ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA
in
Calcolatori Elettronici T

Sviluppo di progetti per fini didattici con Arduino

CANDIDATO
Giovanni Bonaccio

RELATORE:
Prof. Stefano Mattoccia

CORRELATORI
Dott. Matteo Poggi
Dott. Fabio Tosi

Anno Accademico 2017/18

Sessione II

INDICE

CAPITOLO 1 Introduzione 5

CAPITOLO 2 AVR_GCC e AVR_DUDE.....7

2.1 Installazione 7

2.1.1 Installazione AVR-GCC.....7

2.1.2 Installazione AVR-DUDE.....8

2.2 Compilazione e caricamento del codice sorgente 8

2.2.1 Da un singolo file sorgente all'Arduino 8

2.2.2 Compilazione e caricamento di più file sorgente su Arduino 8

2.2.3 Breve spiegazione dei comandi per la compilazione e caricamento 9

CAPITOLO 3 Pin mapping e interazione con i registri 10

CAPITOLO 4 Libreria common_commands.h: semplici operazioni di I/O 13

4.1 Librerie esterne.....13

4.2 Definizioni.....13

4.2.1 Macro 13

4.3 Funzioni 14

4.3.1 Settare la direzione dei dati 14

4.3.2 Assegnazione dei valori logici ai bit.....15

4.3.3 Altre funzioni 16

CAPITOLO 5 Guida alla realizzazione degli esperimenti: lampeggio di led 17

5.1 Esperimento: lampeggio di un singolo led.....17

5.2 Esperimento: semaforo.....20

5.3 Esperimento: lampeggio di otto led montati su un componente a logica negativa.....22

CAPITOLO 6 Guida alla realizzazione degli esperimenti: matrice di led 8x8.....24

6.1 Breve descrizione del componente 24

6.2 Libreria 8x8.h.....25

6.2.1 File 8x8.c.....26

6.3 Esperimento: disegnare sulla matrice led 8x8.....27

CAPITOLO 7 Guida alla realizzazione degli esperimenti: display a 7 segmenti 30
7.1 Breve descrizione del componente 30
7.2 Libreria 7segments.h 31
7.2.1 Macro 31
7.2.2 Variabili e funzioni 32
7.2.3 File 7segments.c 33
7.3 Esperimento: scrivere sul display 35

CAPITOLO 8 Guida alla realizzazione degli esperimenti: display LCD 38
8.1 Breve descrizione del componente 38
8.2 Libreria LCD.h 39
8.2.1 Macro 39
8.2.2 Funzioni 40
8.2.3 LCD.c 41
8.3 Esperimento: scrivere sul display 42

CAPITOLO 9 Guida alla realizzazione degli esperimenti: gestione degli interrupt 45
9.1 Registri per la gestione dei Pin Change Interrupt 45
9.2 Libreria common_commands.h 47
9.2.1 Macro 47
9.2.2 Funzioni 48
9.3 Esperimenti basati sulla gestione degli interrupt 49
9.3.1 Esperimento: accensione e spegnimento di un led al segnale mandato da un sensore PIR 50
9.3.2 Esperimento: visualizzazione della notifica di ricezione di un interrupt su un display LCD 53
9.3.3 Esperimento: contatore degli interrupt ricevuti da un encoder a disco 55

CAPITOLO 10 Conclusioni 58

CAPITOLO 1

Introduzione

Arduino è una piattaforma hardware composta da una serie di schede elettroniche dotate di microcontrollore [1]. È una scheda open source che permette in maniera relativamente semplice la creazione di piccoli dispositivi come controllori di luci, di velocità per motori, sensori di luce, automatismi per il controllo della temperatura e dell'umidità e molti altri progetti che utilizzano sensori, attuatori e comunicazione con altri dispositivi [2].

Tra le componenti principali di una scheda Arduino e che costituiscono inoltre il centro dell'analisi per lo sviluppo di questa tesi, sono presenti il microcontrollore e i pin di I/O. Un Arduino infatti, rappresenta un'interfaccia programmabile per il microcontrollore che monta su di esso.

I progetti che verranno illustrati in questa tesi sono stati implementati e testati su Arduino Uno che monta un microcontrollore Atmel ATMEGA328P.

Per questo motivo infatti, il relativo codice e assetto delle componenti per ogni progetto sono stati realizzati basandosi sulle caratteristiche presenti nel datasheet di questo microcontrollore [3].

Figura 1.1: Logo di Arduino [2]

L'ambiente di sviluppo per la programmazione di questa piattaforma è Arduino (pari al nome della scheda stessa). È fornito di una libreria software C/C++ che rende molto più semplice implementare le comuni operazioni di input/output nascondendo al programmatore l'interazione con i registri e l'assegnazione dei valori alle porte relative ad ogni pin.

I progetti (verranno anche chiamati esperimenti) che verranno illustrati nelle prossime pagine sono stati realizzati in linguaggio C puro, senza l'utilizzo delle librerie ausiliarie messe a disposizione da Arduino (IDE), andando così a interagire direttamente con i registri della scheda hardware.

Questi esperimenti risulteranno utili per fini didattici nel corso di Calcolatori Elettronici

e per tale motivo è stato scelto questo tipo di approccio al fine di invitare gli studenti a una più profonda comprensione degli argomenti trattati dalla materia.

Non avendo implementato il codice attraverso l'IDE Arduino, è stato utilizzato il compilatore AVR per la compilazione, a cui verrà dedicato un capitolo al fine di spiegarne l'uso.

Ogni progetto è caratterizzato dalla comunicazione tra Arduino e un altro dispositivo, perciò come verrà illustrato in seguito, è stata realizzata una libreria per ogni componente che ne permetta un più facile utilizzo e comprensione di quest'ultimo.

Durante la fase di sperimentazione di questa tesi, è risultato evidente come alcuni comandi si ripetessero in buona parte delle operazioni eseguite su ogni componente, perciò è stata realizzata un'ulteriore libreria in cui vengono definiti metodi e comandi comuni a tutti gli esperimenti, chiamata appunto *common_commands.h*, le cui caratteristiche in dettaglio verranno spiegate nei capitoli successivi.

In tutte le operazioni che verranno eseguite, verranno utilizzati valori binari, ma in realtà è possibile anche lavorare con numeri decimali o esadecimali. Si è scelto di questo tipo di approccio perché rende più intuitiva la comprensione di ciò che avviene nella comunicazione tra i dispositivi elettronici.

Durante la spiegazione delle caratteristiche dei vari dispositivi esterni all'Arduino, alcune di esse non verranno spiegate nel dettaglio in quanto risultano non inerenti all'argomento trattato da questa tesi, che si focalizza soprattutto invece, sulla comunicazione che avviene tra i dispositivi, per comprendere al meglio il corso per cui è stata pensata.

Nella parte finale verrà spiegata inoltre l'implementazione del codice di progetti più complessi, caratterizzati dalla comunicazione tra più dispositivi oltre ad Arduino. In questo caso non è stato necessario creare una libreria apposita, ma come vedremo, verrà richiamata per ogni componente la rispettiva libreria.

CAPITOLO 2

AVR-GCC e AVR-DUDE

L'AVR è una famiglia di microcontrollori RISC sviluppati dalla Atmel, e uno di essi, il modello ATMEGA328P, è montato su Arduino Uno.

Atmel ha messo a disposizione i software AVR-GCC e AVR-DUDE per la compilazione del codice e il caricamento di quest'ultimo sull'Arduino.

In questo capitolo verranno spiegate l'installazione e l'utilizzo di questi due software su sistema operativo Debian/Ubuntu Linux.

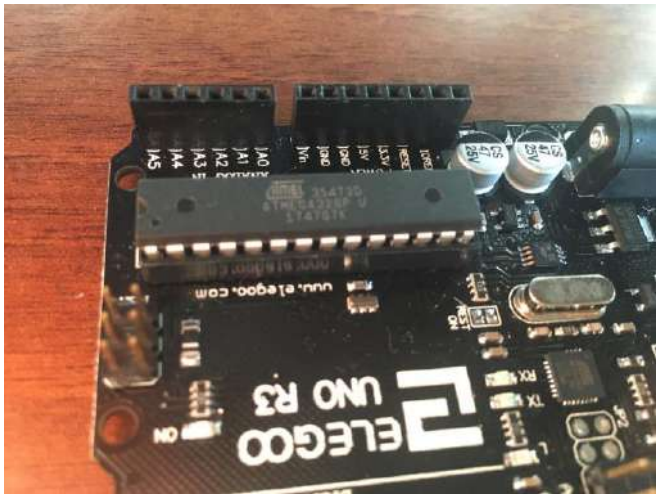


Figura 2.1: Microcontrollore Atmel ATMEGA328P

2.1 Installazione

2.1.1 Installazione AVR-GCC

Per l'installazione del compilatore AVR-GCC, eseguire i seguenti comandi da terminale [4]:

- sudo apt-get update
- sudo apt-get upgrade all
- sudo apt-get install gcc-avr binutils-avr avr-libc
- sudo apt-get install gdb-avr

2.1.2 Installazione AVR-DUDE

Per l'installazione del software AVR-DUDE, eseguire il seguente comando da terminale [4]:

- sudo apt-get install avrdude

2.2 Compilazione e caricamento del codice sorgente

2.2.1 Da un singolo file sorgente all'Arduino

Dopo aver collegato Arduino alla porta USB del pc e aver dato i diritti di accesso al dispositivo per l'utente corrente, supponendo di voler compilare ed eseguire un file *led.c,* lanciare i seguenti comandi da terminale [5]:

- avr-gcc -Os -DF_CPU=16000000UL -mmcu=atmega328p -c -o led.o led.c
- avr-gcc -mmcu=atmega328p led.o -o led
- avr-objcopy -O ihex -R .eeprom led led.hex
- avrdude -F -V -c arduino -p ATMEGA328P -P /dev/ttyACM0 -b 115200 -U flash:w:led.hex

2.2.2 Compilazione e caricamento di più file sorgente su Arduino

Dopo aver collegato Arduino alla porta USB del pc e aver dato i diritti di accesso al dispositivo per l'utente corrente, supponendo di voler compilare ed eseguire i file *led.c* e *common_commands.c*, lanciare i seguenti comandi da terminale:

- avr-gcc -Os -mmcu=atmega328p -DF_CPU=16000000UL -o led.o led.c common_commands.c
- avr-ld -o led.elf led.o
- avr-objcopy -j .text -j .data -O ihex led.o led.hex
- avrdude -F -V -c arduino -p ATMEGA328P -P /dev/ttyACM0 -b 115200 -U flash:w:led.hex

2.2.3 Breve spiegazione dei comandi per la compilazione e caricamento

Il comando avr-gcc si occupa di compilare il file sorgente C in un file oggetto. Le opzioni che seguono indicano al compilatore la frequenza di clock (utile per le funzioni di delay) e il modello del processore per cui compilare il codice. Lo stesso comando nel paragrafo 2.2.1 e il comando avr-ld nel paragrafo 2.2.2, si occupano inoltre della fase di link tra il file oggetto e le librerie di sistema, creando infine un file .ELF contenente il codice macchina.

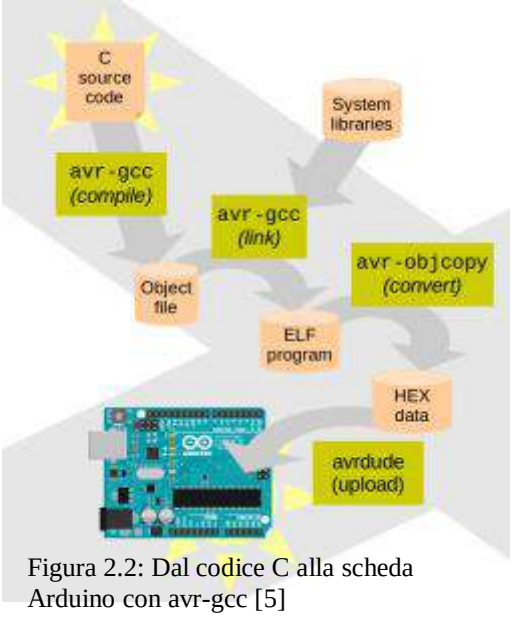


Figura 2.2: Dal codice C alla scheda Arduino con avr-gcc [5]

Il comando avr-objcopy invece, converte il file .ELF in un file .HEX, che contiene il codice binario.

Il comando avrdude infine, carica il file .HEX sulla memoria flash del chip ATMEGA. Le opzioni che seguono indicano al programma avrdude di comunicare usando il protocollo seriale Arduino, attraverso una particolare porta seriale "/dev/ttyACM0" (nome che Linux assegna al dispositivo dopo averlo collegato al pc) e di usare 115200bps come velocità di trasmissione dati [5].

CAPITOLO 3

Pin mapping e interazione con i registri

Un ruolo fondamentale per la realizzazione di un progetto con Arduino è rappresentato dai pin I/O della scheda, che attraverso la loro attribuzione dei valori logici 1 (alta tensione, di solito 5V) o 0 (bassa tensione, di solito 0V), permettono la comunicazione con i dispositivi esterni**.**

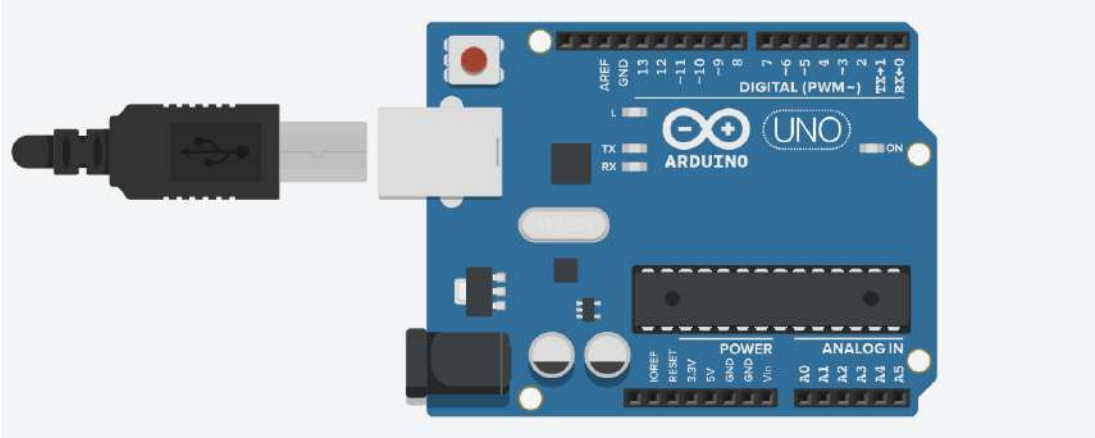


Figura 3.1: Arduino Uno [6]

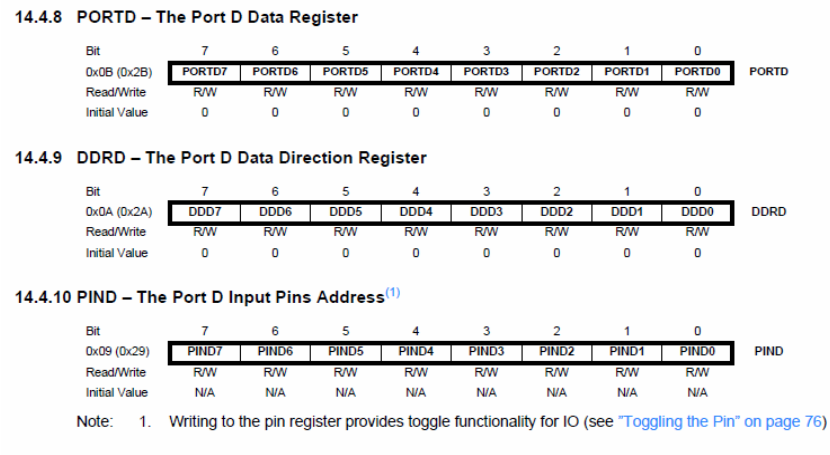
Per accedere ai pin e assegnare quindi loro un valore logico, si utilizzano le porte, che in ATMEGA sono tre:

- Porta B: comprende i pin digitali dal numero 8 al 13.
- Porta C: comprende i bit analogici di input (A0-A5).
- Porta D: comprende i pin digitali dal numero 0 al 7.

Ogni porta permette di manipolare i relativi pin attraverso opportuni registri:
locazioni di memoria note al microcontrollore avente ognuna uno specifico compito.
In Arduino Uno sono presenti tre tipi di registri. Prendendo come riferimento la porta B essi sono:

- DDRB: registro che permette di settare un pin come input o output.
- PORTB: registro che permette di assegnare un valore ad un pin in output

- PINB: registro che permette di leggere il valore di un pin in input. Su questo registro è permessa la sola lettura e non la scrittura. Questo registro non verrà preso in esame nello sviluppo di questa tesi in quanto essa si concentrerà su una gestione dell'input ad "Interrupt", che verrà spiegata nei prossimi capitoli.



14.4.8 PORTD – The Port D Data Register

Bit	7	6	5	4	3	2	1	0	
0x0B (0x2B)	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	PORTD
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	

14.4.9 DDRD – The Port D Data Direction Register

Bit	7	6	5	4	3	2	1	0	
0x0A (0x2A)	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	DDRD
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	

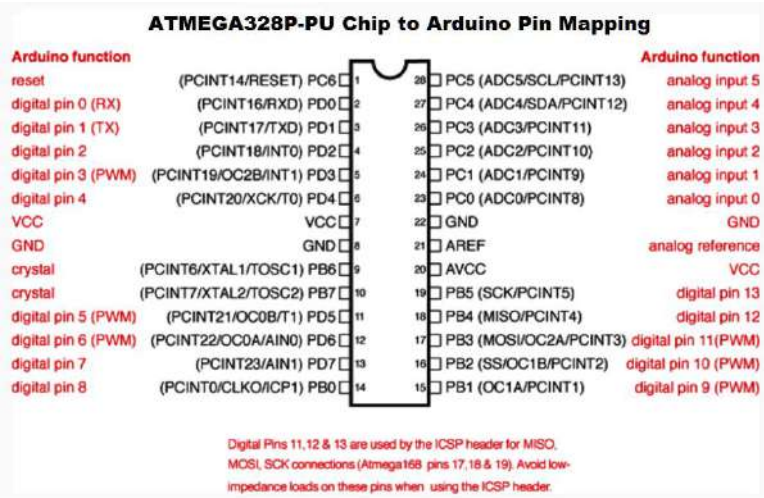
14.4.10 PIND – The Port D Input Pins Address(1)

Bit	7	6	5	4	3	2	1	0	
0x09 (0x29)	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	PIND
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

Note: 1. Writing to the pin register provides toggle functionality for IO (see "Toggling the Pin" on page 76)

Figura 3.2: Descrizione dei registri per la porta D [3]

È possibile comunicare con i registri assegnando ad essi 8 bit (figura 3.2). Ogni bit di una porta si riferisce a un pin di Arduino, come mostrato dal pin mapping in figura 3.3.



Facendo riferimento alla figura 3.3 ad esempio, il pin 9 corrisponde a PD7, ovvero bit 7 della Porta D.

In figura 3.4 è illustrato un esempio d'uso dei registri per rendere più chiara tale spiegazione.

```
1  void main (void)
2  {
3  /* settiamo il pin13 (PB5) come output*/
4      DDRB = DDRB | 0b00100000;
5
6      while(1) {
7  /* accendi il led */
8          PORTB = PORTB | 0b00100000;
9          _delay_ms(500);
10
11 /* spegni il led */
12         PORTB = PORTB & 0b11011111;
13         _delay_ms(500);
14     }
15
16 }
```

Figura 3.4: Esempio di lampeggio
di un led collegato al pin 13 [SublimeText2]

In riga 4 viene settato il pin 13 in output assegnando il valore logico 1 al bit 5 del registro DDRB (bit 5 Porta B, figura 3.3).

In riga 8 e 12 viene acceso e spento il led collegato al pin assegnando rispettivamente 1 e 0 al bit 5 del registro PORTB.

Da notare l'utilizzo di operatori logici AND (&) e OR (|) che permettono di modificare solo il bit che stiamo considerando, lasciando inalterati tutti gli altri [7].

CAPITOLO 4

Libreria common_commands.h:

semplici operazioni di I/O

Come quanto detto già nell'introduzione, è stata realizzata una libreria di nome *common_commands.h*, che racchiude i comandi comuni a tutti i progetti, con l'obiettivo di rendere più semplice e intuiva la comprensione dell'implementazione. In questo capitolo verranno illustrati e commentati tutti i metodi e definizioni presenti nella libreria, utili ai primi esperimenti che verranno spiegati, tralasciando per ora tutta la parte di gestione degli Interrupt.

4.1 Librerie esterne

La figura 4.1 mostra le prime righe, in cui la libreria viene definita. Nelle righe 16-17-18 vengono invece incluse le librerie esterne:

- avr/io.h: libreria di sistema fondamentale per le operazioni di I/O
- util/delay.h: libreria utile per la chiamata della funzione _delay_ms() che verrà spesso utilizzata.

```
12  #ifndef _COMMON_COMMANDS_H_
13  #define _COMMON_COMMANDS_H_
14
15
16  #include <avr/io.h>
17  #include <util/delay.h>
18  #include <avr/interrupt.h>
19
20  #define BYTE unsigned char
21  #define interrupt_handler(vector) ISR(vector)
22
```

Figura 4.1: Righe 12-22 del file common_commands.h [SublimeText2]

- avr/interrupt.h: libreria di sistema fondamentale per la gestione degli Interrupt, che vedremo nei prossimi capitoli.

4.2 Definizioni

4.2.1 Macro

Al fine di rendere più intuitiva la comprensione delle operazioni bit a bit che verranno effettuate, è stata definita una macro (riga 20, figura 4.1) che permette di assegnare a una variabile un "BYTE" invece che un "unsigned char" .

L'utilità della macro in riga 21 verrà spiegata in seguito.

4.3 Funzioni

Dalla riga 38 in avanti vengono dichiarate funzioni quasi sempre utilizzate in tutti gli esperimenti. Di seguito verranno spiegate le modalità d'utilizzo per ognuna di esse.

4.3.1 Settare la direzione dei dati

Le sei funzioni in figura 4.2 si occupano di settare la direzione della trasmissione dei dati per ogni porta.
Un esempio di utilizzo può essere il caso in cui si voglia utilizzare il pin 8 come input e il pin 7 come output, procedendo nella maniera che segue.

Dalla figura 3.3 notiamo che il pin 7 e 8 corrispondono rispettivamente al bit 7 della porta D e il bit 0 della porta B, perciò verranno chiamate le funzioni port_setup_set_data_direction_input_bits_port_B() e port_setup_set_data_direction_output_bits_port_D(), passando come parametri:

- Nella prima un BYTE formato da tutti 1 tranne il bit relativo al pin da utilizzare come input, quindi 0b11111110.

- Nella seconda un BYTE formato da tutti 0 tranne il bit relativo al pin da utilizzare come output, quindi 0b10000000.

```
38  void port_setup_set_data_direction_output_bits_port_B(BYTE code);
39
40  void port_setup_set_data_direction_output_bits_port_C(BYTE code);
41
42  void port_setup_set_data_direction_output_bits_port_D(BYTE code);
43
44  void port_setup_set_data_direction_input_bits_port_B(BYTE code);
45
46  void port_setup_set_data_direction_input_bits_port_C(BYTE code);
47
48  void port_setup_set_data_direction_input_bits_port_D(BYTE code);
```

Figura 4.2: Righe 38-48 del file common_commands.h [SublimeText2]

La figura 4.3 mostra a titolo d'esempio il corpo della funzione port_setup_set_data_direction_output_bits_port_B().
Da notare infatti che all'interno della funzione non avviene altro che ciò che veniva spiegato nel capitolo 3.

```
3  void port_setup_set_data_direction_output_bits_port_B(BYTE code)
4  {
5      DDRB = DDRB | code;
6  }
```

Figura 4.3: Righe 3-7 del file common_commands.c [SublimeText2]

4.3.2 Assegnazione dei valori logici ai bit

Le sei funzioni in figura 4.4 si occupano di assegnare i valori logici HIGH e LOW (1 e 0) ai bit delle porte relativi ai pin.
Un esempio di utilizzo può essere il caso in cui si voglia assegnare al pin 8 il valore LOW e al pin 7 il valore HIGH, procedendo nella maniera che segue.

Dalla figura 3.3 notiamo che il pin 7 e 8 corrispondono rispettivamente al bit 7 della porta D e il bit 0 della porta B, perciò verranno chiamate le funzioni port_set_low_bits_port_B() e port_set_high_bits_port_D(), passando come parametri:

- Nella prima un BYTE formato da tutti 1 tranne il bit relativo al pin a cui si vuole assegnare LOW, quindi 0b11111110.

- Nella seconda un BYTE formato da tutti 0 tranne il bit relativo al pin a cui si vuole assegnare HIGH, quindi 0b10000000.

```
66  void port_set_high_bits_port_B(BYTE code);
67
68  void port_set_high_bits_port_C(BYTE code);
69
70  void port_set_high_bits_port_D(BYTE code);
71
72  void port_set_low_bits_port_B(BYTE code);
73
74  void port_set_low_bits_port_C(BYTE code);
75
76  void port_set_low_bits_port_D(BYTE code);
```

Figura 4.4: Righe 66-76 del file common_commands.h [SublimeText2]

La figura 4.5 mostra a titolo d'esempio il corpo della funzione port_set_high_bits_port_B().

Anche in questo caso infatti all'interno avviene ciò che veniva spiegato nel capitolo 3.

```
33  void port_set_high_bits_port_B(BYTE code)
34  {
35      PORTB = PORTB | code;
36  }
```

Figura 4.5: Righe 33-36 del file common_commands.h [SublimeText2]

4.3.3 Altre funzioni

Le sei funzioni in figura 4.6 hanno lo stesso compito dei metodi spiegati nei due paragrafi precedenti, ma la differenza sostanziale è il mancato utilizzo delle operazioni & e |, sostituite da un'assegnazione diretta come viene mostrato in figura 4.7. È da tenere presente che in questo caso vengono modificati tutti i bit dei registri.

```
86  void port_set_all_bits_port_B(BYTE code);
87
88  void port_set_all_bits_port_C(BYTE code);
89
90  void port_set_all_bits_port_D(BYTE code);
91
92  void port_set_all_bits_data_direction_port_B(BYTE code);
93
94  void port_set_all_bits_data_direction_port_C(BYTE code);
95
96  void port_set_all_bits_data_direction_port_D(BYTE code);
```

Figura 4.6: Righe 86-96 del file common_commands.h [SublimeText2]

```
63  void port_set_all_bits_port_B(BYTE code)
64  {
65      PORTB = code;
66  }
```

Figura 4.7: Righe 63-66 del file common_commands. [SublimeText2]

Le funzioni in figura 4.8 infine, restituiscono il valore attuale dei registri.

```
106  BYTE port_get_bits_port_B();
107
108  BYTE port_get_bits_port_C();
109
110  BYTE port_get_bits_port_D();
```

Figura 4.8: Righe 106-110 del file common_commands.h [SublimeText2]

CAPITOLO 5

Guida alla realizzazione degli esperimenti: lampeggio di led

Per quanto riguarda l'esperimento del lampeggio di led, non è risultato necessario la dichiarazione di ulteriori variabili o metodi, perciò la libreria definita per questo progetto, *lampeggioLED.h,* non fa nient'altro che richiamare *common_command.h* (figura 5.1).

Questa libreria è stata utilizzata per implementare i progetti che verranno spiegati nei paragrafi successivi di questo capitolo.

```
7   #ifndef _LAMPEGGIOLED_H_
8   #define _LAMPEGGIOLED_H_
9
10
11  #include "common_commands.h"
12
13  #endif
```

Figura 5.1: Contenuto del file common_commands.h [SublimeText2]

5.1 Esperimento: lampeggio di un singolo led

In questo esperimento si vuole far lampeggiare un led ogni secondo e per fare ciò sono necessari, oltre alla scheda Arduino:

- Una breadboard
- Un led
- Una resistenza
- Due fili per breadboard

Nel main implentato in quest'esempio verrà utilizzato il pin 8 come output., in quanto connesso al led rosso.

Perciò in questo caso occorrerà effettuare i collegamenti come in figura 5.2

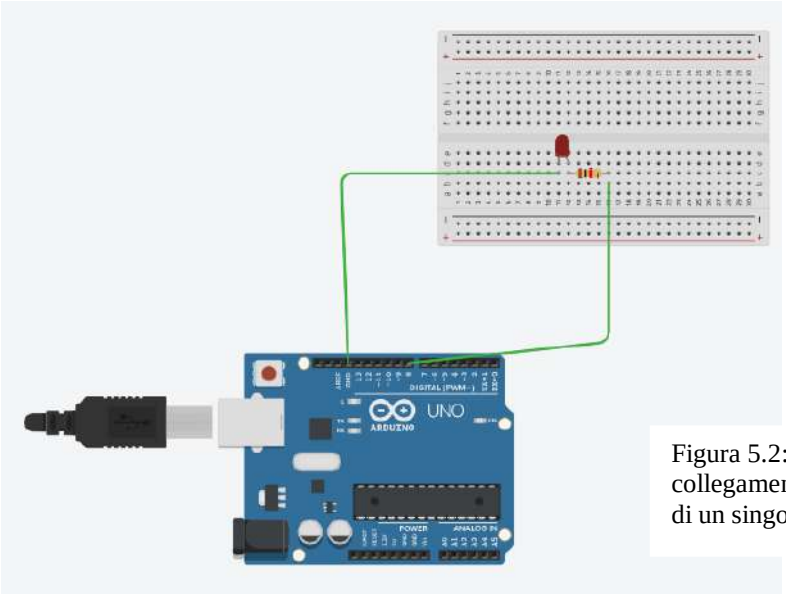


Figura 5.2: Assemblaggio dei collegamenti per il lampeggio di un singolo led [6]

La resistenza è montata per evitare che arrivi troppa corrente al led, impedendo cosi che quest'ultimo possa fulminarsi.

Nel collegare la breadboard ai pin, bisogna fare particolare attenzione nel connettere a massa il catodo del led (il piede più corto) e la tensione alla resistenza che dovrà essere collegata a sua volta all'anodo del led (piede più lungo).

Dopo aver assemblato le parti si può ora caricare il codice su Arduino come spiegato nel paragrafo 2.2.2.

I file da caricare saranno *common_commands.c*, file sorgente della libreria illustrata nel capitolo 4, *lampeggioLED.c*, file sorgente della libreria dell'esperimento attuale, e *lampeggioLED_main.c*, file che contiene un main d'esempio per il funzionamento di questo progetto.

Il contenuto del file *lampeggioLED_main.c* è mostrato in figura 5.3 e di seguito ne verrà spiegato il funzionamento.

Nella riga 1 viene inclusa la libreria realizzata per questo esperimento.

In riga 8 viene richiamata la funzione port_setup_set_data_direction_output_bits_port_B() utilizzata per assegnare il pin 8 in output, come spiegato nel paragrafo 4.3.1.

In riga 11 inizia un ciclo che non smette mai di eseguire ("while(1)"), che verrà utilizzato tra l'altro in tutti i progetti, in cui viene spento e acceso il led ogni secondo,

richiamando le due funzioni port_set_high_bits_port_B() e port_set_low_bits_port_B(), il cui uso è stato spiegato nel paragrafo 4.3.2.

Da notare inoltre l'impiego della funzione _delay_ms(), che ritarda l'esecuzione dell'istruzione successiva dell'intervallo di tempo espresso in millisecondi, passato come parametro.

Confrontando queste istruzioni con il codice che si utilizza con le librerie messe a disposizione dall'IDE Arduino, si nota che la parte in cui si definisce la direzione dei dati corrisponde alla funzione di setup(), mentre il ciclo che segue corrisponde alla funzione di loop().

```
1  #include "lampeggioLED.h"
2
3  void main(void)
4  {
5          //  Output: Bit 0 della porta b (pin 8)
6
7          //  setup del pin 8 come output
8      port_setup_set_data_direction_output_bits_port_B(0b00000001);
9
10
11     while(1)
12     {     //  alterno lo stato del bit 0 della porta b : accendo e spengo led
13         port_set_high_bits_port_B(0b00000001);
14         _delay_ms(1000);
15         port_set_low_bits_port_B(0b00000000);
16         _delay_ms(1000);
17     }
18
19 }
```

Figura 5.3: Contenuto del file lampeggioLED_main.c [SublimeText2]

La figura 5.4 mostra il risultato dell'esperimento dopo aver eseguito i passaggi appena spiegati e aver connesso l'Arduino all'alimentazione.

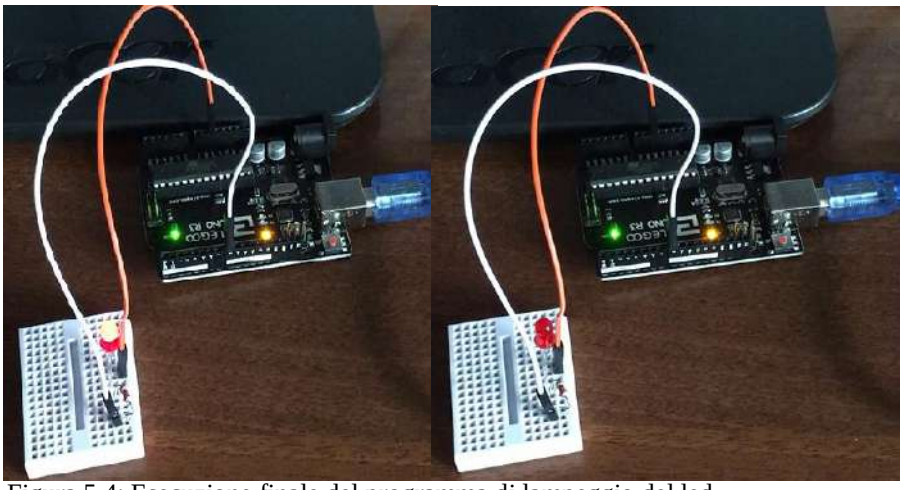


Figura 5.4: Esecuzione finale del programma di lampeggio del led

5.2 Esperimento: semaforo

Si vuole implementare in questo caso una rete simile a quella di un semaforo e per far ciò sono necessari, oltre alla scheda Arduino:

- Una breadboard
- Tre led
- Tre resistenze
- Sette fili per breadboard

Nel main implentato in quest'esempio verranno utilizzati i pin 7, 8 e 12 come output, in quanto collegati rispettivamente al led rosso, giallo e verde.

Perciò in questo caso occorrerà effettuare i collegamenti come in figura 5.5.

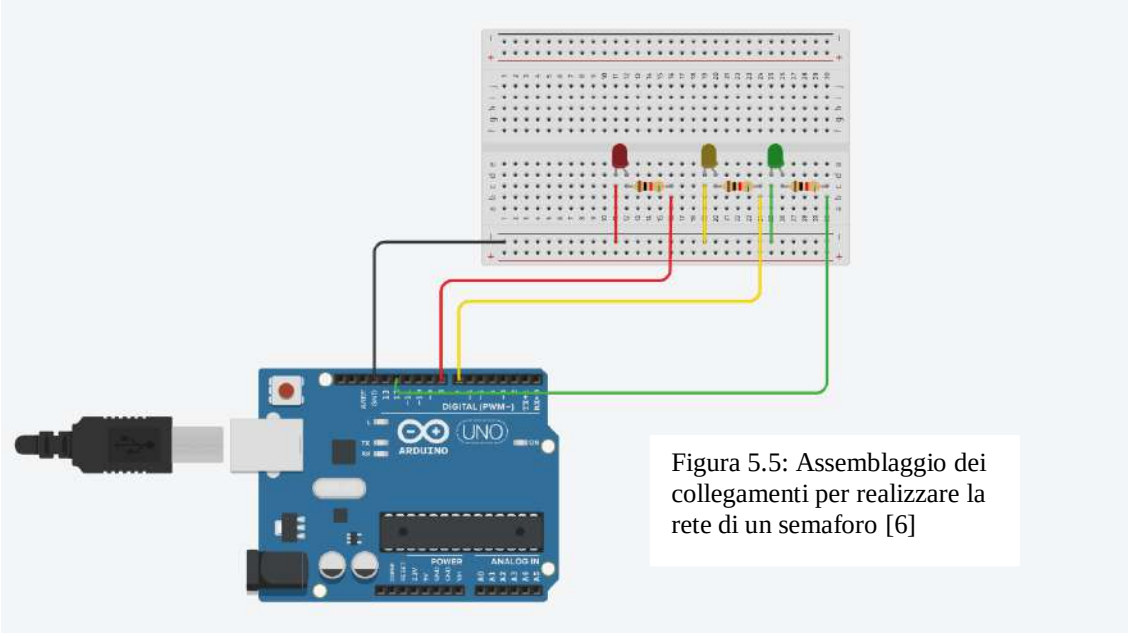


Figura 5.5: Assemblaggio dei collegamenti per realizzare la rete di un semaforo [6]

Anche in questo caso bisogna avere cura di montare le resistenze e fare attenzione ai collegamenti dell'anodo e del catodo dei led, come spiegato nel paragrafo precedente.

Dopo aver assemblato le parti si può ora caricare il codice su Arduino come spiegato nel paragrafo 2.2.2.

I file da caricare saranno *common_commands.c*, file sorgente della libreria illustrata nel capitolo 4, *lampeggioLED.c*, file sorgente della libreria dell'esperimento attuale, e *semaforo_main.c*, file che contiene un main d'esempio per il funzionamento di questo progetto.

Il contenuto del file *semaforo_main.c* è mostrato in figura 5.6 e di seguito ne verrà spiegato il funzionamento.

```
1  #include "lampeggioLED.h"
2
3  void main(void)
4  {
5
6          //  Output: Bit 0 della porta B (pin 8) [LED ROSSO] - Bit 7 della porta D (pin 7) [LED GIALLO] - Bit 4 della porta B (pin 12)
7
8          //  setup del bit 0 e bit 4 della porta b, bit 7 della porta D come OUTPUT
9      port_setup_set_data_direction_output_bits_port_B(0b00010001);
10     port_setup_set_data_direction_output_bits_port_D(0b10000000);
11     while(1)
12     {
13             //  alternanza led
14         port_set_high_bits_port_B(0b00000001);
15         _delay_ms(5000);
16         port_set_low_bits_port_B(0b11111110);
17         port_set_high_bits_port_B(0b00010000);
18         _delay_ms(5000);
19         port_set_low_bits_port_B(0b11101111);
20         port_set_high_bits_port_D(0b10000000);
21         _delay_ms(2000);
22         port_set_low_bits_port_D(0b01111111);
23     }
24
25 }
```

Figura 5.6: Contenuto del file semaforo_main.c [SublimeText2]

Nella riga 1 viene inclusa la libreria realizzata per questo esperimento.

In riga 9 vengono richiamate le funzioni port_setup_set_data_direction_output_bits_port_B() e port_setup_set_data_direction_output_bits_port_D() utilizzate per direzionare i pin 7,8 e 12 in output, come spiegato nel paragrafo 4.3.1.

In riga 11 inizia il ciclo infinito in cui, al fine di simulare il funzionamento di un semaforo, vengono spenti e accesi i led seguendo l'ordine delle luci di quest'ultimo, richiamando le funzioni port_set_high_bits_port_B(), port_set_low_bits_port_B(), port_set_high_bits_port_D() e port_set_low_bits_port_D().

La figura 5.7 mostra il risultato dell'esperimento dopo aver eseguito i passaggi appena spiegati e aver connesso l'Arduino all'alimentazione.

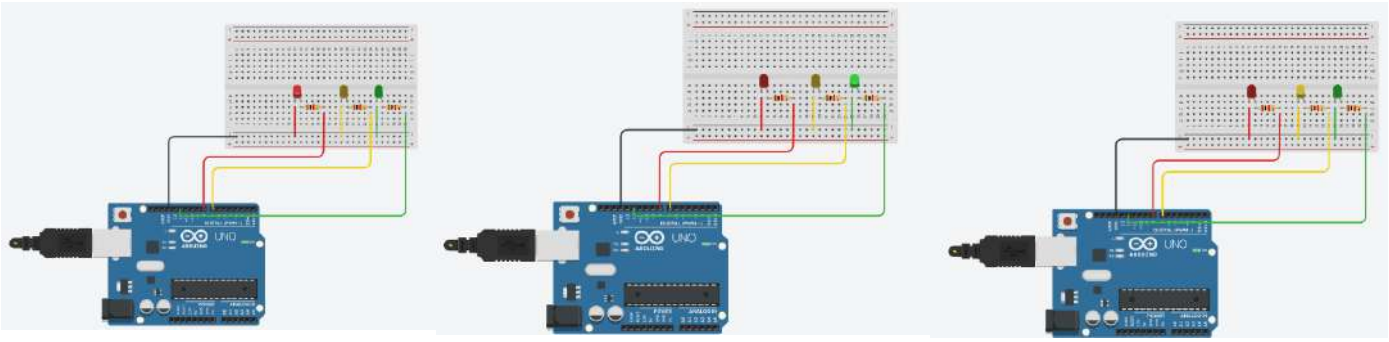


Figura 5.7: Esecuzione finale del programma semaforo [6]

5.3 Esperimento: lampeggio di otto Led montati su un componente a logica negativa

In questo progetto si vuole implementare il codice per l'alternanza tra accensione e spegnimento di otto led montati su di un unico componente (figura 5.8).

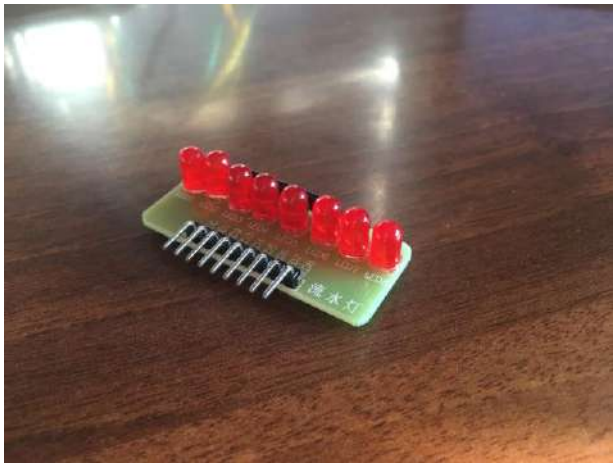


Figura 5.8: Otto led montati su di un unico componente collegato a tensione massima

In questo caso sono necessari, oltre alla scheda Arduino:

- Una breadboard
- Il componente con gli 8 led
- Una resistenza
- Dieci fili per breadboard

Nel main implentato in quest'esempio verranno utilizzati i pin da 0 a 7 come output, perciò occorrerà effettuare i collegamenti come in figura 5.9.

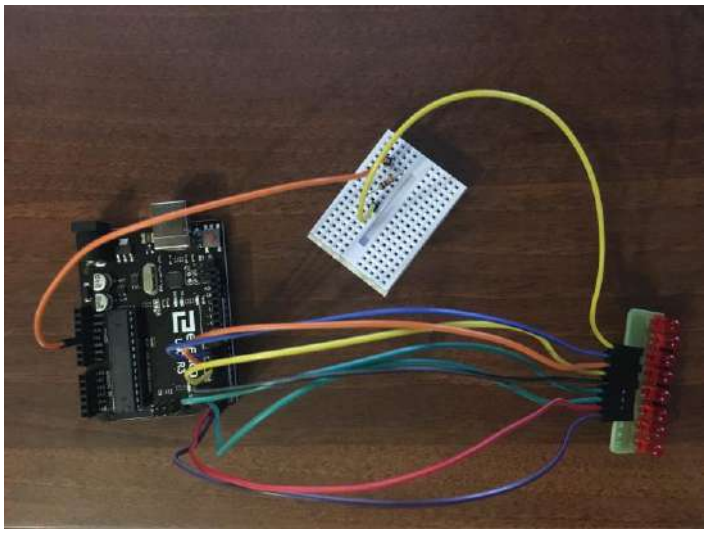


Figura 5.9: Assemblaggio dei collegamenti per far lampeggiare gli otto

Il componente richiede inoltre di esser collegato a tensione massima (5V, tenendo cura anche stavolta di utilizzare una resistenza), infatti esso sfrutta

una logica negativa, andando così a invertire la logica delle istruzioni da implementare come mostra la figura 5.10: per accendere un led viene settato il relativo bit a LOW, in caso contrario invece, viene settato il relativo bit al valore logico HIGH.

Dopo aver assemblato le parti si può ora caricare il codice su Arduino come spiegato nel paragrafo 2.2.2.

I file da caricare saranno *common_commands.c*, file sorgente della libreria illustrata nel capitolo 4, *lampeggioLED.c*, file sorgente della libreria dell'esperimento attuale, e *lampeggio8LED_main.c*, file che contiene un main d'esempio per il funzionamento di questo progetto.

L'implementazione del main è molto simile alle precedenti, perciò non verrà in questa caso spiegata.

```c
#include "lampeggioLED.h"

void main(void)
{
        //  Output: Bit 0-7 della porta D (pin 0-7)

        //  setup dei pin 0-7 come output
    port_setup_set_data_direction_output_bits_port_D(0b11111111);


    while(1)
    {       //  accendo/spengo i led [QUESTO COMPONENTE UTILIZZA UNA LOGICA NEGATIVA]
        port_set_high_bits_port_D(0b11111111);
        port_set_low_bits_port_D(0b01111110);
        _delay_ms(1000);
        port_set_high_bits_port_D(0b11111111);
        port_set_low_bits_port_D(0b10111101);
        _delay_ms(1000);
        port_set_high_bits_port_D(0b11111111);
        port_set_low_bits_port_D(0b11011011);
        _delay_ms(1000);
        port_set_high_bits_port_D(0b11111111);
        port_set_low_bits_port_D(0b11100111);
        _delay_ms(1000);
    }

}
```

Figura 5.10: Contenuto del file lampeggio8LED_main.c [SublimeText2]

La figura 5.11 mostra il risultato dell'esperimento dopo aver eseguito i passaggi appena spiegati e aver connesso l'Arduino all'alimentazione.

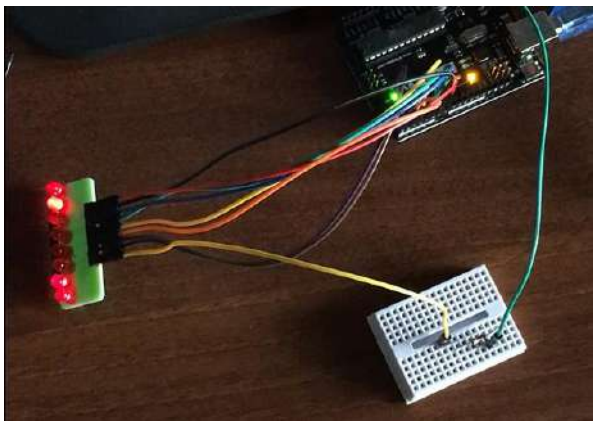


Figura 5.11: Esecuzione del programma di lampeggio degli otto led

CAPITOLO 6

Guida alla realizzazione degli esperimenti:

matrice di led 8x8

Questo progetto è caratterizzato dallo scambio di dati tra l'Arduino e un dispositivo particolare, formato da una matrice di led 8x8 e un microcontrollore (figura 6.1). Come molti altri componenti infatti, di logica più complessa rispetto ai precedenti, esso utilizza un microcontrollore per risparmiare pin (ne sarebbero serviti sessantaquattro in questo caso) e scandire lo scambio dei dati attraverso un CLK (clock) e un CS (chip select).

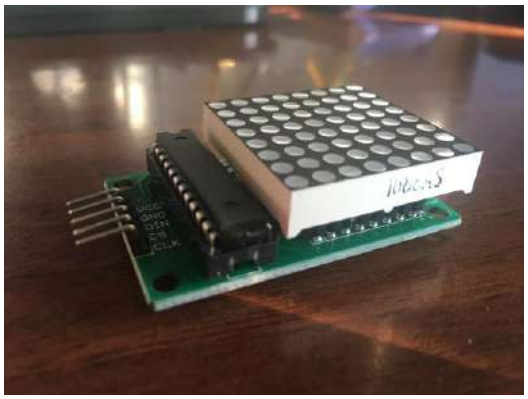


Figura 6.1: Matrice di led 8x8 con microcontrollore Maxim MAX7219CNG

6.1 Breve descrizione del componente

In questo caso, ogni singolo led non è direttamente collegato all'Arduino come avveniva in precedenza, perciò la loro accensione sarà leggermente più complessa. Si accede ad ogni singolo led attraverso un "indirizzo", che corrisponde al numero della colonna del dispositivo (matrice orientata con i pin VCC, GND, CLK, DIN, CS rivolti verso l'alto): all'indirizzo 0b00000001 (numero 1 decimale in binario) corrisponderà la prima colonna della matrice.

Una volta scelto l'indirizzo si può accedere agli otto led della rispettiva colonna attraverso un BYTE, accendendoli o spegnendoli assegnando loro un valore 0 o 1, come ad esempio: assegnare all'indirizzo 0b00000010 il BYTE 0b11111111 significa accendere tutti i led della seconda colonna.

I dati vengono inviati al dispositivo tramite il pin DIN e campionati su un fronte di salita del CLK. I led si accenderanno poi su un fronte di salita del CS.

6.2 Libreria 8x8.h

Al fine di semplificare l'utilizzo di questo dispositivo e nascondere i passaggi più complessi, è stata realizzata per questo componente la libreria *8x8.h*. Oltre che richiamare la *common_commands.h,* essa dichiara le due funzioni:

- void led_matrix_8x8_write_data_output(BYTE address, BYTE data):
  permette di scrivere i dati all'indirizzo passato come parametro.
- void led_matrix_8x8_set_default_setting(): setta le impostazioni di default, inviando i dati a indirizzi particolari che il dispositivo si aspetta prima dell'esecuzione delle istruzioni. Vengono impostati:
  - Luminosità pari a 2 su 15 valori possibili.
  - Decodifica dei dati disabilitata
  - Massimo numero di colonne da mostrare
  - Display acceso
  - Test del display disabilitato

  Questa parte d'impostazioni è una caratteristica che, come detto nell'introduzione, non verrà spiegata nel dettaglio per i motivi già indicati. Questi valori inoltre, risultano sufficienti per lo sviluppo di questo progetto e non verranno mai modificati.

```
7   #ifndef _8x8_H
8   #define _8x8_H
9   #include "common_commands.h"
10
11      // CLK -> pin 12 (PB4), DIN -> pin 10 (PB2), CS -> pin 11 (PB3)
12
13  /*
14  *   Scrive i dati all'indirizzo passato come parametro
15  *
16  */
17  void led_matrix_8x8_write_data_output(BYTE address, BYTE data);
18
19  /*
20  *
21  *   Setta impostazioni di default
22  *
23  */
24  void led_matrix_8x8_set_default_setting();
25
26  #endif
```

Figura 6.2: Contenuto del file 8x8.h [SublimeText2]

La libreria è funzionante solo nel caso in cui vengano collegati i fili come indicato in riga 11 della figura 6.2, in quanto è stata realizzata interagendo con le porte relative a quei pin.

6.2.1 File 8x8.c

Nel file *8x8.c*, oltre a implementare i metodi presentati nel paragrafo precedente, viene dichiarata e implementata un'ulteriore funzione statica non visibile all'esterno della libreria, utile alla funzione led_matrix_8x8_write_data_output(), la cui firma è la seguente:

- static void led_matrix_8x8_write_byte(BYTE data): si occupa di inviare sul dispositivo un solo BYTE, inviando un bit alla volta.

Di seguito, le figure 6.3, 6.4 e 6.5 spiegheranno il funzionamento di ogni funzione riga per riga.

Figura 6.3: Corpo della funzione led_matrix_8x8_write_byte() [SublimeText2]

```
33  void led_matrix_8x8_write_byte(BYTE data)
34  {
35      int counter_bit;
36          //  Invio un bit alla volta
37      for(counter_bit=0; counter_bit < 8; counter_bit++)
38      {
39              //  CLK -> LOW
40          port_set_low_bits_port_B(0b11101111);
41
42              // Discrimino se mandare 0 o 1
43          if((data & 0b10000000) == 0)
44              port_set_low_bits_port_B(0b11111011);
45          else
46              port_set_high_bits_port_B(0b00000100);
47              // Shifto al prossimo bit
48          data = data << 1;
49              //  CLK -> HIGH
50          port_set_high_bits_port_B(0b00010000);
51      }
52  }
```

Il corpo della funzione è formato da un ciclo che compie otto giri. Per ogni BYTE infatti, viene mandato un bit alla volta al dispositivo tramite il pin DIN(data input): in riga 43 si va a verificare se il bit più significativo sia uno 0 o un 1, mandando cosi valore logico LOW nel primo caso o HIGH nel secondo, facendo uno shift a sinistra ad ogni giro. Le righe 40 e 50 invece servono a generare un fronte di salita del CLK, perché come già spiegato nel paragrafo precedente, il microcontrollore campiona i dati solo in quel momento. Alla fine degli otto giri verrà così inviato tutto il BYTE.

```
 7  void led_matrix_8x8_write_data_output(BYTE address, BYTE data)
 8  {
 9          //  CS -> LOW
10      port_set_low_bits_port_B(0b11110111);
11          //  Passo l'indirizzo su cui scrivere i dati
12      led_matrix_8x8_write_byte(address);
13          //  Scrivo i dati
14      led_matrix_8x8_write_byte(data);
15          //  CS -> HIGH
16      port_set_high_bits_port_B(0b00001000);
17  }
```

Figura 6.4: Corpo della funzione led_matrix_8x8_write_data_output() [SublimeText2]

Il corpo della funzione led_matrix_8x8_write_data_output() invece è molto semplice, non fa altro che scrivere sul dispositivo gli indirizzi e i relativi dati, richiamando il metodo led_matrix_8x8_write_byte, e generare infine un fronte di salita del CS.

```
19  void led_matrix_8x8_set_default_setting()
20  {
21          //  decode (always no decode)
22      led_matrix_8x8_write_data_output(0b00001001, 0b00000000);
23          //  Brightness -> 2 (0..15)
24      led_matrix_8x8_write_data_output(0b00001010, 0b00000010);
25          //  Display digits (always 7, digit = column)
26      led_matrix_8x8_write_data_output(0b00001011, 0b00000111);
27          //  Shutdown (0x01 Normal | 0x00 Shutdown)
28      led_matrix_8x8_write_data_output(0b00001100, 0b00000001);
29          //  Display test (0x01 Test | 0x00 Normal)
30      led_matrix_8x8_write_data_output(0b00001111, 0b00000000);
31  }
```

Figura 6.5: Corpo della funzione led_matrix_8x8_set_default_setting() [SublimeText2]

La figura 6.5 mostra invece il metodo led_matrix_8x8_set_default_setting() che setta le impostazioni di default richiamando la funzione in figura 6.4.

6.3 Esperimento: disegnare sulla matrice led 8x8

L'obiettivo di questo esperimento è quello di disegnare sulla matrice una faccia felice e una triste, alternandole ogni due secondi.

In questo caso sono necessari, oltre alla scheda Arduino:

- Matrice di led 8x8 con microcontrollore MAX7219CNG
- Cinque fili per breadboard

Nel main implementato in quest'esempio verranno utilizzati i pin 12, 10 e 11 come output, in quanto collegati rispettivamente ai pin CLK, DIN e CS. Bisogna inoltre connettere la VCC alla tensione massima di 5V e il pin GND a massa. Perciò occorrerà effettuare i collegamenti come in figura 6.6.

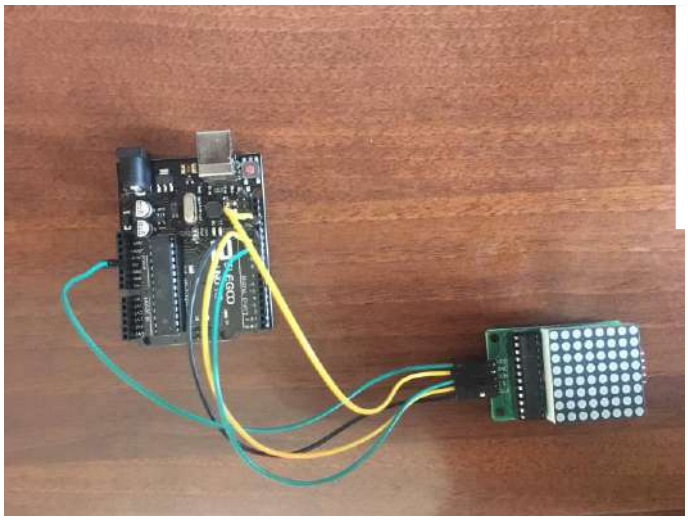


Figura 6.6: Assemblaggio dei collegamenti per disegnare sulla matrice led 8x8 con microcontrollore MAX7219CNG

Dopo aver assemblato le parti si può ora caricare il codice su Arduino come spiegato nel paragrafo 2.2.2.

I file da caricare saranno *common_commands.c*, file sorgente della libreria illustrata nel capitolo 4, *8x8.c*, file sorgente della libreria dell'esperimento attuale, e *8x8_main.c*, file che contiene un main d'esempio per il funzionamento di questo progetto.

Il contenuto del file *8x8_main.c* è mostrato in figura 6.7 e di seguito ne verrà spiegato il funzionamento.

Nella riga 1 viene inclusa la libreria realizzata per questo esperimento.

In riga 8 vengono settati i pin 10, 11 e 12 in output, collegati rispettivamente a DIN, CS e CLK. Quest'ultimi vengono inoltre posti a LOW come setup nella riga 11.

Successivamente vengono settate le impostazioni di default richiamando la funzione led_matrix_8x8_set_default_setting() definita nella libreria *8x8.h*.

Infine, inizia il ciclo infinito in cui si alternano la faccia felice e quella triste, generate richiamando la funzione led_matrix_8x8_write_data_output(), disegnandole seguendo il procedimento spiegato nel paragrafo 6.1.

Nelle righe 28 e 37 viene richiamata la funzione _delay_ms(), il cui funzionamento è stato già spiegato più volte nei capitoli precedenti.

```c
#include "8x8.h"

    //  Output: CLK -> pin 12 (PB4), DIN -> pin 10 (PB2), CS -> pin 11 (PB3)

void main()
{
        //  CLK -> OUTPUT, DIO -> OUTPUT, CS -> OUTPUT
    port_setup_set_data_direction_output_bits_port_B(0b00011100);

        //  CLK -> LOW, DIO -> LOW, CS -> LOW
    port_set_low_bits_port_B(0b11100011);

        //  Setto impostazioni di default
    led_matrix_8x8_set_default_setting();

        //  Scrivo i dati (accendo led)

    while(1)
    {
        led_matrix_8x8_write_data_output(0b00000001, 0b00111100);
        led_matrix_8x8_write_data_output(0b00000010, 0b01000010);
        led_matrix_8x8_write_data_output(0b00000011, 0b10101001);
        led_matrix_8x8_write_data_output(0b00000100, 0b10000101);
        led_matrix_8x8_write_data_output(0b00000101, 0b10000101);
        led_matrix_8x8_write_data_output(0b00000110, 0b10101001);
        led_matrix_8x8_write_data_output(0b00000111, 0b01000010);
        led_matrix_8x8_write_data_output(0b00001000, 0b00111100);
        _delay_ms(2000);
        led_matrix_8x8_write_data_output(0b00000001, 0b00111100);
        led_matrix_8x8_write_data_output(0b00000010, 0b01000010);
        led_matrix_8x8_write_data_output(0b00000011, 0b10100101);
        led_matrix_8x8_write_data_output(0b00000100, 0b10001001);
        led_matrix_8x8_write_data_output(0b00000101, 0b10001001);
        led_matrix_8x8_write_data_output(0b00000110, 0b10100101);
        led_matrix_8x8_write_data_output(0b00000111, 0b01000010);
        led_matrix_8x8_write_data_output(0b00001000, 0b00111100);
        _delay_ms(2000);
    }

}
```

Figura 6.7: Contenuto del file 8x8_main.c [SublimeText2]

La figura 6.8 mostra il risultato dell'esperimento dopo aver eseguito i passaggi appena spiegati e aver connesso l'Arduino all'alimentazione.

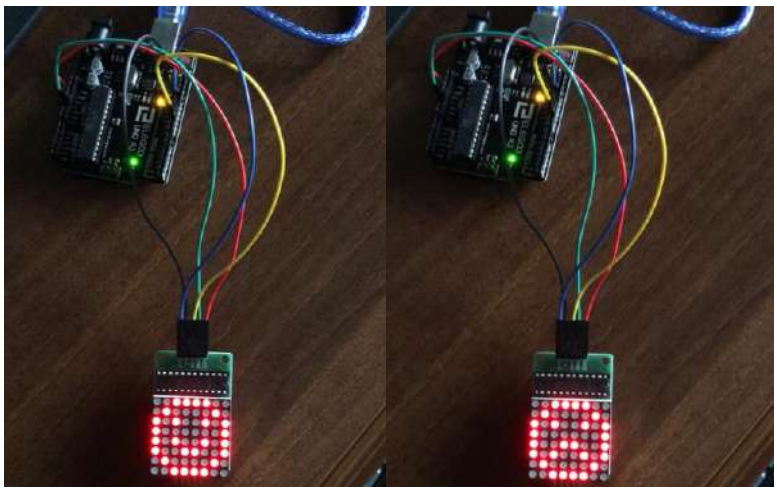


Figura 6.8: Esecuzione del programma che alterna la faccia triste e felice

CAPITOLO 7

Guida alla realizzazione degli esperimenti:

display a 7 segmenti

Verrà illustrato in questo capitolo un progetto caratterizzato dalla connessione diretta tra Arduino e un display a 7 segmenti a quattro cifre (figura 7.1), che monta anch'esso un microcontrollore per le stesse motivazioni spiegate nel capitolo precedente.

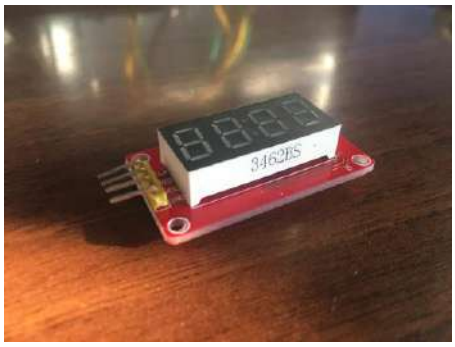


Figura 7.1: Display a 7 segmenti a quattro cifre con microcontrollore TM1637

7.1 Breve descrizione del componente

Questo componente monta solamente quattro pin: VCC, GND, DIO, CLK.

Anche in questo caso la comunicazione con esso risulta più complessa rispetto ai primi componenti presentati, ma nonostante ciò la libreria implementata ne rende più semplice l'interazione. Ogni volta che bisogna inviare un dato al display infatti, esso necessita dell'esecuzione di tre comandi in sequenza, che corrispondono a dei valori in BYTE che vengono inviati al dispositivo: il primo indica al microcontrollore che sta per iniziare l'interazione, il secondo lo avvisa che stanno per essere inviati i dati, indicando inoltre in quale delle quattro cifre assegnarli, il terzo infine avvisa della fine della sequenza dei dati e permette di settare le impostazioni per il display. Il dispositivo restituisce inoltre un ACK per indicare la corretta ricezione dei dati (per questo motivo il pin si chiama DIO – data input/output – e non DIN).

7.2 Libreria 7segments.h

Al fine di semplificare l'utilizzo di questo dispositivo e nascondere i passaggi più complessi, è stata realizzata per questo componente la libreria *7segments.h.*

Le figura 7.2 e 7.3 mostrano il contenuto del file, che verrà spiegando riga per riga nei prossimi paragrafi.

7.2.1 Macro

Dopo aver incluso la libreria *common_commands.h*, vengono definite le seguenti macro per rendere più intuitive le operazioni:

- CONTROLLER_IO_COMMAND1: inviando tale indirizzo al dispositivo, viene indicato l'inizio dell'interazione.
- CONTROLLER_IO_COMMAND2: : inviando tale indirizzo al dispositivo, viene indicato che stanno per essere inviati i dati. Inoltre, ponendo questo BYTE in OR con una delle 4 macro definite nelle righe 18-19-20-21, si decide in quale cifra scrivere i dati, come vedremo in seguito.
- CONTROLLER_IO_COMMAND3: inviando tale indirizzo al dispositivo, viene indicata la fine dei dati da inviare e dell'interazione. Inoltre, ponendo questo BYTE in OR con una delle macro definite nelle righe 15-16 o con la variabile dichiarata in riga 23, è possibile settare delle impostazioni sul display.
- CONTROLLER_SET_DISPLAY_OFF/ON: impostazioni del display.
- DISPLAY_X_ADDR: indica l'indirizzo della cifra X del display.

```
7   #ifndef _7SEGMENTS_H_
8   #define _7SEGMENTS_H_
9
10  #include "common_commands.h"
11
12  #define CONTROLLER_IO_COMMAND1        0b01000000
13  #define CONTROLLER_IO_COMMAND2        0b11000000
14  #define CONTROLLER_IO_COMMAND3        0b10000000
15  #define CONTROLLER_SET_DISPLAY_OFF    0b00000000
16  #define CONTROLLER_SET_DISPLAY_ON     0b00001000
17
18  #define DISPLAY_3_ADDR 3
19  #define DISPLAY_2_ADDR 2
20  #define DISPLAY_1_ADDR 1
21  #define DISPLAY_0_ADDR 0
22
23  static BYTE default_brightness = 7;
24
25  static BYTE digitToSegment[] = {
26
27    0b00111111,    // 0
28    0b00000110,    // 1
29    0b01011011,    // 2
30    0b01001111,    // 3
31    0b01100110,    // 4
32    0b01101101,    // 5
33    0b01111101,    // 6
34    0b00000111,    // 7
35    0b01111111,    // 8
36    0b01101111,    // 9
37    0b01110111,    // A
38    0b01111100,    // b
39    0b00111001,    // C
40    0b01011110,    // d
41    0b01111001,    // E
42    0b01110001     // F
43  };
```

Figura 7.2: Righe 7-43 del file 7segments.h [SublimeText2]

7.2.2 Variabili e Funzioni

In riga 23 e 25 vengono dichiarate le seguenti variabili globali statiche:

- default_brightness: corrisponde al valore della luminosità di default. Questa variabile non viene mai modificata.
- digitToSegment[]: è un array a cui ad ogni indice corrisponde la rappresentazione in binario dell'indice stesso, ad esempio:
  in posizione 0 sarà contenuto un BYTE che corrisponde alla rappresentazione dello zero sul display (figura 7.3).

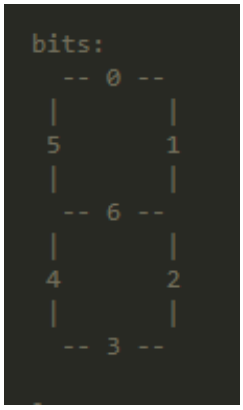


Figura 7.3: Corrispondenza tra i bit
di un BYTE e i led del display

Nelle righe 69 e 76 (figura 7.4) vengono definiti i due metodi accessibili dall'esterno:

- void display_7segments_set_segments(BYTE segments[], BYTE lenght, BYTE starter_position): accende i segmenti sul display passati come parametro a partire dalla posizione iniziale.
- void display_7segments_set_segment(BYTE segment, BYTE position): accende i segmenti di una sola cifra sul display passati come parametro sulla posizione indicata.

La libreria è funzionante solo nel caso in cui vengano collegati CLK al pin 12 e DIO al pin 11, in quanto è stata realizzata interagendo con le porte relative a quei pin.

```
63
64  /*
65  * Accende i segmenti sul display
66  * passati come parametro a partire
67  * dalla posizione iniziale
68  */
69  void display_7segments_set_segments(BYTE segments[], BYTE lenght, BYTE starter_position);
70
71  /*
72  * Accende i segmenti di una sola cifra
73  * sul display passati come parametro
74  * sulla posizione indicata
75  */
76  void display_7segments_set_segment(BYTE segment, BYTE position);
77
78  #endif
```

Figura 7.4: Righe 63-78 del file 7segments.h [SublimeText2]

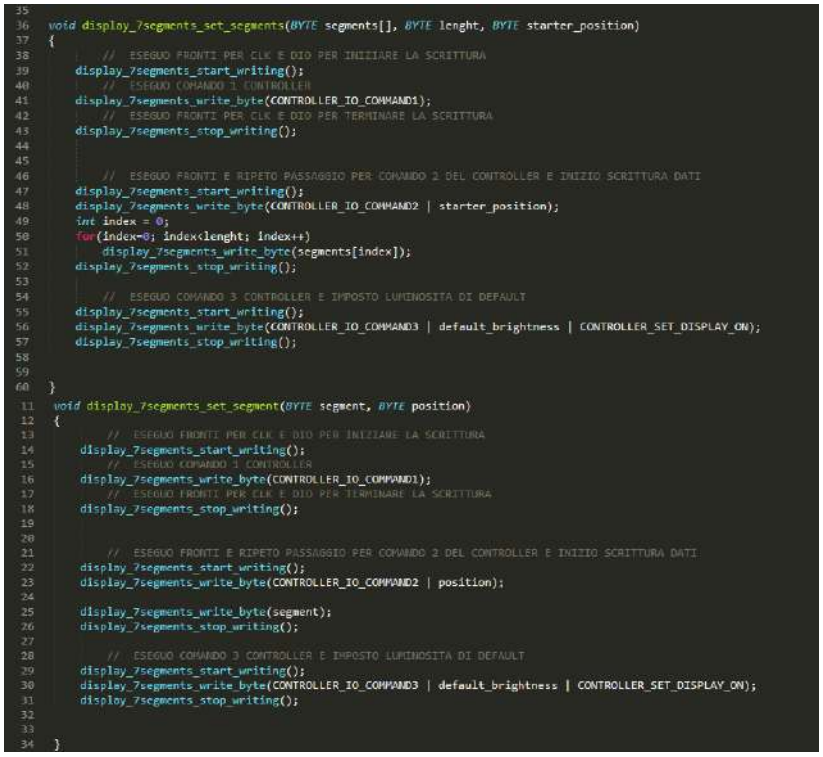
7.2.3 File 7segments.c

Nel file *7segments.c*, oltre a implementare i metodi presentati nel paragrafo precedente, vengono dichiarate e implementate altre funzioni ausiliarie statiche non visibili all'esterno della libreria, la cui firme sono le seguenti:

- static void display_7segments_start_writing(): genera fronti di salita/discesa per l'inizio della scrittura dei dati (figura 7.5).
- static void display_7segments_stop_writing(): genera fronti di salita/discesa per la fine della scrittura dei dati (figura 7.5).
- static void display_7segments_write_byte(BYTE data): scrive il BYTE passato come parametro un bit alla volta sul dispositivo.

```
61
62  void display_7segments_start_writing()
63  {
64          //  CLK -> HIGH
65      port_set_high_bits_port_B(0b00010000);
66          //  DIO -> HIGH
67      port_set_high_bits_port_B(0b00001000);
68          //  DIO -> LOW
69      port_set_low_bits_port_B(0b11110111);
70
71  }
72
73  void display_7segments_stop_writing()
74  {
75          //  CLK -> LOW
76      port_set_low_bits_port_B(0b11101111);
77          //  DIO -> LOW
78      port_set_low_bits_port_B(0b11110111);
79          //  CLK -> HIGH
80      port_set_high_bits_port_B(0b00010000);
81
82          //  DIO -> HIGH
83      port_set_high_bits_port_B(0b00001000);
84  }
```

Figura 7.5: Righe 61-84 del file 7segments.c [SublimeText2]



```
35
36  void display_7segments_set_segments(BYTE segments[], BYTE lenght, BYTE starter_position)
37  {
38          //  ESEGUO FRONTI PER CLK E DIO PER INIZIARE LA SCRITTURA
39      display_7segments_start_writing();
40          //  ESEGUO COMANDO 1 CONTROLLER
41      display_7segments_write_byte(CONTROLLER_IO_COMMAND1);
42          //  ESEGUO FRONTI PER CLK E DIO PER TERMINARE LA SCRITTURA
43      display_7segments_stop_writing();
44
45
46          //  ESEGUO FRONTI E RIPETO PASSAGGIO PER COMANDO 2 DEL CONTROLLER E INIZIO SCRITTURA DATI
47      display_7segments_start_writing();
48      display_7segments_write_byte(CONTROLLER_IO_COMMAND2 | starter_position);
49      int index = 0;
50      for(index=0; index<lenght; index++)
51          display_7segments_write_byte(segments[index]);
52      display_7segments_stop_writing();
53
54          //  ESEGUO COMANDO 3 CONTROLLER E IMPOSTO LUMINOSITA DI DEFAULT
55      display_7segments_start_writing();
56      display_7segments_write_byte(CONTROLLER_IO_COMMAND3 | default_brightness | CONTROLLER_SET_DISPLAY_ON);
57      display_7segments_stop_writing();
58
59
60  }
11  void display_7segments_set_segment(BYTE segment, BYTE position)
12  {
13          //  ESEGUO FRONTI PER CLK E DIO PER INIZIARE LA SCRITTURA
14      display_7segments_start_writing();
15          //  ESEGUO COMANDO 1 CONTROLLER
16      display_7segments_write_byte(CONTROLLER_IO_COMMAND1);
17          //  ESEGUO FRONTI PER CLK E DIO PER TERMINARE LA SCRITTURA
18      display_7segments_stop_writing();
19
20
21          //  ESEGUO FRONTI E RIPETO PASSAGGIO PER COMANDO 2 DEL CONTROLLER E INIZIO SCRITTURA DATI
22      display_7segments_start_writing();
23      display_7segments_write_byte(CONTROLLER_IO_COMMAND2 | position);
24
25      display_7segments_write_byte(segment);
26      display_7segments_stop_writing();
27
28          //  ESEGUO COMANDO 3 CONTROLLER E IMPOSTO LUMINOSITA DI DEFAULT
29      display_7segments_start_writing();
30      display_7segments_write_byte(CONTROLLER_IO_COMMAND3 | default_brightness | CONTROLLER_SET_DISPLAY_ON);
31      display_7segments_stop_writing();
32
33
34  }
```

Figura 7.6: Corpo delle funzioni display_7segments_set_segments() e display_7segments_set_segment(). [SublimeText2]

La figura 7.6 mostra il corpo delle due funzioni visibili all'esterno di questa libreria. Esse risultano apparentemente uguali, l'unica differenza è che nel caso della funzione display_7segments_set_segments() è presente un ciclo utile a scrivere tutte le cifre dell'array sul dispositivo.

Osservando il corpo delle due funzioni, si nota come ogni volta che viene inviato un comando, vengono generati fronti per l'inizio della scrittura prima, e per la fine dopo. Nella prima parte viene inviato il comando di inizio operazione e successivamente il comando di inizio sequenza dei dati, in OR con l'indirizzo della posizione della cifra su cui si sta tentando di scrivere. In riga 56 infine, vengono inviati il comando di fine interazione e le impostazioni di visualizzazione sul display.

In figura 7.7 invece, è mostrato il corpo della funzione display_7segments_write_byte(), che si occupa di scrivere il BYTE passato come parametro sul componente.

Nelle 89-112 è presente un ciclo simile a quello illustrato in figura 6.3, in cui viene mandato il dato un bit alla volta seguendo lo stesso procedimento.

Nella seconda parte invece, si attende l'ACK dal dispositivo, avendo settato in input il pin DIO e andando a controllare il valore di quest'ultimo dal registro PINB (unico caso

in cui viene utilizzato questo registro). Infine, viene reimpostato il pin DIO in output dopo aver assegnato il valore logico LOW a tutti i pin in gioco.

```
85
86  void display_7segments_write_byte(BYTE data)
87  {
88      int bit; BYTE ack;
89      for(bit=0; bit<8; bit++)
90      {
91          //  CLK -> LOW
92          port_set_low_bits_port_B(0b11101111);
93
94          // Discrimino se mandare 0 o 1
95          if(data & 0b00000001)
96          {
97              //  DIO -> HIGH
98              port_set_high_bits_port_B(0b00001000);
99          }
100         else
101         {
102             //  DIO -> LOW
103             port_set_low_bits_port_B(0b11110111);
104         }
105
106
107             //  CLK -> HIGH
108         port_set_high_bits_port_B(0b00010000);
109
110             //  Shifto al prossimo bit
111         data = data >> 1;
112     }
113         //  CLK -> LOW
114     port_set_low_bits_port_B(0b11101111);
115         //  DIO -> INPUT
116     port_setup_set_data_direction_input_bits_port_B(0b11110111);
117         //  DIO -> HIGH
118     port_set_high_bits_port_B(0b00001000);
119     ack = (((PINB & 0b00001000) > 0) ? 1 : 0);
120     if(ack)
121     {       //  DIO -> OUTPUT
122         port_setup_set_data_direction_output_bits_port_B(0b00001000);
123             //  DIO -> LOW
124         port_set_low_bits_port_B(0b11110111);
125     }
126
127         //  CLK -> HIGH
128     port_set_high_bits_port_B(0b00010000);
129         //  CLK -> LOW
130     port_set_low_bits_port_B(0b11101111);
131         //  DIO -> OUTPUT
132     port_setup_set_data_direction_output_bits_port_B(0b00001000);
133
134 }
```

Figura 7.7: Righe 85-134 del file 7segments.c [SublimeText2]

7.3 Esperimento: scrivere sul display

L'obiettivo di questo esperimento è quello di scrivere la parola "done" sul display. Per far ciò sono necessari, oltre alla scheda Arduino:

- Display a 7 segmenti a quattro cifre con microcontrollore TM1637
- Quattro fili per breadboard

Nel main implementato in quest'esempio verranno utilizzati i pin 12 e 11 come output, in quanto collegati rispettivamente al CLK E DIO. Bisogna inoltre connettere la VCC alla tensione massima di 5V e il pin GND a massa. Perciò occorrerà effettuare i collegamenti come in figura 7.8.

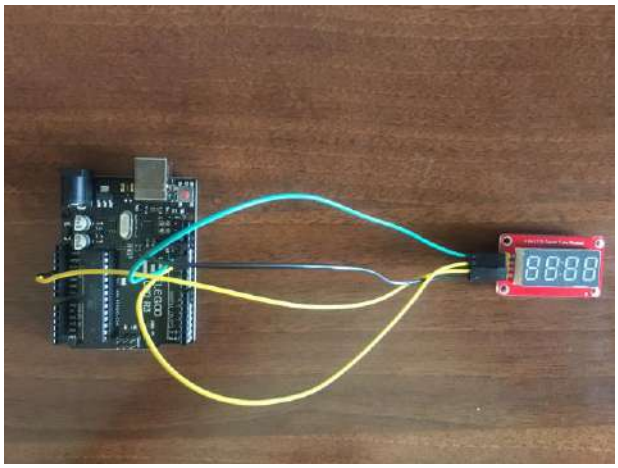


Figura 7.8: Assemblaggio dei collegamenti per scrivere sul display a 7 segmenti a 4 cifre con microcontrollore TM1637

Dopo aver assemblato le parti si può ora caricare il codice su Arduino come spiegato nel paragrafo 2.2.2.

I file da caricare saranno *common_commands.c*, file sorgente della libreria illustrata nel capitolo 4, *7segments.c*, file sorgente della libreria dell'esperimento attuale, e *7segments_main.c*, file che contiene un main d'esempio per il funzionamento di questo progetto.

Il contenuto del file *7segments_main.c* è mostrato in figura 7.9 e di seguito ne verrà spiegato il funzionamento.

Nella riga 1 viene inclusa la libreria realizzata per questo esperimento.

In riga 8 vengono settati i pin 11 e 12 in output, collegati rispettivamente a DIO e CLK. Quest'ultimi vengono inoltre posti a LOW come setup nelle righe 11 e 13.

Nelle righe 16-18 vengono accesi tutti i segmenti (per accendere i due punti bisogna porre a 1 il bit 7 della cifra in posizione 1) richiamando la funzione di libreria display_7segments_set_segments(), spiegata nei paragrafi precedenti.

Infine, in riga 19 inizia il ciclo infinito in cui viene scritto ad ogni giro la parola "done" sul display, allo stesso modo del passaggio precedente.

```c
#include "7segments.h"

        //  Output: CLK a pin 12 (PB4) e DIO a pin 11 (PB3)

int main()
{
        //  DIO -> OUTPUT
        //  CLK -> OUTPUT
    port_setup_set_data_direction_output_bits_port_B(0b00011000);
        //  CLK -> LOW
    port_set_low_bits_port_B(0b11101111);
        //  DIO -> LOW
    port_set_low_bits_port_B(0b11110111);

        //  Accendo i segmenti
    BYTE initial_segments[] = { 0b00111111, 0b10111111, 0b00111111, 0b00111111 };
    display_7segments_set_segments(initial_segments, 4, 0);
    _delay_ms(3000);
    while(1)
    {
        BYTE done[] = { 0b11011110, 0b00111111, 0b01010100, 0b01111001 };
        display_7segments_set_segments(done, 4, 0);
    }
}
```

Figura 7.9: Contenuto del file 7segments_main.c [SublimeText2]

La figura 7.10 mostra il risultato dell'esperimento dopo aver eseguito i passaggi appena spiegati e aver connesso l'Arduino all'alimentazione.

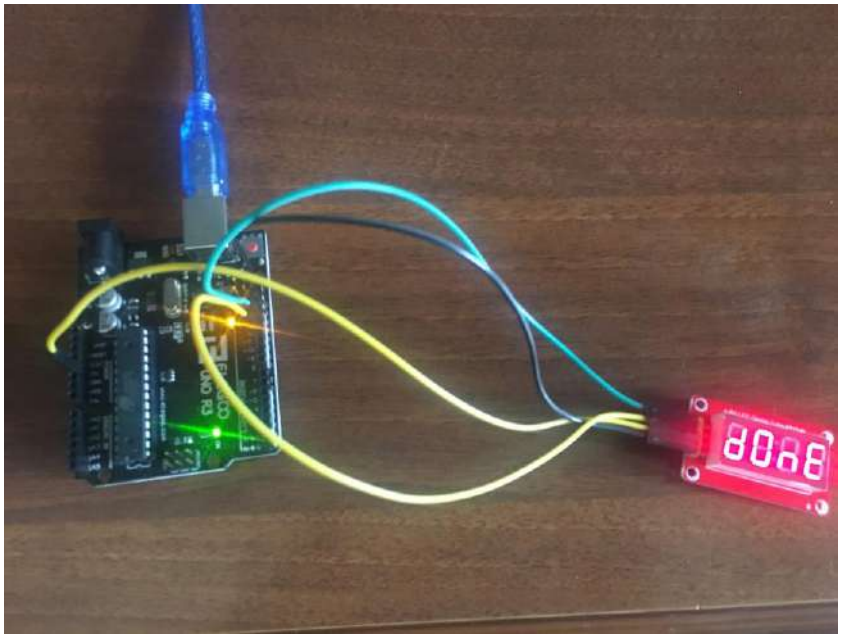


Figura 7.10: Esecuzione del programma che scrive la parola "done" sul display

CAPITOLO 8

Guida alla realizzazione degli esperimenti:

display LCD

Il progetto che verrà mostrato in questo capitolo è incentrato stavolta sulla comunicazione dei dati inviati dall'Arduino su di un display LCD. In questo caso però, tra il display e la scheda hardware non si interpone un microcontrollore, si andrà perciò a lavorare direttamente sui pin del display stesso.

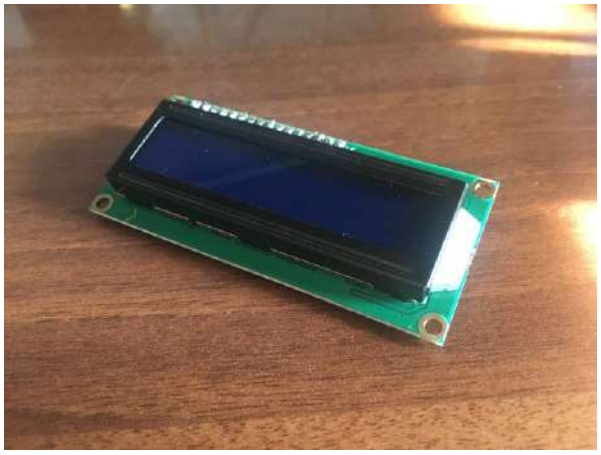


Figura 8.1: Display LCD

8.1 Breve descrizione del componente

Il display non monta un microcontrollore, integra bensì sedici pin:

- VSS: tensione minima, collegato a massa.
- VDD: tensione massima, collegato a 5V
- V0: pin che si occupa del contrasto dei caratteri, che varia in base alla tensione a cui è collegato. Perciò viene di solito connesso ad un potenziometro.
- RS: se posto al valore logico 1 permette di memorizzare i dati da visualizzare nel registro.
- R/W: permette di selezionare la modalità lettura nel caso in cui venisse assegnato il valore 1 a questo pin, scrittura in caso contrario. In questa tesi verrà utilizzato sempre in scrittura, verrà perciò connesso a massa.

- E: utile per abilitare la scrittura nei registri.
- D0-D7: pin utilizzati per lo scambio dei dati effettivi
- A (Anodo): pin a cui collegare una tensione positiva utile per attivare la retroilluminazione del display.
- K (Catodo): pin da collegare a massa per consentire la retroilluminazione.

Il display è formato da una matrice di caratteri 16x2 (due righe e sedici colonne), perciò ogni volta che si vuole scrivere su di esso, bisogna scegliere la riga su cui scrivere. Per discriminare la colonna invece, il display possiede un cursore che scorre in avanti ad ogni carattere.

Oltre ad inviare i dati da visualizzare, è possibile settare una serie di impostazioni sul display, come vedremo in seguito. Il display riceve i dati su di un fronte di discesa del pin E accompagnato dal valore LOW di RS, nel caso di scrittura di un comando, o dal valore HIGH, nel caso di scrittura di un carattere da visualizzare. Per scrivere un carattere basta assegnarlo alla variabile della porta su cui sono connessi i pin di dato, come vedremo in seguito.

8.2 Libreria LCD.h

Anche per questo componente si è scelto di realizzare una libreria per semplificarne l'utilizzo, nominata *LCD.h*. La figura 8.2 mostra il contenuto di questo file, che verrà spiegato nei prossimi paragrafi.

8.2.1 Macro

Dopo aver incluso la libreria *common_commands.h*, vengono definite le seguenti macro per rendere più intuitivo il settaggio delle impostazioni:

- LCD_DISPLAY_CHOSE_FIRST_ROW_COMMAND: indirizzo che, se inviato al dispositivo, permette di selezionare la prima riga per la scrittura dei caratteri.
- LCD_DISPLAY_CHOSE_SECOND_ROW_COMMAND: indirizzo che, se inviato al dispositivo, permette di selezionare la seconda riga per la scrittura dei caratteri.
- LCD_DISPLAY_CLEAR_DATA_COMMAND: indirizzo che, se inviato al dispositivo, cancella tutto ciò che è visualizzato sul display.

- LCD_DISPLAY_TURN_ON_COMMAND_WITH_CURSOR: indirizzo che, se inviato al dispositivo, accende il display visualizzando il cursore.
- LCD_DISPLAY_TURN_ON_COMMAND_WITHOUT_CURSOR: indirizzo che, se inviato al dispositivo, accende il display senza visualizzare il cursore.
- LCD_DISPLAY_INIZIALIZE_7x5_MATRIX_COMMAND: indirizzo che, se inviato al dispositivo, inizializza le matrici 7x5 che conterranno i caratteri. Quest'impostazione verrà sempre effettuata.

8.2.2 Funzioni

Nelle righe 27,34 e 42 vengono definite le seguenti funzioni visibili dall'esterno:

- void lcd_display_set_default_setting(): setta impostazioni di default LCD_DISPLAY_INIZIALIZE_7x5_MATRIX_COMMAND, LCD_DISPLAY_TURN_ON_COMMAND_WITHOUT_CURSOR, LCD_DISPLAY_CLEAR_DATA_COMMAND, LCD_DISPLAY_CHOSE_FIRST_COLUMN_COMMAND. Permette così di evitare l'utilizzo della lcd_display_set_command().
- void lcd_display_set_command(BYTE code): permette di impostare sul display un comando (impostazione) passato come parametro.
- void lcd_display_write_character(BYTE character): permette di scrivere sul display un carattere passato come parametro, seguendo le impostazioni settate con la lcd_display_set_command().

```
#ifndef _LCD_H_
#define _LCD_H_

#include "common_commands.h"

#define LCD_DISPLAY_CHOSE_FIRST_ROW_COMMAND                 0b10000000
#define LCD_DISPLAY_CHOSE_SECOND_ROW_COMMAND                0b11000000
#define LCD_DISPLAY_CLEAR_DATA_COMMAND                      0b00000001
#define LCD_DISPLAY_TURN_ON_COMMAND_WITH_CURSOR             0b00001110
#define LCD_DISPLAY_TURN_ON_COMMAND_WITHOUT_CURSOR          0b00001100
#define LCD_DISPLAY_INIZIALIZE_7x5_MATRIX_COMMAND           0b00111000
/*
*   Setta Impostazioni di default:
*   LCD_DISPLAY_INIZIALIZE_7x5_MATRIX_COMMAND
*   LCD_DISPLAY_TURN_ON_COMMAND_WITHOUT_CURSOR
*   LCD_DISPLAY_CLEAR_DATA_COMMAND
*   LCD_DISPLAY_CHOSE_FIRST_COLUMN_COMMAND
*   Permette di evitare l'utilizzo
*   della lcd_display_set_command()
*/
void lcd_display_set_default_setting();

/*
*   Permette di impostare sul display
*   un comando (impostazione) passato
*   come parametro
*/
void lcd_display_set_command(BYTE code);

/*
*   Permette di scrivere sul display
*   un carattere passato come parametro
*   seguendo le impostazioni settate
*   con la lcd_display_set_command().
*/
void lcd_display_write_character(BYTE character);

#endif
```

Figura 8.2: Contenuto del file LCD.h [SublimeText2]

8.2.3 LCD.c

Nel file *LCD.c*, oltre a implementare i metodi presentati nel paragrafo precedente, vengono dichiarate e implementate altre funzioni ausiliarie statiche non visibili all'esterno della libreria, la cui firme sono le seguenti:

- static void lcd_display_command_enable(): genera un fronte di discesa per il pin E dopo aver posto RS a valore LOW, in modo da scrivere un comando sul dispositivo (figura 8.3).
- static void lcd_display_data_enable(): (): genera un fronte di discesa per il pin E dopo aver posto RS a valore HIGH, in modo da scrivere un carattere sul dispositivo (figura 8.3).

```
void lcd_display_command_enable()
{
    //  RS -> LOW (Devo settare della impostazioni perciò non serve che il registro salvi i dati)
    port_set_low_bits_port_B(0b11111101);
    //  E -> HIGH (Preparo fronte di discesa)
    port_set_high_bits_port_B(0b00000001);
    _delay_ms(50);
    //  E -> LOW (Fronte di discesa)
    port_set_low_bits_port_B(0b11111110);
}

void lcd_display_data_enable()
{
    //  RS -> high (Seleziono il registro perché sto per mandare i dati)
    port_set_high_bits_port_B(0b00000010);
    //  E -> HIGH (Preparo fronte di discesa)
    port_set_high_bits_port_B(0b00000001);
    _delay_ms(50);
    //  E -> LOW (Fronte di discesa)
    port_set_low_bits_port_B(0b11111110);
}
```

Figura 8.3: Righe 33-53 del file LCD.h [SublimeText2]

In figura 8.4 invece è presente il corpo delle funzioni dichiarate nel file LCD.h, la cui firma è stata enunciata nel paragrafo 8.2.2.

La funzione lcd_display_set_default_setting() setta le impostazioni di default richiamando la funzione lcd_display_set_command().

La funzione lcd_display_set_command() invece, non fa altro che assegnare il BYTE del comando passato come parametro alla porta a cui sono collegati i pin di dato, richiamando infine la funzione lcd_display_command_enable() per permettere al dispositivo di campionare i dati ricevuti.

Analogo è il comportamento della funzione lcd_display_write_character(), con l'unica differenza che in questo caso viene chiamata la funzione lcd_display_data_enable(), per far si che il carattere venga campionato e scritto sul dispositivo.

```
9   void lcd_display_set_default_setting()
10  {
11      lcd_display_set_command(LCD_DISPLAY_INIZIALIZE_7x5_MATRIX_COMMAND);
12      lcd_display_set_command(LCD_DISPLAY_TURN_ON_COMMAND_WITHOUT_CURSOR);
13      lcd_display_set_command(LCD_DISPLAY_CLEAR_DATA_COMMAND);
14      lcd_display_set_command(LCD_DISPLAY_CHOSE_FIRST_ROW_COMMAND);
15  }
16
17  void lcd_display_set_command(BYTE code)
18  {
19          //  Setto i bit della porta D secondo il comando passato come parametro
20      port_set_all_bits_port_D(code);
21          //  Abilito il comando settato
22      lcd_display_command_enable();
23  }
24
25  void lcd_display_write_character(BYTE character)
26  {
27          //  Scrivo il carattere nella porta D
28      port_set_all_bits_port_D(character);
29          //  Abilito la visualizzazione dei dati settati
30      lcd_display_data_enable();
31  }
```

Figura 8.4: Righe 9-32 del file LCD.h [SublimeText2]

8.3 Esperimento: scrivere sul display

Questo esperimento deve far si che venga scritto la classica frase "Hello World!" sul display LCD. Per come è stato impostato il main, sono necessari, oltre alla scheda Arduino:

- Display LCD
- 23 fili per breadboard
- Un potenziometro
- Una resistenza

Per l'assemblaggio dei pin far riferimento alla figura 8.5.

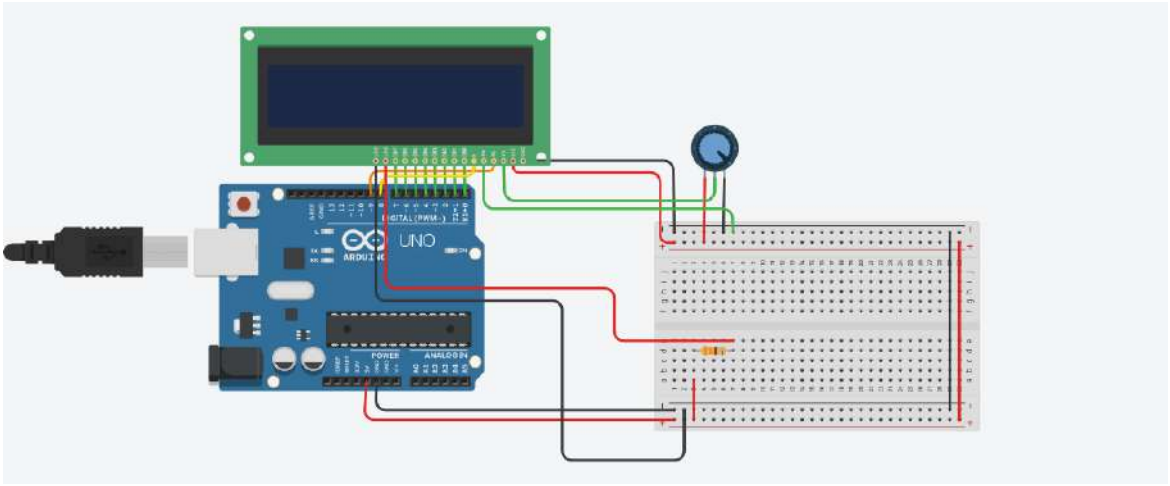


Figura 8.5: Assemblaggio per la realizzazione dell'esperimento di scrittura sul display LCD [6]

Il potenziometro è utile per regolare il contrasto del display, mentre il ruolo della resistenza è quello di evitare che possa arrivare troppa corrente all'anodo.

Dopo aver assemblato le parti si può ora caricare il codice su Arduino come spiegato nel paragrafo 2.2.2.

I file da caricare saranno *common_commands.c*, file sorgente della libreria illustrata nel capitolo 4, *LCD.c*, file sorgente della libreria dell'esperimento attuale, e *LCD_main.c*, file che contiene un main d'esempio per il funzionamento di questo progetto.

Il contenuto del file *LCD_main.c* è mostrato in figura 8.6 e di seguito ne verrà spiegato il funzionamento.

Nella riga 1 viene inclusa la libreria realizzata per questo esperimento.

Nelle righe 8,10 e 12 vengono posti tutti i pin di dato, E e RS, in output.

Dopo aver settato le impostazioni di default, viene scritta la frase sul display attraverso la funzione lcd_display_write_character(), per poi entrare in un ciclo infinito in cui non viene eseguita alcuna istruzione.

```
1  #include "LCD.h"
2
3      //  Suppongo D0-D7 collegati alla PORTA D, RS a PIN 9 (PORTA B BIT 1), E a PIN 8 (PORTA B BIT 0)
4
5  int main(void)
6  {
7          //  PORTD -> OUTPUT
8      port_set_all_bits_data_direction_port_D(0b11111111);
9          //  RS -> OUTPUT
10     port_setup_set_data_direction_output_bits_port_B(0b00000010);
11         //  E  -> OUTPUT
12     port_setup_set_data_direction_output_bits_port_B(0b00000001);
13
14         //  SETTO IMPOSTAZIONI DI DEFAULT SUL DISPLAY
15     lcd_display_set_default_setting();
16
17         //  SCRIVO SU DISPLAY
18     BYTE frase[] = { 'H', 'E', 'L', 'L', '0', ' ', 'W', 'O', 'R', 'L', 'D', '!' };
19     int index;
20     for(index=0; index<sizeof(frase); index++)
21         lcd_display_write_character(frase[index]);
22     while(1);
23
24 }
```

Figura 8.6: Contenuto del file LCD_main.c [SublimeText2]

La figura 8.7 mostra il risultato dell'esperimento dopo aver eseguito i passaggi appena spiegati e aver connesso l'Arduino all'alimentazione.

Figura 8.7: Esecuzione del programma di scrittura della frase "Hello World!"
sul display LCD

La libreria è funzionante solo nel caso in cui vengano collegati i fili come indicato in figura 8.5, in quanto è stata realizzata interagendo con le porte relative a quei pin.

CAPITOLO 9

Guida alla realizzazione degli esperimenti: gestione degli interrupt

In questo capitolo verrà illustrato il modo in cui si gestiscono gli interrupt in Arduino, utilizzando anche stavolta delle librerie apposite, che saranno fondamentali per la realizzazione degli ultimi esperimenti esplicati in questa tesi.

Nella scheda Arduino Uno sono presenti due tipologie diverse di interrupt:

- External Interrupt: sono solamente due, INT0 e INT1, mappati rispettivamente sui pin 2 e 3.
- Pin Change Interrupt: sono 24 (denominati PCINTX, dove X sta per il numero dell'interrupt) e sono distribuiti fra i vari pin della scheda.

La differenza sostanziale sta nel fatto che i primi sono interrupt interpretati direttamente dall'hardware e perciò risultano molto veloci, mentre nel secondo caso sono collegati alle porte e perciò spetta al programmatore verificare l'eventuale variazione del loro stato.

Negli esperimenti illustrati in questa tesi, si utilizzeranno solamente Pin Change Interrupt.

9.1 Registri per la gestione dei Pin Change Interrupt

Per abilitare l'interrupt di un determinato pin, in Arduino sono presenti tre registri (PCMSK0, PCMSK1, PCMSK2), chiamati *maschere*, a cui ad ogni bit corrisponde un interrupt di un pin diverso, com'è visibile nelle figure 9.2, 9.3, 9.4.

Ogni maschera è inizialmente disattivata, quindi all'accensione della scheda gli interrupt di tutti pin non sono considerati.

Il registro che si occupa dell'attivazione delle maschere è il PCICR, registro di controllo dei pin change interrupt, in cui, come si vede in figura 9.1, solamente i bit 0,1 e 2 hanno un compito ben specifico, quello di abilitare, ognuno, una determinata maschera, ad esempio: nel bit 0 del registro PCICR è presente il PCIE0 (Pin Change Interrupt Enable 0), che se settato a 1, abilita la PCMSK0.

Questo procedimento potrà sembrare poco chiaro, perciò di seguito verrà spiegato un esempio più completo.

Supponiamo di voler attivare l'interrupt del pin 12.
Dalla figura 3.3 notiamo che al pin 12 corrisponde il PCINT4.
Il PCINT4 corrisponde invece al bit 4 della PCMSK0, perciò lo settiamo al valore logico 1, lasciando tutti gli altri a 0.
A questo punto bisogna attivare la maschera 0, ponendo quindi a 1 il bit 0 del PCICR, attivando così l'interrupt relativo al pin 12 [3].

17.2.4. Pin Change Interrupt Control Register

Name: PCICR
Offset: 0x68
Reset: 0x00
Property: -

Bit	7	6	5	4	3	2	1	0
						PCIE2	PCIE1	PCIE0
Access						R/W	R/W	R/W
Reset						0	0	0

Figura 9.1: Configurazione dei bit del registro PCICR [3]

17.2.6. Pin Change Mask Register 2

Name: PCMSK2
Offset: 0x6D
Reset: 0x00
Property: -

Bit	7	6	5	4	3	2	1	0
	PCINT23	PCINT22	PCINT21	PCINT20	PCINT19	PCINT18	PCINT17	PCINT16
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	0	0	0

Figura 9.2: Configurazione dei bit del registro PCMSK2 [3]

17.2.7. Pin Change Mask Register 1

Name: PCMSK1
Offset: 0x6C
Reset: 0x00
Property: -

Bit	7	6	5	4	3	2	1	0
		PCINT14	PCINT13	PCINT12	PCINT11	PCINT10	PCINT9	PCINT8
Access		R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset		0	0	0	0	0	0	0

Figura 9.3: Configurazione dei bit del registro PCMSK1 [3]

17.2.8. Pin Change Mask Register 0

Name: PCMSK0
Offset: 0x6B
Reset: 0x00
Property: -

Bit	7	6	5	4	3	2	1	0
	PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	0	0	0

Figura 9.4: Configurazione dei bit del registro PCMSK0 [3]

9.2 Libreria common_commands.h

Si è parlato di questa libreria nel capitolo 4, tralasciando però tutta la parte di gestione degli interrupt, che verrà invece illustrata in questo paragrafo.

9.2.1 Macro

Nella riga 21 del file common_commands.h, è presente una macro in cui la macro ISR(vector) viene ridefinita in interrupt_handler(vector). ISR sta per “Interrupt Service Routine” dichiarata nella libreria avr/interrupt.h nel seguente modo:

- # define __INTR_ATTRS used, externally_visible
- #define ISR(vector, ...) extern "C" void vector (void) __attribute__ ((signal,__INTR_ATTRS)) __VA_ARGS__; \ void vector (void) [10]

Per definire in C un handler di un certo interrupt infatti, bisognerebbe dichiararlo come:

- extern "C" void PCINT0_vect (void) __attribute__ ((signal)); void PCINT0_vector (void) {...}

• PCINT0_vect è il vettore che indica al linker a quale interrupt associare l'handler e che verrà poi spiegato più nel dettaglio nel paragrafo successivo. [nel paragrafo seguente spiego a quale pin si riferisce ogni vettore]

• 'extern "C"' è una direttiva utilizzata per indicare al compilatore di non modificare il nome simbolico associato alla funzione dopo la compilazione, evitando così il “mangling”. Senza l'utilizzo di questa dichiarazione, è possibile che il compilatore assegni un valore simbolico diverso alla funzione.

• '__attribute__ ((signal))' indica al compilatore le caratteristiche da associare alla funzione che affianca e in questo caso signal indica che la funzione è un handler di segnale. [9]

Il precompilatore quindi, ogni volta che incontra l'espressione ISR(PCINT0_vect) sostituisce a essa la dichiarazione appena spiegata. Da notare che il nome del vettore viene sostituito al nome della funzione "vector", perciò la macro funge da puntatore a funzione.

In C un puntatore a funzione è una variabile a cui possono essere assegnati gli indirizzi in cui risiedono una o più funzioni e tramite cui possono essere richiamate le funzioni stesse.

Confrontiamo la dichiarazione di una funzione:

- int somma (int x, int y);

con la dichiarazione di un puntatore a funzione:

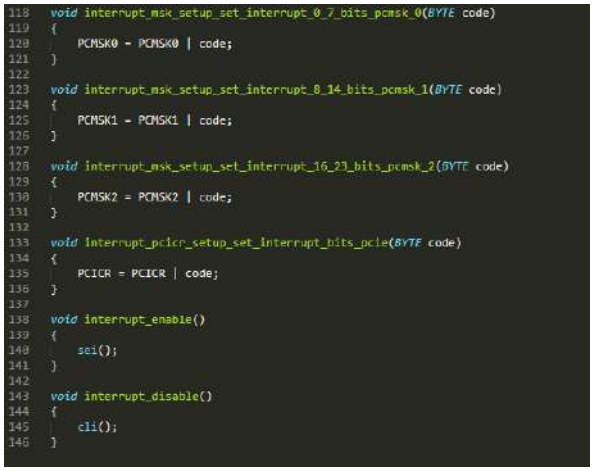
- int (* puntatore_funzione) (x, y)

Si può eseguire quindi la funzione somma attraverso il puntatore nel seguente modo:
puntatore_funzione= somma;
int result = puntatore_funzione(x,y);
Nel nostro caso, richiamando la macro ISR(PCINT0_vect), la funzione "vector" punterà alla funzione void PCINT0_vect (void){} che diventerà un handler attraverso le direttive spiegate in precedenza e che il linker assocerà agli handler di gestione degli interrupt dei relativi pin.
La macro ISR(vector) è stata ridefinita in "interrupt_handler(vector)" nella libreria common_commands.h al fine di rendere più leggibile il codice.

9.2.2 Funzioni

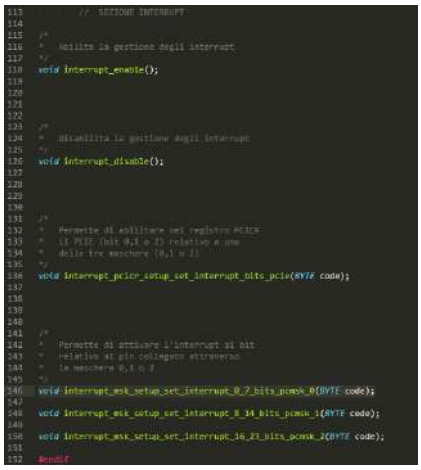
Per rendere più semplice l'utilizzo degli interrupt, sono state definite nella libreria le seguenti funzioni (figura 9.5):

- void interrupt_enable(): abilita la gestione degli interrupt. In figura 9.6 ne è visibile l'implementazione, in cui viene richiamata la funzione sei(), dichiarata nella libreria *avr/interrupt.h* e si occupa di attivare la gestione degli interrupt.
- void interrupt_disable(): disabilita la gestione degli interrupt. In figura 9.6 ne è visibile l'implementazione, in cui viene richiamata la funzione cli(), dichiarata nella libreria *avr/interrupt.h* e si occupa di disattivare la gestione degli interrupt.
- void interrupt_pcicr_setup_set_interrupt_bits_pcie(BYTE code): permette di attivare la maschera relativa al pin in cui si vuole attivare l'interrupt, passandogli un BYTE come parametro, con il bit relativo alla maschera settato a 1. In figura 9.6 ne è visibile l'implementazione.
- void interrupt_msk_setup_set_interrupt_X_Y_bits_pcmsk_Z(BYTE code): permette di attivare gli interrupt che vanno dal PCINTX al PCINTY della maschera Z (PCMSKZ), passandogli come parametro un BYTE, con i relativi bit settati a 1. In figura 9.6 ne è visibile l'implementazione.



```
118 void interrupt_msk_setup_set_interrupt_0_7_bits_pcmsk_0(BYTE code)
119 {
120     PCMSK0 = PCMSK0 | code;
121 }
122
123 void interrupt_msk_setup_set_interrupt_8_14_bits_pcmsk_1(BYTE code)
124 {
125     PCMSK1 = PCMSK1 | code;
126 }
127
128 void interrupt_msk_setup_set_interrupt_16_23_bits_pcmsk_2(BYTE code)
129 {
130     PCMSK2 = PCMSK2 | code;
131 }
132
133 void interrupt_pcicr_setup_set_interrupt_bits_pcie(BYTE code)
134 {
135     PCICR = PCICR | code;
136 }
137
138 void interrupt_enable()
139 {
140     sei();
141 }
142
143 void interrupt_disable()
144 {
145     cli();
146 }
```

Figura 9.6: Righe 118-146 del file common_commands.c [SublimeText2]



```
113         // SEZIONE INTERRUPT
114
115 /*
116 *   Abilita la gestione degli interrupt
117 */
118 void interrupt_enable();
119
120
121
122
123 /*
124 *   Disabilita la gestione degli interrupt
125 */
126 void interrupt_disable();
127
128
129
130
131 /*
132 *   Permette di abilitare nel registro PCICR
133 *   il PCIE (bit 0,1 o 2) relativo a una
134 *   delle tre maschere (0,1 o 2)
135 */
136 void interrupt_pcicr_setup_set_interrupt_bits_pcie(BYTE code);
137
138
139
140
141 /*
142 *   Permette di attivare l'interrupt ai bit
143 *   relativi ai pin collegati attraverso
144 *   la maschera 0,1 o 2
145 */
146 void interrupt_msk_setup_set_interrupt_0_7_bits_pcmsk_0(BYTE code);
147
148 void interrupt_msk_setup_set_interrupt_8_14_bits_pcmsk_1(BYTE code);
149
150 void interrupt_msk_setup_set_interrupt_16_23_bits_pcmsk_2(BYTE code);
151
152 #endif
```

Figura 9.5: Righe 113-152 del file common_commands.h [SublimeText2]

9.3 Esperimenti basati sulla gestione degli interrupt

Per quanto riguarda i progetti realizzati con dispositivi che comunicano con gli interrupt, non è risultato necessario la dichiarazione di ulteriori variabili o metodi, perciò la libreria definita per questo progetto, *interrupt_lib.h*, oltre a richiamare la libreria *common_commands.h*, dichiara solamente la funzione:

- interrupt_handler(PCINT0_vect): handler dell'interrupt. Scatta quando uno degli interrupt del vettore passato come parametro subisce un fronte di salita (se pull down) o discesa (se pull-up). Questa funzione viene implementata nel main e non nel file sorgente di libreria.

  PCINTX_vect è il parametro della funzione, esso fa si che l'handler scatti alla ricezione degli interrupt dei pin che vanno [11]:

  - Dal pin digitale 8 al pin digitale 13 se l'argomento è PCINT0_vect
  - Dal pin analogico 0 al pin analogico 5 se l'argomento è PCINT1_vect
  - Dal pin digitale 0 al pin digitale 7 se l'argomento è PCINT2_vect

  Questa libreria è funzionante solo per i pin 8-13, in quanto è stata dichiarata con argomento PCINT0_vect. Nel caso in cui si dovessero usare pin diversi, bisognerebbe cambiare l'argomento con quello corretto.

9.3.1 Esperimento: accensione e spegnimento di un led al segnale mandato da un sensore PIR

Il sensore PIR (Passive InfraRed) è un sensore di movimento, che manda un segnale ogni volta che ne rileva uno. Possiede tre pin: OUT, pin da cui il sensore manda il segnale, VCC, che va collegato alla tensione massima e GND, che andrà invece connesso a massa.

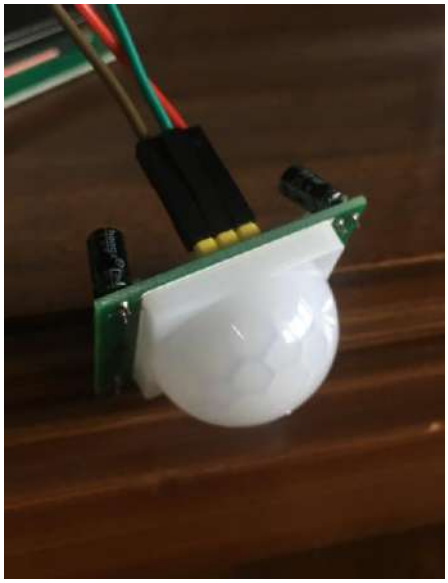


Figura 9.7: Sensore PIR

In questo esperimento si vuole accendere e spegnere un led ogni volta che un movimento è stato rilevato dal sensore. Per ciò sono necessari, oltre alla scheda Arduino:

- Una breadboard
- Un led
- Una resistenza
- Cinque fili per breadboard
- Un sensore PIR

Nel main implementato in quest'esempio verrà utilizzato il pin 7 come output., in quanto connesso al led rosso, e il pin 12 come interrupt, in quanto connesso al pin OUT del sensore.

Perciò in questo caso occorrerà effettuare i collegamenti come in figura 9.8.

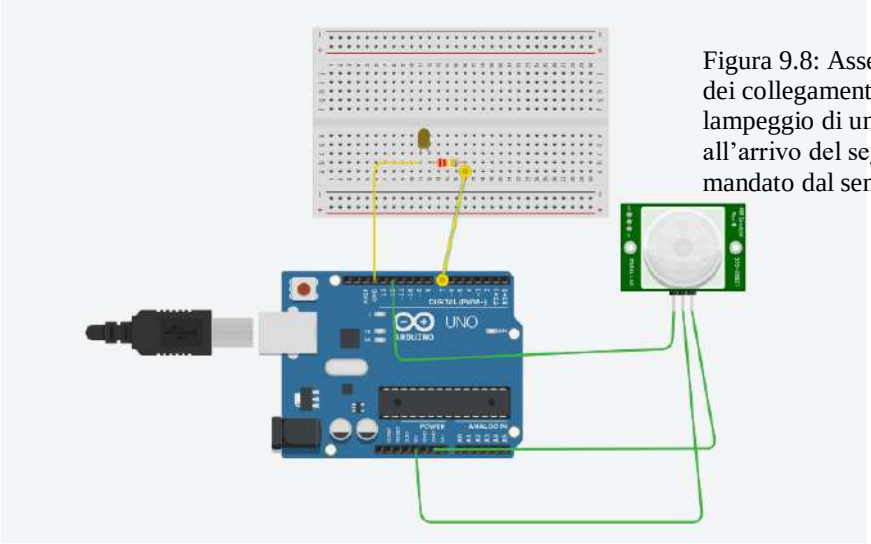


Figura 9.8: Assemblaggio dei collegamenti per il lampeggio di un singolo led all'arrivo del segnale mandato dal sensore [6]

Dopo aver assemblato le parti si può ora caricare il codice su Arduino come spiegato nel paragrafo 2.2.2.

I file da caricare saranno *common_commands.c*, file sorgente della libreria illustrata nel capitolo 4, *interrupt_lib.c*, file sorgente della libreria dell'esperimento attuale, e *PIR_interrupt_main.c*, file che contiene un main d'esempio per il funzionamento di questo progetto.

Il contenuto del file PIR_interrupt_*main.c* è mostrato in figura 9.9 e di seguito ne verrà spiegato il funzionamento.

Nella riga 1 viene inclusa la libreria realizzata per questo esperimento.

In riga 18 vengono disabilitati gli interrupt.

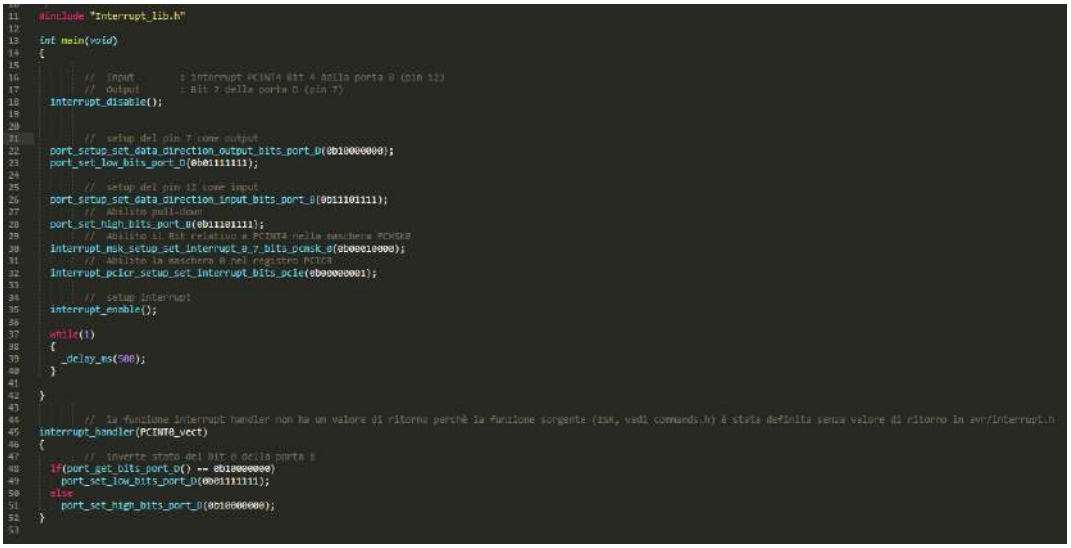
Nelle righe 22 e 23 viene impostato il pin 7 come output e poi posto al valore logico LOW.

Nelle righe 22-32 viene impostato l'interrupt del pin 12, seguendo il procedimento spiegato nel paragrafo 9.1.

In riga 35 vengono abilitati gli interrupt.

In riga 37 inizia il ciclo che non smette mai di eseguire, mettendosi così in attesa infinita di segnali.

Alla ricezione dell'interrupt viene chiamato l'handler in riga 45, che verifica se il led sia acceso per spegnerlo, o invece se sia spento per poi accenderlo.



```
#include "Interrupt_lib.h"

int main(void)
{
        //  Input       : Interrupt PCINT4 Bit 4 della porta B (Pin 12)
        //  Output      : Bit 7 della porta D (pin 7)
  interrupt_disable();

        //  setup del pin 7 come output
  port_setup_set_data_direction_output_bits_port_D(0b10000000);
  port_set_low_bits_port_D(0b01111111);

        //  setup del pin 12 come input
  port_setup_set_data_direction_input_bits_port_B(0b11101111);
        //  abilito pull-down
  port_set_high_bits_port_B(0b11101111);
        //  Abilito il Bit relativo a PCINT4 nella maschera PCMSK0
  interrupt_msk_setup_set_interrupt_0_7_bits_pcmsk_0(0b00010000);
        //  Abilito la maschera 0 nel registro PCICR
  interrupt_pcicr_setup_set_interrupt_bits_pcie(0b00000001);

        //  setup interrupt
  interrupt_enable();

  while(1)
  {
    _delay_ms(500);
  }

}

        //  la funzione interrupt handler non ha un valore di ritorno perché la funzione sorgente (isr, vedi commands.h) è stata definita senza valore di ritorno in avr/interrupt.h
interrupt_handler(PCINT0_vect)
{
        //  inverte stato del Bit 0 della porta B
  if(port_get_bits_port_D() == 0b10000000)
    port_set_low_bits_port_D(0b01111111);
  else
    port_set_high_bits_port_D(0b10000000);
}
```

Figura 9.9: Contenuto del file PIR_interrupt_*main*.c
[SublimeText2]

La figura 9.10 mostra il risultato dell'esperimento dopo aver eseguito i passaggi appena spiegati e aver connesso l'Arduino all'alimentazione.

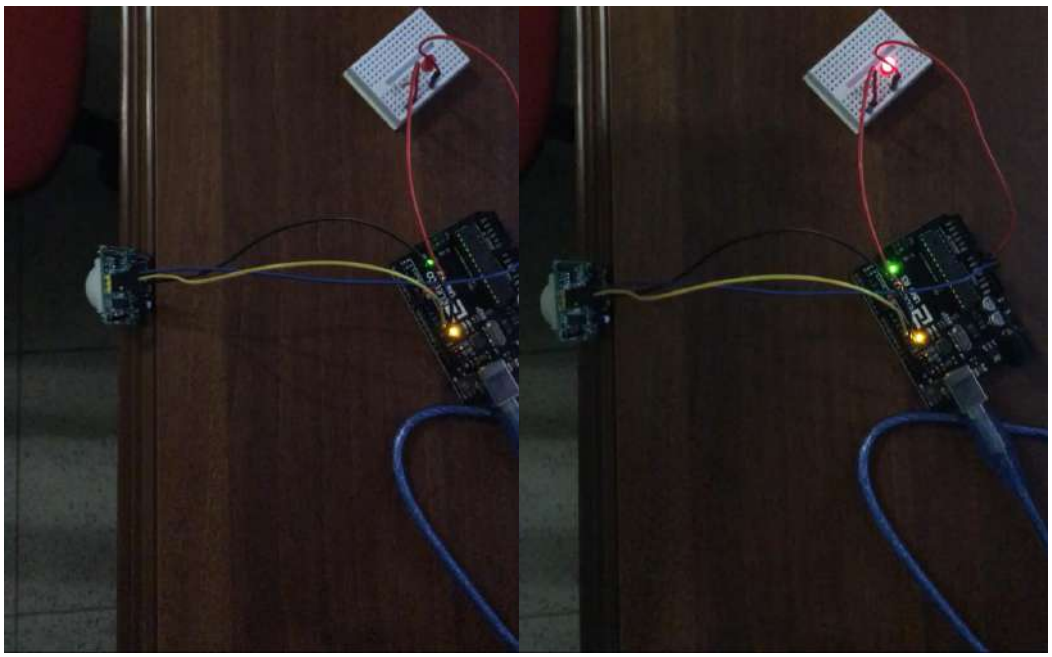


Figura 9.10: Esecuzione finale del programma di accensione e spegnimento del led al rilevamento di un movimento

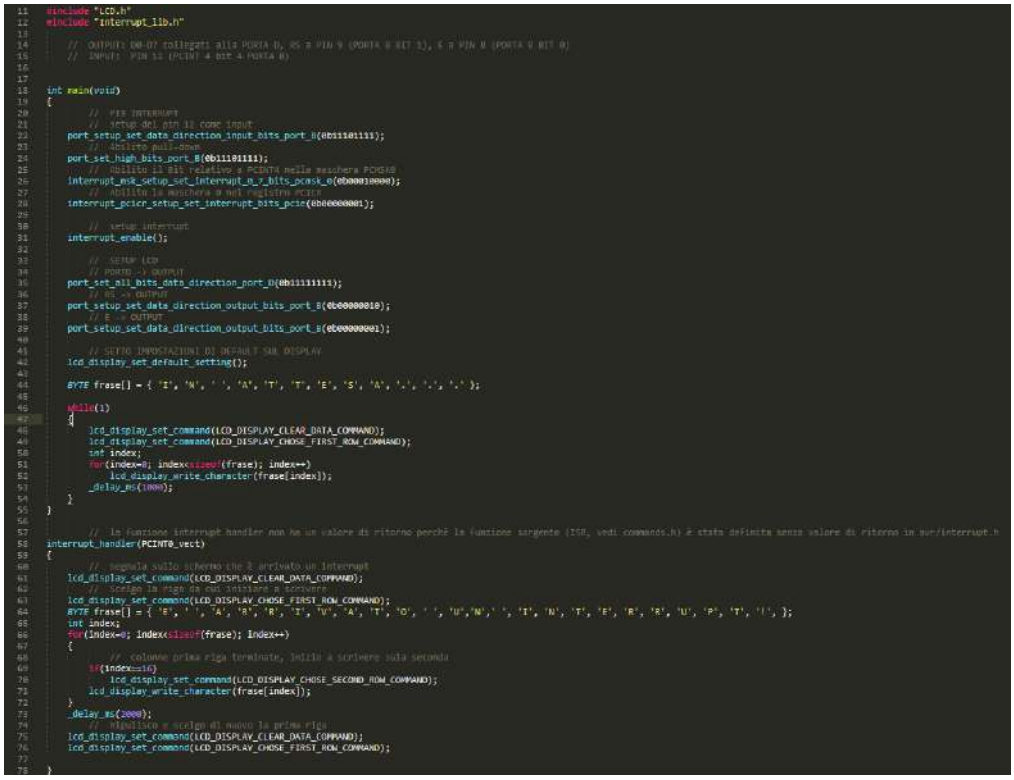
9.3.2 Esperimento: visualizzazione della notifica di ricezione di un interrupt su un display LCD

In questo esperimento si vuole visualizzare su un display LCD la notifica di ricezione di un interrupt, ogni volta che viene rilevato un movimento da un sensore PIR.
Per fare ciò sono necessari, oltre alla scheda Arduino:

- Una breadboard
- Un potenziometro
- Un sensore PIR
- Un display LCD
- 26 fili per breadboard

In questo caso bisogna effettuare i collegamenti come fatto negli esperimenti di paragrafo 8.3 e 9.3.1.
Dopo aver assemblato le parti si può ora caricare il codice su Arduino come spiegato nel paragrafo 2.2.2.
I file da caricare saranno *common_commands.c*, file sorgente della libreria illustrata nel capitolo 4, *interrupt_lib.c*, file sorgente della libreria dell'esperimento attuale, LCD.c, file sorgente della libreria del display LCD, e *LCD_PIR_Interrupt_main.c*, file che contiene un main d'esempio per il funzionamento di questo progetto.
Il contenuto del file *LCD_PIR_Interrupt_main.c* è mostrato in figura 9.11 e di seguito ne verrà spiegato il funzionamento.

In riga 11 e 12 vengono incluse le librerie necessarie per l'esecuzione di questo esperimento: le librerie realizzate per il display LCD e per la definizione dell'handler degli interrupt.
Dalla riga 22 alla riga 31 vengono eseguite le impostazioni di settaggio dell'interrupt, come spiegato nel paragrafo precedente.
Dalla riga 35 alla riga 43 vengono eseguite le impostazioni di settaggio del display, come spiegato nel paragrafo 8.3.
In riga 46 inizia il ciclo infinito in cui viene scritto ogni secondo la frase "IN ATTESA…".
Alla ricezione dell'interruzione, viene eseguita la funzione handler in riga 58 in cui viene scritta sul display la frase "E' ARRIVATO UN INTERRUPT!"



```c
#include "LCD.h"
#include "interrupt_lib.h"

    // OUTPUT: D0-D7 collegati alla PORTA B, RS a PIN 9 (PORTA B BIT 1), E a PIN 8 (PORTA B BIT 0)
    // INPUT:  PIN 11 (PCINT 4 BIT 4 PORTA B)


int main(void)
{
        // PIR INTERRUPT
        //  setup del pin 11 come input
    port_setup_set_data_direction_input_bits_port_B(0b11101111);
        //  Abilito pull-down
    port_set_high_bits_port_B(0b11101111);
        //  Abilito il bit relativo a PCINT4 nella maschera PCMSK0
    interrupt_msk_setup_set_interrupt_m_y_bits_pcmsk_m(0b00010000);
        //  Abilito la maschera a nel registro PCICR
    interrupt_pcicr_setup_set_interrupt_bits_pcie(0b00000001);

        //  setup interrupt
    interrupt_enable();

        //  SETUP LCD
        // PORTB -> OUTPUT
    port_set_all_bits_data_direction_port_D(0b11111111);
        // RS -> OUTPUT
    port_setup_set_data_direction_output_bits_port_B(0b00000010);
        // E -> OUTPUT
    port_setup_set_data_direction_output_bits_port_B(0b00000001);

        // SETTO IMPOSTAZIONI DI DEFAULT SUL DISPLAY
    lcd_display_set_default_setting();

    BYTE frase[] = { 'I', 'N', ' ', 'A', 'T', 'T', 'E', 'S', 'A', '.', '.', '.' };

    while(1)
    {
        lcd_display_set_command(LCD_DISPLAY_CLEAR_DATA_COMMAND);
        lcd_display_set_command(LCD_DISPLAY_CHOSE_FIRST_ROW_COMMAND);
        int index;
        for(index=0; index<sizeof(frase); index++)
            lcd_display_write_character(frase[index]);
        _delay_ms(1000);
    }
}

        //  la funzione interrupt_handler non ha un valore di ritorno perché la funzione sorgente (ISR, vedi commands.h) è stata definita senza valore di ritorno in avr/interrupt.h
interrupt_handler(PCINT0_vect)
{
    //  segnala sullo schermo che è arrivato un interrupt
    lcd_display_set_command(LCD_DISPLAY_CLEAR_DATA_COMMAND);
    //  scrivo la riga da cui iniziare a scrivere
    lcd_display_set_command(LCD_DISPLAY_CHOSE_FIRST_ROW_COMMAND);
    BYTE frase[] = { 'E', ' ', 'A', 'R', 'R', 'I', 'V', 'A', 'T', 'O', ' ', 'U','N',' ', 'I', 'N', 'T', 'E', 'R', 'R', 'U', 'P', 'T', '!', };
    int index;
    for(index=0; index<sizeof(frase); index++)
    {
        //  colonne prima riga terminate, inizio a scrivere sulla seconda
        if(index==16)
            lcd_display_set_command(LCD_DISPLAY_CHOSE_SECOND_ROW_COMMAND);
        lcd_display_write_character(frase[index]);
    }
    _delay_ms(2000);
    //  ripulisco e scrivo di nuovo la prima riga
    lcd_display_set_command(LCD_DISPLAY_CLEAR_DATA_COMMAND);
    lcd_display_set_command(LCD_DISPLAY_CHOSE_FIRST_ROW_COMMAND);
}
```

Figura 9.11: Contenuto del file LCD_PIR_Interrupt_main.c [SublimeText2]

La figura 9.12 mostra il risultato dell'esperimento dopo aver eseguito i passaggi appena spiegati e aver connesso l'Arduino all'alimentazione.

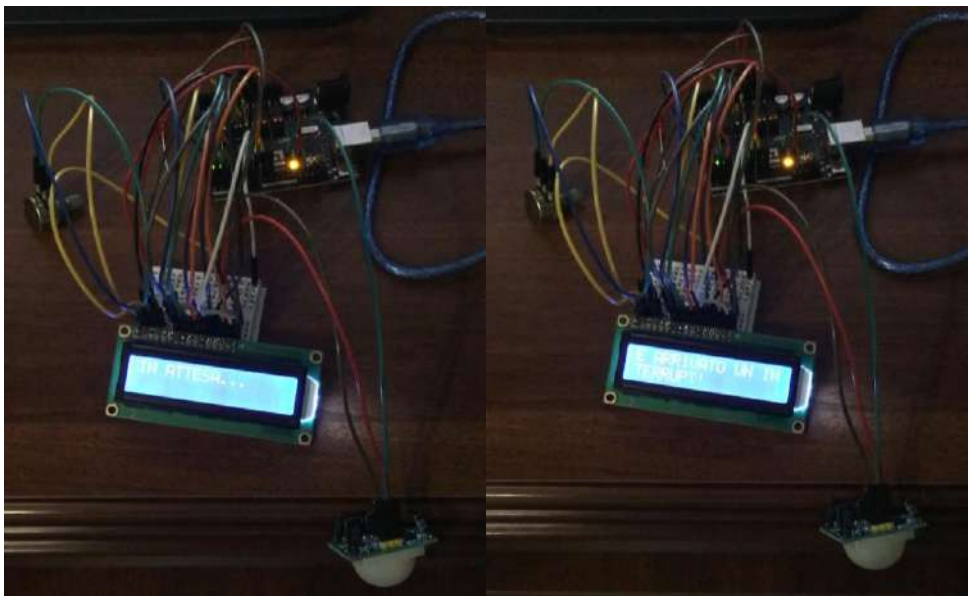


Figura 9.12: Esecuzione finale del programma di visualizzazione di notifica dell'interrupt su un display LCD [SublimeText2]

9.3.3 Esperimento: contatore degli interrupt ricevuti da un encoder a disco.

Un encoder è un particolare dispositivo formato da un diodo, che emana energia luminosa, e un fotosensore di ricezione. Facendo ruotare un disco forato in mezzo alle due componenti, l'encoder manda un segnale ogni volta che la luce non viene rilevata dal fotosensore (figura 9.13). Anche questo dispositivo possiede tre pin: OUT, da cui il sensore manda il segnale, VCC, che va collegato alla tensione massima e GND, che andrà invece connesso a massa.

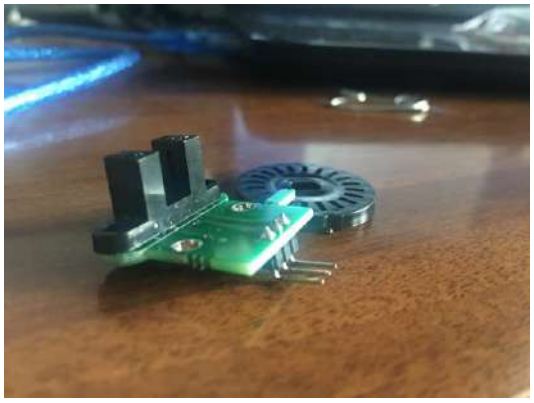


Figura 9.13: Encoder a disco

In questo esperimento si vuole contare il numero degli interrupt ricevuti da un encoder attraverso un display a 7 segmenti. Per fare ciò sono necessari, oltre alla scheda Arduino:

- Una breadboard
- Un encoder a disco
- Un display a 7 segmenti
- 10 fili per breadboard

In questo caso bisogna effettuare i collegamenti come fatto negli esperimenti di paragrafo 7.3 e 9.3.1, in quanto l'encoder possiede pin analoghi ai pin del sensore PIR.

Dopo aver assemblato le parti si può ora caricare il codice su Arduino come spiegato nel paragrafo 2.2.2.

I file da caricare saranno *common_commands.c*, file sorgente della libreria illustrata nel capitolo 4, *interrupt_lib.c*, file sorgente della libreria dell'esperimento attuale, *7segments.c*, file sorgente della libreria del display a 7 segmenti, e

7segment_Encoder_Interrupt_main.c, file che contiene un main d'esempio per il funzionamento di questo progetto.
Il contenuto del file 7segment_Encoder_Interrupt_*main.c* è mostrato in figura 9.14 e di seguito ne verrà spiegato il funzionamento.

Oltre alle librerie necessarie per i dispositivi collegati in questo esperimento, 7segments.h e Interrupt_lib.h, vengono incluse le librerie stdio.h, stdlib.h e string.h utili per la realizzazione dell'algoritmo del contatore.
In riga 18 viene dichiarata la variabile globale counter, un array di tanti elementi quanti sono le cifre del display.
Nel main di questo file sono presenti istruzioni equivalenti a quelle utilizzate negli esperimenti precedenti, non verranno perciò nuovamente spiegate.
In riga 56 è presente il corpo dell'interrupt_handler che si occupa di gestire il conteggio degli interrupt attraverso la variabile globale "counter", dichiarata precedentemente.
Questo algoritmo utilizza un array di char invece che un intero, perché rende più semplice la scrittura del valore corrente sul display.
Da notare l'utilizzo della variabile digitToSegment, la cui utilità è stata illustrata nel paragrafo 7.2.2.
Un ultimo aspetto particolare è l'istruzione "(int) counter[DISPLAY_3_ADDR] - '0'", utile per effettuare il cast da char a int.

La figura 9.15 mostra il risultato dell'esperimento dopo aver eseguito i passaggi appena spiegati e aver connesso l'Arduino all'alimentazione.

```c
#include "Interrupt_lib.h"
#include "7segments.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

        //  Contatore Interrupt Ricevute
char counter[] = { '0', '0', '0', '0'};

int main(void)
{
        //  Input       : interrupt PCINT2 Bit 2 della porta B (pin 10)
        //  Output      : CLK a pin 12 (PB4) e DIO a pin 11 (PB3)
  interrupt_disable();

    //  DIO -> OUTPUT
    //  CLK -> OUTPUT
  port_setup_set_data_direction_output_bits_port_B(0b00011000);
    //  CLK -> LOW
  port_set_low_bits_port_B(0b11101111);
    //  DIO -> LOW
  port_set_low_bits_port_B(0b11110111);

    //  setup del pin 10 come input
  port_setup_set_data_direction_input_bits_port_B(0b11111011);
    //  Abilito pull-down
  port_set_high_bits_port_B(0b11111011);
    //  Abilito il Bit relativo a PCINT4 nella maschera PCMSK0
  interrupt_msk_setup_set_interrupt_0_7_bits_pcmsk_0(0b00000100);
    //  Abilito la maschera 0 nel registro PCICR
  interrupt_pcicr_setup_set_interrupt_bits_pcie(0b00000001);

    //  setup interrupt
  interrupt_enable();

    //  Inizializzo i display tutti a 0
  BYTE initial_segments[] = { 0b00111111, 0b00111111, 0b00111111, 0b00111111 };
  display_7segments_set_segments(initial_segments, 4, 0);
  while(1)
  {
    _delay_ms(500);
  }
}

        //  la funzione interrupt handler non ha un valore di ritorno perchè la funzione sorgente (ISR,
interrupt_handler(PCINT0_vect)
{
    //  Algoritmo contatore
  int counter_temp = atoi(counter);
  counter_temp++;
  if(counter_temp<=9)
    sprintf(counter, "000%d", counter_temp);
  else if(counter_temp>9 && counter_temp<=99)
    sprintf(counter, "00%d", counter_temp);
  else if(counter_temp>99 && counter_temp<=999)
    sprintf(counter, "0%d", counter_temp);
  else if(counter_temp>999 && counter_temp<=9999)
    sprintf(counter, "%d", counter_temp);
  else
    strcpy(counter, "0000");

    //  Scrittura del numero attuale
  display_7segments_set_segment(digitToSegment[(int) counter[DISPLAY_3_ADDR] - '0'], DISPLAY_3_ADDR);
  display_7segments_set_segment(digitToSegment[(int) counter[DISPLAY_2_ADDR] - '0'], DISPLAY_2_ADDR);
  display_7segments_set_segment(digitToSegment[(int) counter[DISPLAY_1_ADDR] - '0'], DISPLAY_1_ADDR);
  display_7segments_set_segment(digitToSegment[(int) counter[DISPLAY_0_ADDR] - '0'], DISPLAY_0_ADDR);
}
```

Figura 9.14: Contenuto del file 7segment_Encoder_Interrupt_main.c [SublimeText2]

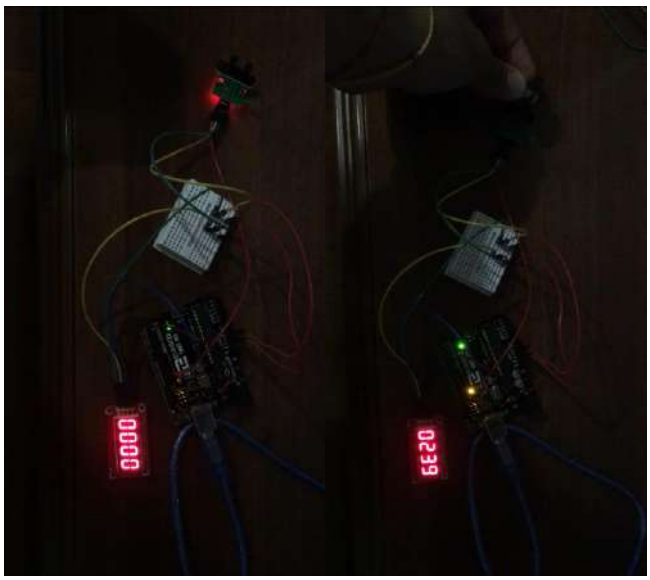


Figura 9.15: Esecuzione del programma di conteggio degli interrupt mandati da un encoder a disco, tramite un display a 7 segmenti.

CAPITOLO 10

Conclusioni

L'elaborato di questa tesi ha permesso di semplificare l'interfacciamento da parte del programmatore con Arduino, senza utilizzare le librerie messe a disposizione dall'IDE, che si pongono su di un livello più alto d'astrazione, mascherando troppo l'interazione con la scheda hardware. Le librerie sviluppate in questa tesi infatti, sono formate da istruzioni essenzialmente semplici e che mantengono un contatto diretto con Arduino.

Si è deciso di seguire questo tipo di approccio per rendere più evidente ciò che avviene tra il microcontrollore e le relative periferiche e per permettere al programmatore di toccare con mano ogni registro della interfaccia hardware.

Tutto ciò è inoltre molto utile a comprendere più a fondo la materia Calcolatori Elettronici, esercitando infatti, in maniera pratica, le argomentazioni e i concetti di quest'ultima.

Lo sviluppo di questo progetto ha agevolato una mia più profonda comprensione della materia e della programmazione hardware, richiedendo la pratica di concetti chiave di questo corso ma anche di altri corsi seguiti durante i miei studi, risultando perciò un punto d'incontro tra la programmazione e l'elettronica. Ogni esperimento infatti, oltre che per lo sviluppo del codice, richiedeva particolare attenzione anche per la fase di assemblaggio, in cui bisognava disporre e collegare le varie componenti in maniera da evitare di fulminare i dispositivi, facendo quindi attenzione ai valori della corrente e della tensione.

In questa tesi sono state create librerie solo per i dispositivi che sarebbero risultati utili a fini didattici, ma in eventuali sviluppi futuri se ne potranno implementare di nuove dedicate ad altri sensori, display o dispositivi simili, seguendo sempre la stessa logica, al fine di creare anche esperimenti che richiedono una difficoltà maggiore.

Ringraziamenti

Lo sviluppo di questa tesi rappresenta un nuovo capitolo da aggiungere ad una maturità personale e accademica che mi sta attraversando in questi anni. Una fase ancora incompiuta, che però ha coinvolto nel corso degli anni persone che si stanno rivelando fondamentali per la mia crescita ed eventuale realizzazione futura, e vorrei perciò dedicare ad esse quest'ultima parte della tesi.

In primo luogo, ringrazio il professore Stefano Mattoccia per il supporto, la fiducia e la disponibilità datomi in questi mesi.

Ringrazio papà Emanuele e mamma Carolina per avermi cresciuto e formato come persona, ma soprattutto per avermi dato la possibilità di studiare attraverso la loro dedizione e i loro sacrifici. Ringrazio mia sorella Francesca e mio cognato Gianluigi che, attraverso la loro esperienza, hanno sempre saputo indicarmi la strada più giusta. Ringrazio gli zii Antonio, Silvia, Anna e Salvatore, i cugini Francesco, Giulia e Andrea, che mi hanno dato modo di ambientarmi e mi hanno sempre supportato durante il mio soggiorno bolognese. Ringrazio gli zii Giovanni, Rina, Giuseppina e Rosario, i cugini Francesca, Rosario, Emanuele e Francesca, per essere sempre stati un punto di riferimento sin dalla mia infanzia. Ringrazio nonna Palma per essere stata la mia seconda madre quando i miei genitori non c'erano per lavoro, ma anche nonno Francesco, di cui ho pochi ricordi, tra cui il momento in cui faceva la "tana" dopo avermi trovato dietro l'armadio giocando a nascondino a 4 anni. Ringrazio nonno Rosario, perché essendo stato il nipote più piccolo ha sempre guardato me con occhi diversi ed è sempre stato premuroso, e ringrazio inoltre nonna Francesca, che purtroppo non ho mai conosciuto.

Tengo inoltre a ringraziare gli insegnanti Maria Chiara Abbruzzese, Filomena Gallella, Antonio Donnici, Luigina Alaggia, Maria Greco e Tommaso Cortese che, dall'elementari al liceo, sono stati maestri di vita oltre che degli ottimi insegnanti e tutt'ora ne conservo un bellissimo ricordo.

Un ringraziamento particolare va ora agli amici d'infanzia, Alfonso, Rosario, Luigi, Veronica, Giulio, Patrizia e tutti gli altri, per esser sempre stati presenti ed aver colto insieme a me tutte le gioie che l'adolescenza e la gioventù possono regalare. Dedico un "grazie" altrettanto importante a Edoardo, Alessandro, Paolo, Enrico, Giulio, Niccolò, Andrea, Arianna e tutti gli altri amici fantastici che ho conosciuto qui a Bologna, che hanno condiviso insieme a me tutte le ansie e le soddisfazioni che l'università può dare.

Un ultimo ringraziamento molto importante va a zio Domenico, che sin da piccolo ha fatto nascere in me la passione per l'informatica, mi ha supportato nell'affrontare il nuovo mondo dell'università e della programmazione, e ha sciolto ogni mia difficoltà grazie alla sua preparazione e alla sua esperienza nel campo.

Bibliografia

- [1] Massimo Banzi, BetaBook, il manuale di Arduino, *in* Apogeo, versione β, *13 dicembre 2009. URL consultato il 12 luglio 2011.*

- [2] Wikipedia, Arduino, https://it.wikipedia.org/wiki/Arduino_(hardware)

- [3] http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf

- [4] http://maxembedded.com/2015/06/setting-up-avr-gcc-toolchain-on-linux-and-mac-os-x/

- [5] https://balau82.wordpress.com/2011/03/29/programming-arduino-uno-in-pure-c/

- [6] https://www.tinkercad.com

- [7] http://www.automazioneos.com/programmare-larduino-in-puro-c-primi-passi/

- [8] https://www.instructables.com/image/F2SFHSDH3Z3V3P4

- [9] https://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Function-Attributes.html

- [10] https://github.com/vancegroup-mirrors/avr-libc/blob/master/avr-libc/include/avr/interrupt.h

- [11] https://playground.arduino.cc/Main/PinChangeInterrupt