.

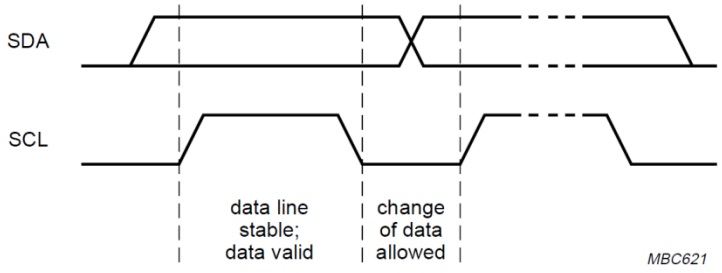


Figura 6. Invio e ricezione di un bit

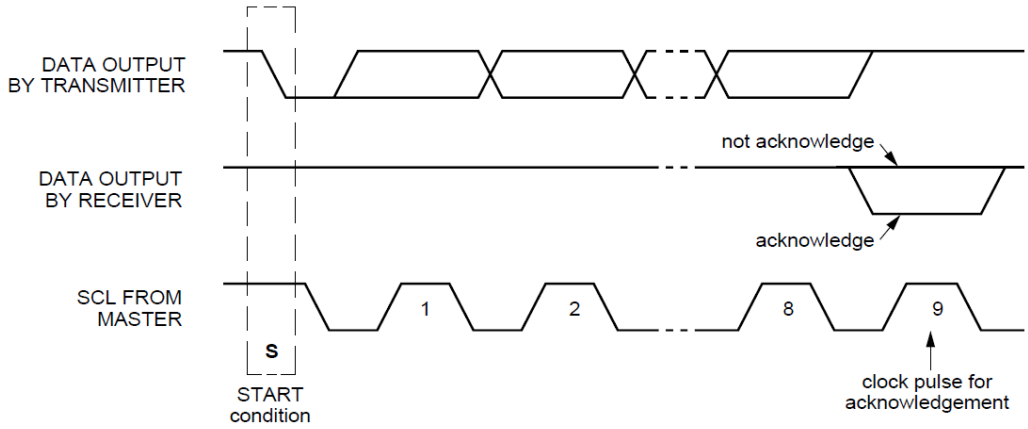


Figura 8. Segnale di Acknowledge o di not-Acknowledge

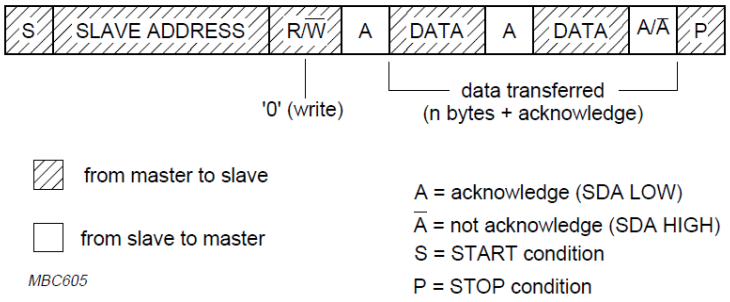


Figura 7. Sequenza di trasferimento dati dal master allo slave

4.4 Trasferimento di dati dal master allo slave

Il master invia la sequenza S, ADDR, W, quindi aspetta il bit di Acknowledge (A) dallo slave identificato dall'indirizzo inviato dal master. Se tale bit viene ricevuto correttamente dal master, questo invia il byte dei dati e aspetta un altro Acknowledge dallo slave. Dopo aver trasferito tutti i byte il master genera un bit di stop (Figura 8).

4.5 Trasferimento di dati dallo slave al master

Un processo analogo si verifica quando il master legge dei dati dallo slave, ma in tal caso, sarà inviato un R invece di un W. Dopo che i dati sono trasmessi dallo slave, sarà il master ad inviare il bit di Acknowledge; se invece il master non vuole altri dati, deve inviare un not-Acnowledge, che indica allo slave che deve liberare il bus. Questo permette al master di inviare il bit di stop (Figura 9).

I dispositivi nascono generalmente con un proprio indirizzo che solitamente può essere cambiato dall'utente attraverso una opportuna sequenza reperibile nei data-sheet del produttore.

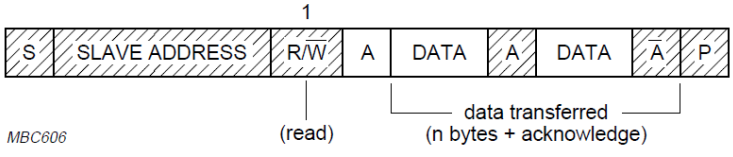


Figura 9. Sequenza di trasferimento dati dallo slave al master

4.6 Repeated START (Sr)

Per inviare più sequenze è possibile farle tutte iniziare con un bit di start e finire con uno di stop:

```
S ADDR (R/W) DATA A P,
```

oppure si può operare nel modo seguente:

```
S ADDR (R/W) DATA A Sr ADDR (R/W) DATA A P,
```

dove Sr indica un 'Repeated Start'; tale operazione può essere ripetuta indefinitamente.

5. Il protocollo I²C con Arduino

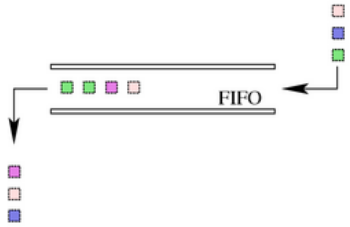
Presentiamo ora una modalità di implementazione del protocollo I²C utilizzando il sistema di sviluppo Arduino. Tra le librerie standar del sistema ne troviamo una appositamente dedicata al nostro protocollo: la libreria *Wire.h*. Informazioni dettagliate possono essere reperite sul sito www.arduino.cc; qui presentiamo solo le caratteristiche salienti ed una breve descrizione delle funzioni.

Su Arduino Uno troviamo le due linee del bus sui pin A4 (SDA) e A5 (SCL). La libreria *Wire.h* usa un indirizzamento a 7 bit; le sue funzioni sono le seguenti:

Funzione	**Uso**[8]	**Return**	**Parametri**	**Descrizione**
begin()	M S	void	()	Inizializza il dispositivo come master (da inserire nel setup())
			(address)	Inizializza il dispositivo come slave e gli assegna l'indirizzo 'address' (da inserire nel setup())
OPERAZIONI DI LETTURA DEI DATI IN ARRIVO				
requestFrom()	M	#byte restituiti dallo slave	(address, quantity)	Usato dal master per richiedere allo slave di indirizzo 'address' tanti byte quanti indicati da 'quantity'. I byte saranno poi recuperati con le funzioni available() e read(). La sequenza viene terminata con uno Stop.
			(address, quantity, stop)	Come il precedente. Se 'stop' è TRUE viene inviato un segnale di Stop al termine della sequenza, se è FALSE viene inviato un restart (Sr)
available()	M S	#byte disponibili	()	Restituisce il numero di byte disponibili (ovvero inviati da un altro dispositivo); tali byte dovranno essere recuperati con la funzione read()[9]. Tale funzione può essere usata: • da un master: in tal caso dovrà trovarsi dopo una

[8] Indica in quali tipologie di dispositivi può essere utilizzata: M = Master, S = Slave

[9] Per essere più precisi: i dati ricevuti vengono progressivamente inseriti in una **coda FIFO** (First In First Out, vedi figura), pronti per essere prelevati dalla funzione read().



Possiamo dire quindi che la funzione available() ci dice quanti byte sono presenti nella coda, mentre la funzione read() preleva un byte dalla coda (per essere precisi preleva il primo byte che era stato inserito).

Funzione	Uso[8]	Return	Parametri	Descrizione	
				chiamata a requestFrom() • da uno slave: dovrà essere chiamata all'interno della funzione indicata in onReceive()	
read()	M S	Il prossimo byte ricevuto	()	Legge un byte: • trasmesso da uno slave ad un master dopo una chiamata a requestFrom() • trasmesso da un master ad uno slave	
OPERAZIONI DI INVIO DI DATI AD ALTRI DISPOSITIVI					
beginTransmission()	M	void	(address)	Inizia una trasmissione I2C verso lo slave di indirizzo 'address'. Successivamente bisognerà mettere in una coda i byte da trasmettere tramite la funzione write(), ed infine inviare tali byte con la funzione endTransmission()	
write()	M S	#byte inviati	(value)	Invia 'value' come singolo byte	Tale funzione può essere usata sia da un master che da uno slave. Nel primo caso dovrà essere preceduta da un beginTransmission() e seguita da un endTransmission(), nel secondo caso no.
			(string)	Invia la stringa 'string' come serie di singoli byte	
			(data, length)	Invia un array di dati come singoli byte; la lunghezza dell'array è 'length'	
endTransmission()	M	Byte che indica lo stato della trasmissione[10]	()	Conclude la trasmissione ad uno slave iniziata con beginTransmission() e write() inviando i dati ed uno Stop	
			(stop)	Dopo aver inviato i dati, invia uno STOP se 'stop' è TRUE; se 'stop' è FALSE invia un segnale di Restart (Sr).	
GESTIONE DELLE OPERAZIONI DELLO SLAVE IN RISPOSTA ALLE INDICAZIONI DEL MASTER					
onRequest()	S	void	(handler)	Serve per indicare quale funzione (handler) deve essere chiamata nel momento in cui un master richiede dati a questo slave; la funzione che deve essere chiamata non deve avere parametri né deve restituire alcunché	
onReceive()	S	Void	(handler)	Serve per indicare quale funzione (handler) deve essere chiamata nel momento in cui questo slave riceve dati da un master; la funzione che deve essere chiamata non deve restituire alcunché e deve avere come unico parametro il numero di byte ricevuti dal master	

Per descrizioni più dettagliate si rimanda al Reference di www.arduino.cc. Presentiamo ora alcuni esempi di utilizzo del protocollo.

[10] Il significato di tale byte è il seguente (cfr. Reference di www.arduino.cc):
0:success; 1: data too long to fit in transmit buffer; 2: received NACK on transmit of address; 3: received NACK on transmit of data; 4: other error

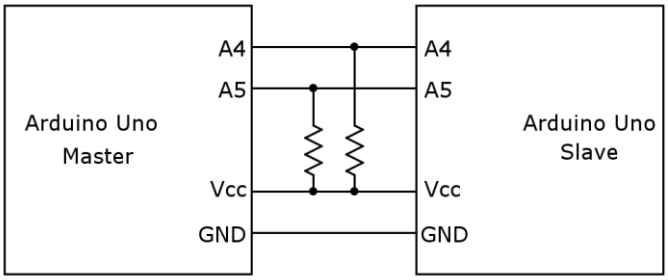


Figura 10. Connessione di due Arduino attraverso un bus I^2C

5.1 Il master chiede 2 byte ad uno slave

I due Arduino devono essere collegati come in Figura 10. Le due resistenze di pull-up possono essere scelte, per esempio, da 2,2 kΩ. Sarà sufficiente alimentare uno dei due Arduino (tramite USB o tramite un'alimentazione esterna), le connessioni indicate in figura provvederanno all'alimentazione anche dell'altro. Ad onor del vero, la libreria provvede ad attivare delle reistenze di pull-up interne che rendono quelle esterne a tutti gli effetti superflue. Bisogna però tenere in considerazione che le resistenze interne, avendo valori particolarmente elevati (oltre i 20 kΩ), possono impedire alte velocità di trasmissione (cfr. Appendice B); in tal caso si renderà necessaria una modifica della libreria oppure una disattivazione "manuale" di dette resistenze via software.

L'Arduino che funziona come master viene caricato con il seguente sketch (adattato, con leggere modifiche, dall'esempio standard 'master_reader' rintracciabile su www.arduino.cc):

```
#include <Wire.h>

int slaveAddress = 5;     // indirizzo dello slave

void setup()
{
  Wire.begin();           // attiva il bus I2C; il dispositivo è predisposto come master
  Serial.begin(9600);     // attiva la comunicazione seriale
}

void loop()
{
  Wire.requestFrom(slaveAddress, 2);    // richiesta di 2 byte al dispositivo di indirizzo
                                        // slaveAddress

  while(Wire.available())    // lo slave potrebbe inviare meno di quanto richiesto
  {
    char c = Wire.read();    // riceve un byte come carattere
    Serial.print(c);         // stampa il carattere sul monitor seriale
  }

  delay(100);
}
```

Mentre sullo slave sarà caricato il seguente (adattato dall'esempio standard 'slave_sender'):

```
#include <Wire.h>

int slaveAddress = 5;       // indirizzo dello slave

void setup()
{
```

```
  Wire.begin(slaveAddress);     // attiva il bus I2C; il dispositivo è predisposto come slave
  Wire.onRequest(requestEvent); // indica quale funzione deve essere chiamata nel momento in cui
                                // il master richiede dati a questo slave
}

void loop()     // non viene eseguita alcuna operazione se il master non invia richieste di byte
{
  delay(100);
}

// Questa funzione viene eseguita se il master richiede dei dati
void requestEvent()
{
  Wire.write("Hi");       // risponde con un messaggio di 2 byte, come richiesto dal master
}
```

In Figura 11 è riportato il diagramma temporale rilevato con un analizzatore di stati logici. Possiamo notare che il master non invia nessuna informazione relativa al numero di byte attesi; evidentemente questa informazione deve essere già in possesso allo slave. Nel nostro esempio l'informazione riguardante il numero di byte è scritta direttamente nel codice; più realisticamente essa dovrebbe essere trasferita dal master allo slave tramite un'apposita comunicazione.

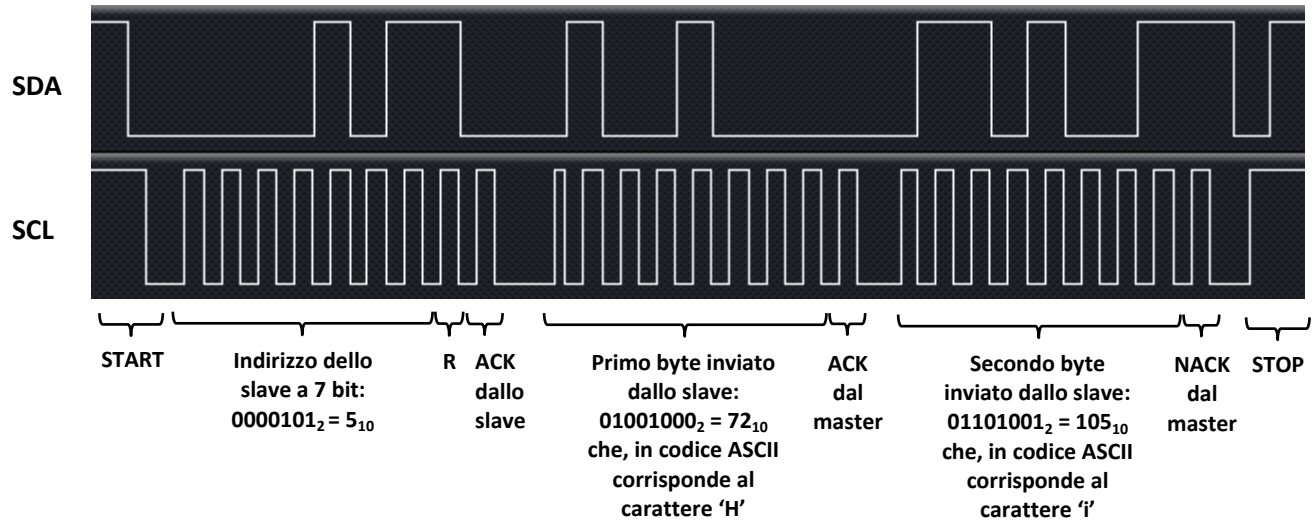


Figura 11. Il master chiede due byte allo slave di indirizzo 5; lo slave risponde inviando i caratteri 'H' ed 'i'

5.2 Il master chiede 2 byte ad uno slave, nessuno slave risponde

Immaginiamo ora di sconnettere, dal circuito di Figura 10, l'Arduino che funge da slave. È facile immaginare la seguente sequenza:

```
S ADDR R NACK P;
```

infatti nessuno slave risponderà con un Acknowledge (il che equivale ad un not-Acknowledge); al che il master interromperà la sequenza con uno Stop. La Figura 12, rilevata con un analizzatore di stati logici, conferma la nostra ipotesi.

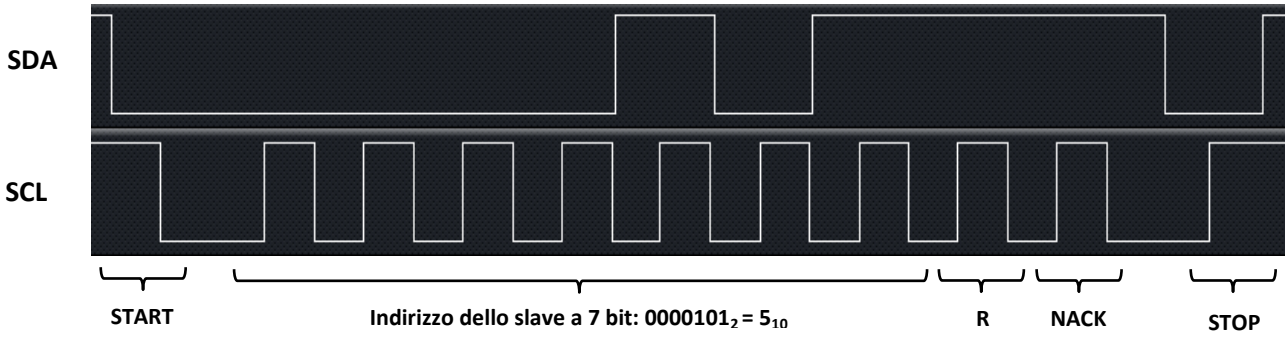


Figura 12. Il master chiede due byte allo slave di indirizzo 5; non essendo presente sul bus nessuno slave con tale indirizzo, nessun dispositivo risponde con un ACK.

5.3 Il master invia tre byte ad uno slave

Consideriamo nuovamente il circuito di Figura 10. L'Arduino che funziona come master viene caricato con il seguente sketch (adattato, con leggere modifiche, dall'esempio standard 'master_writer' rintracciabile in www.arduino.cc):

```
#include <Wire.h>

int slaveAddress = 4;    // indirizzo dello slave

void setup()
{
  Wire.begin();          // attiva il bus I2C; il dispositivo è predisposto come master
}


void loop()
{
  Wire.beginTransmission(slaveAddress);     // inizia la trasmissione verso lo slave di indirizzo
                                            // slaveAddress
  Wire.write("x=");                         // inserisce due byte nella coda dei dati da
                                            //trasmettere
  Wire.write(17);                           // inserisce un byte nella coda dei dati
                                            // trasmettere
  Wire.endTransmission();                   // invia I byte caricati nella coda e invia uno stop

  delay(500);
}
```

Sullo slave sarà invece caricato il seguente (adattato dall'esempio standard 'slave_receiver'):

```
#include <Wire.h>

int slaveAddress = 4;      // indirizzo dello slave

void setup()
{
  Wire.begin(slaveAddress);       // attiva il bus I2C; il dispositivo è predisposto come slave
  Wire.onReceive(receiveEvent);   // indica quale funzione deve essere chiamata nel momento in
                                  // cui il master richiede dati a questo slave
  Serial.begin(9600);             // attiva la comunicazione seriale
```

```
}

void loop()     // non viene eseguita alcuna operazione se il master non invia dati
{
  delay(100);
}

// Questa funzione viene eseguita se il master invia dei dati
void receiveEvent(int howMany)
{
  while(Wire.available() > 1) // tale ciclo stampa a monitor seriale tutti i byte tranne l'ultimo
  {
    char c = Wire.read();     // preleva un byte dalla coda dei dati in arrivo…
    Serial.print(c);          // … e lo stampa sul monitor seriale
  }
  int x = Wire.read();        // preleva dalla coda l'ultimo byte ricevuto…
  Serial.println(x);          // … e lo stampa sul monitor seriale
}
```

In Figura 13 è riportato il diagramma temporale della trasmissione, la cui interpretazione è lasciata come esercizio.

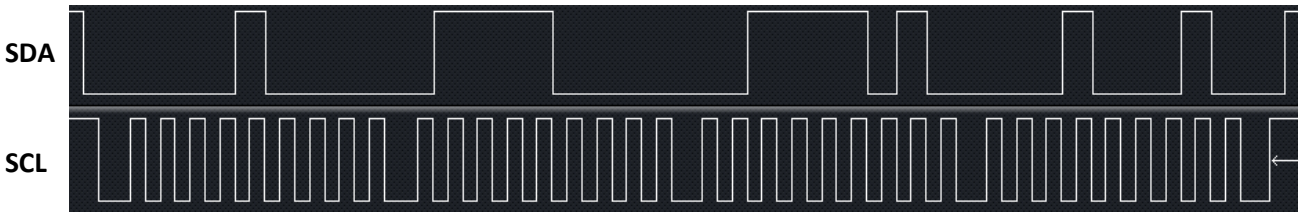


Figura 13. Il master invia tre byte allo slave di indirizzo 4

Esercizi

1. Interpretare le seguenti sequenze di trasmissione I2C, indicando il significato dei vari gruppi di bit e chi li ha inviati:
   a) S 001101010110011001 P
   b) S 010001100011000110 P
   c) S 100001011 P
   d) S 001010001 P
   e) S 0100001100111011110110011001 P
   f) S 000111100100100100011101010 P
2. Scrivere uno sketch per un Arduino che, connesso ad un bus I2C, si comporti come master inviando la seguente sequenza: S 0001010 R; qualora lo slave risponda con un ACK deve essere in grado di leggere 5 byte dallo slave.
3. Scrivere uno sketch per un Arduino che risponda come slave al master dell'esercizio precedente inviandogli 5 numeri casuali compresi tra 0 e 255.
4. Scrivere due sketch che permettano la comunicazione tra due Arduino tramite I2C; il sistema deve compiere le seguenti operazioni:
   - il master richiedere allo slave un numero
   - lo slave risponde con un byte contenente un numero casuale compreso tra 0 e 127
   - il master calcola il doppio del numero ricevuto e lo reinvia allo slave
   - lo slave visualizza il numero ricevuto sul monitor seriale del PC
5. Scrivere due sketch che permettano la comunicazione tra due Arduino tramite I2C; il sistema deve compiere le seguenti operazioni:
   - il master richiedere allo slave un numero
   - lo slave risponde con un byte generato casualmente
   - il master invia allo slave la stringa "pari" se il numero ricevuto è pari, altrimenti invia la stringa "dispari"
   - lo slave visualizza la stringa ricevuta sul monitor seriale del PC
6. Progettare un semplice sistema con bus I2C cui si connettano due Arduino, un master ed uno slave (avente indirizzo 20). Connettere al master un pulsante ed un LED, allo slave un interruttore. Il master, ogni qual volta viene premuto il pulsante, deve chiedere allo slave lo stato dell'interruttore e accendere o spegnere il LED di conseguenza.
7. Progettare un semplice sistema con bus I2C cui si connettano tre Arduino, un master e due slave (aventi indirizzi 72 e 73). Il master deve essere connesso al PC; i due slave devono essere in grado di leggere un valore di tensione prelevato sul cursore di un trimmer connesso tra VCC e massa. Quando il master riceve il carattere 'a' dal PC deve fare richiesta di un byte al dispositivo di indirizzo 72 e poi stamparne il valore sul monitor seriale; quando riceve il carattere 'b' deve eseguire la medesima operazione col dispositivo di indirizzo 73. Qualora, invece, ricevesse un qualsiasi altro carattere deve inviare al monitor seriale il messaggio "Carattere non valido". Scrivere gli sketch per tutti e tre i dispositivi; il byte inviato dagli slave sarà generato in base alla lettura del trimmer (255 corrisponderà a 5 V, 0 a 0 V).
8. Progettare un semplice sistema con bus I2C cui si connettano due arduino, un master ed uno slave. Lo slave contiene otto registri, simulati da un array ad otto elementi. Implementare due sketch in modo che il sistema sia in grado di compiere le seguenti operazioni:
   - il master invia allo slave l'indirizzo del registro a cui vuole accedere (si supponga che gli indirizzi dei registri vadano da 0 a 7);
   - lo slave risponde con il byte contenuto nel registro richiesto dal master.

Bibliografia e Sitografia

http://www.arduino.cc

http://www.youblisher.com/p/568614-Collage-sul-protocollo-I2c-Creato-da-Tumino-Enrico/

http://www.best-microcontroller-projects.com/i2c-tutorial.html

https://sites.google.com/site/sitodelguru/elettronica/il-protocollo-i2c

http://it.wikipedia.org/wiki/I%C2%B2C

Data-sheet del protocollo I^2C della Philips

Faludi R., *Building Wireless Sensor Networks*, 2011, O'Reilly

Per le immagini di pubblico dominio:

http://openclipart.org/

Appendice A – Struttura interna dei dispositivi connessi al bus I^2C

Per comprendere pienamente il funzionamento del bus I^2C e in generale di qualsiasi bus che si basa su uscite di tipo open-collector possiamo fare riferimento alla Figura 14.

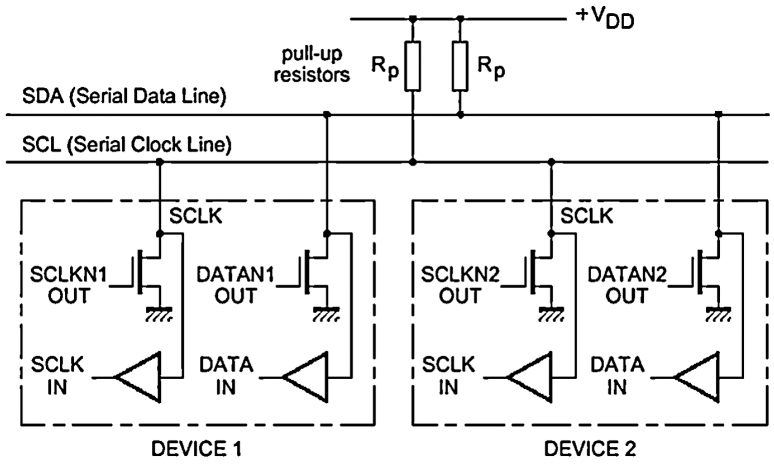


Figura 14. Struttura interna dei dispositivi connessi ad un bus I^2C

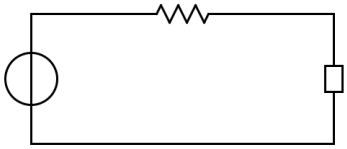
Appendice B – Osservazioni sulla capacità di un bus

Adattato da http://www.i2c-bus.org/termination-versus-capacitance/

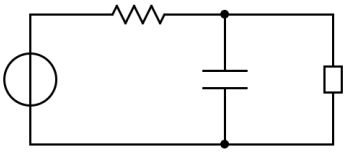
Un bus è composto da più linee; immaginiamo, per semplificare, che sia composto solamente da due linee, una di andata ed una di ritorno. Idealmente le immaginiamo come due corto-circuiti:

Sappiamo in realtà dall'elettrotecnica di base che esse presentano una loro resistenza; immaginando di inglobare la resistenza di entrambi i conduttori in un unico resitore, otteniamo il seguente modello:

Se poi vogliamo essere ancora più raffinati, dobbiamo tener presente il fatto che i due conduttori



possono essere visti come le armature di un condensatore (di capacità più meno grande a seconda della loro lunghezza, della distanza…); il modello precedente può allora essere modificato così:



Tralasciando gli effetti di tipo induttivo, ci rendiamo conto che un gradino di tensione all'ingresso diventa, sul carico, un gradino "smussato", in base ai ben noti fenomeni transitori.

Proviamo allora a vedere, con l'aiuto di alcuni grafici, come i valori della resistenza di pull-up R_p e della capacità del bus C_p (che sarà in realtà qualcosa di più complicato di quanto visto nella figura precedente) influiscano sul funzionamento del bus. In particolare il loro aumento limita la massima velocità del bus, ovvero la massima frequenza ammissibile per il clock.

Immaginiamo in primo luogo una resistenza di pull-up di 10 kΩ ed una capacità del bus di 300 pF. Com'è facile intuire, all'arrivo del gradino di ingresso la tensione sulle due linee cresce con legge esponenziale (Figura 15).

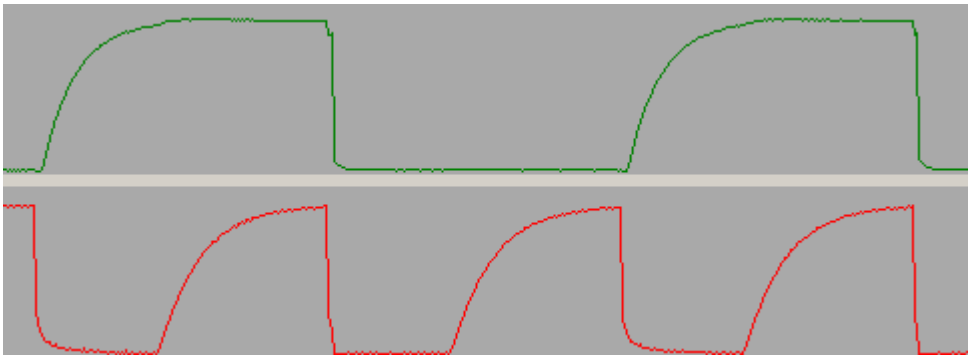


Figura 15. Andamento dei segnali SDA (sopra) ed SCL (sotto) ipotizzando una resistenza di pull-up R_p=10 kΩ, una capacità del bus C_p=300 pF ed una frequenza del segnale di clock di 100 kHz.

In Figura 16 vediamo cosa accade se diminuiamo R_p a 2 kΩ lasciano invariata la capacità. Possiamo notare che il transitorio è più breve, grazie al valore più basso della resistenza. È a questo punto abbastanza semplice capire che non è possibile aumentare troppo R_p senza rendere il segnale "illeggibile"; in tal caso, infatti, la tensione sulle linee non giungerebbe al suo valore nominale in tempo utile per essere correttamente letta dai dispositivi.

Notiamo infine che C_p cresce considerevolmente all'aumentare della lunghezza dei conduttori: le connessioni I2C dovrebbero sempre essere il più corte possibile.

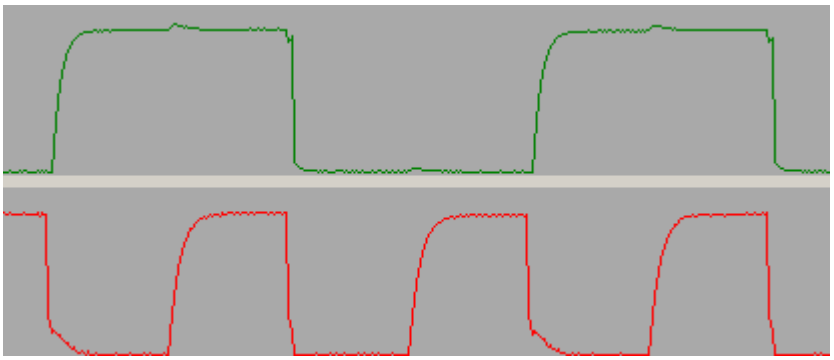


Figura 16. Andamento dei segnali SDA (sopra) ed SCL (sotto) ipotizzando una resistenza di pull-up R_p=2 kΩ, una capacità del bus C_p=300 pF ed una frequenza del segnale di clock di 100 kHz.

Appendice C – Circuiti adattatori di livello

Adattamento tratto dall'Application Note AN10441 della NXP

Dovendo connettere ad uno stesso bus dispositivi che funzionano a tensioni diverse, per esempio 3,3 e 5 V, è sufficiente utilizzare due MOSFET a canale n secondo la configurazione indicata in Figura 17, pur non essendo questa l'unica soluzione possbile.

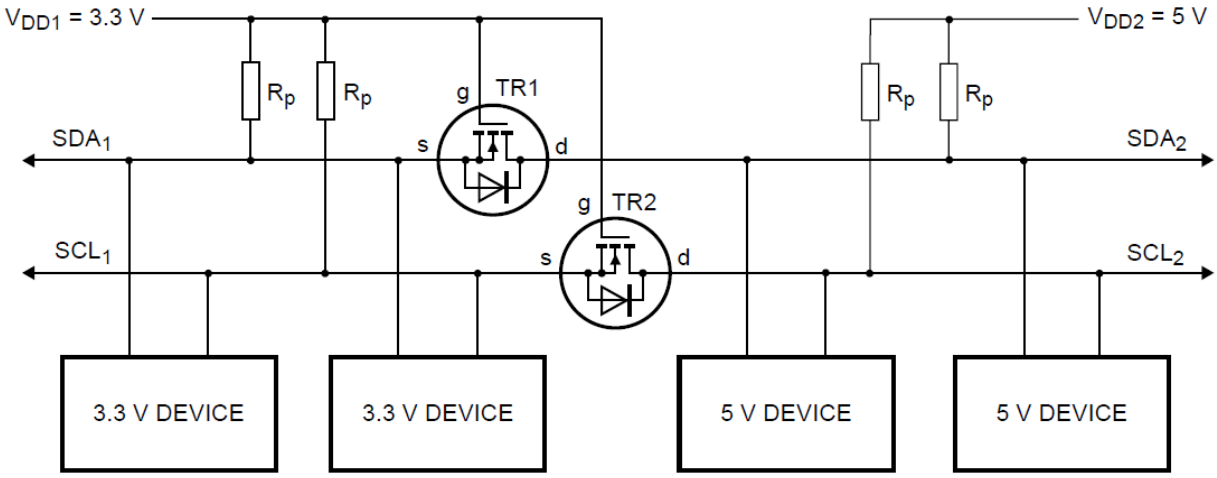


Figura 17. Adattatore di livello bidirezionale per la connessione di dispositivi a 3,3 V con dispositivi a 5 V in un sistema con bus I^2C

Analizziamo il funzionamento del primo transistor (TR1); il ragionamento potrà poi essere ripetuto per il secondo transistor. Dobbiamo considerare tre casi separati: 1) nessun dispositivo sta portando la linea SDA a livello basso; 2) un dispositivo a 3,3 V sta portando la linea a livello basso; 3) un dispositivo a 5 V sta portando la linea a livello basso. Analizzizmo separatamente i tre casi:

1. Se nessun dispositivo sta portando la linea a livello basso, la linea della sezione a "bassa tensione" viene mantenuta alta dal resistore di pull-up connesso ai 3,3 V. Poiché, quindi, gate e source di TR1 si trovano entrambi a 3,3 V, la loro d.d.p., essendo nulla, è inferiore della tensione di soglia VGS(th) e il transistor è off. Ciò permette alla linea della sezione ad "alta tensione" di essere mantenuta alta dal resistore di pull-up connesso ai 5 V. Le due sezioni del bus, quindi, si trovano entrambe a livello alto, ma ad un differente valore di tensione.
2. Se un dispositivo della sezione a "bassa tensione" porta la linea a zero volt, TR1 entra in conduzione; infatti VGS, che varrà 3,3 V, sarà maggiore di VGS(th). Tra drain e source si instaurerà, idealmente, un corto-circuito; nella realtà VDS sarà diverso da zero ma comunque sufficientemente basso da poter ritenere che anche la sezione ad "alta tensione" vada a zero volt.
3. Se un dispositivo della sezione ad "alta tensione" porta la linea a zero volt il diodo drain-substrato11 (in Figura 17 rappresentato tra source e drain) avrà il catodo a 0 V e l'anodo a 3,3 V ed entrerà quindi in conduzione, portando il source a circa 0,7 V. A questo punto VGS varrà circa 2,6 V (infatti VG=3,3 V e VS=0,7 V) portando il MOS in conduzione; ci ritroveremo perciò di nuovo nella situazione descritta al punto 2. Anche in questo caso, quindi, entrambe le sezioni della linea SDA si troveranno a livello basso.

[11] Il diodo drain-substrato, detto anche diodo drain-body o più semplicemente diodo di body, non è un diodo "aggiunto" al transistor MOS, ma è intrinseco alla struttura fisica del MOS stesso. In altre parole è sempre presente e non sarebbe possibile toglierlo senza compromettere la struttura del MOS e, con ciò, la sua funzionalità.

I tre stati descritti mostrano che i livelli logici sono trasferiti in entrambi i versi. Il primo stato realizza la funzione di traslazione del livello di tensione; il secondo ed il terzo stato realizzano una wired-AND[12] tra le due sezioni del bus.

Possono essere applicate tensioni di alimentazione diverse da 3,3 V per V_{DD1} e da 5 V per V_{DD2}; per esempio 2 V per V_{DD1} e 10 V per V_{DD2}; è comunque necessario che V_{DD1} sia minore o uguale a V_{DD2}. In caso contrario, infatti, il diodo drain-substrato entrerebbe in conduzione anche quando nessun dispositivo sta portando la linea a livello basso, e non sarebbe perciò più garantito la funzionalità del circuito.

Esistono in commercio dei moduli già pronti che realizzano le funzioni sopra descritte; in Figura 18 ne è rappresentato un esempio.

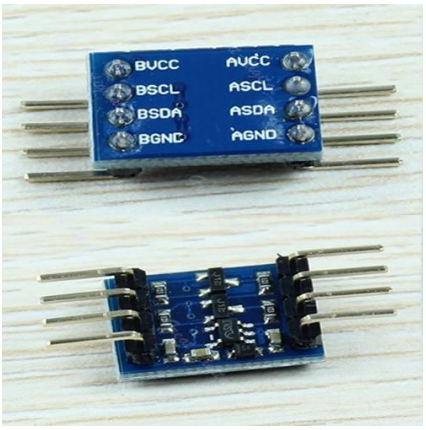


Figura 18. Modulo per la conversione 5-3,3 V per bus I^2C

[12] Si parla di wired-AND quando viene realizzata la funzione logica AND senza une vera e propria porta logica. In questo caso, in particolare, possiamo notare che la linea si trova a livello basso se almeno un dispositivo la porta a livello basso, in tutti gli altri casi essa rimarrà a livello alto. Tradotto nei termini più familiari delle tavole di verità: se anche un solo dispositivo è a zero, l'uscita va a zero, altrimenti resta a uno; immaginando tre dispositivi connessi al bus la tabella diventa:

1° dispositivo	2° dispositivo	3° dispositivo	Livello logico del bus
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

È possbile notare che, non appena un dispositivo porta la propria uscita a zero, il bus si porta a zero; nella tabella è facile riconoscere la funzione AND.