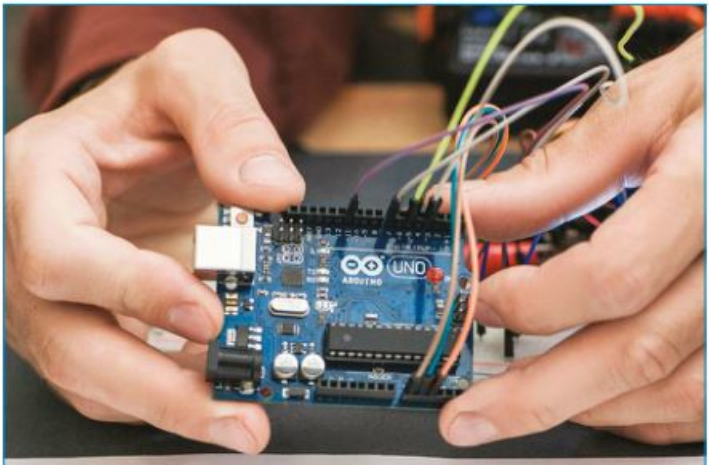
ELEMENTI DI BASE PER L'AUTOMAZIONE CON ARDUINO

modulo 7



CONOSCENZE

- Conoscere la scheda programmabile Arduino e i suoi elementi principali.
- Conoscere la programmazione base di Arduino (istruzioni base, operatori, ecc.)
- Conoscere le principali funzioni applicabili nell'automazione con Arduino.
- Conoscere le caratteristiche d'impiego dei componenti elettrici, elettronici e meccanici al fine del loro utilizzo negli automatismi.

ABILITÀ

- Interpretare le condizioni di esercizio di apparecchiature, componenti e impianti indicate in schemi e disegni.
- Assemblare componenti elettrici, elettronici e meccanici attraverso la lettura di schemi e disegni.
- Scrivere programmi che permettano la realizzazione di semplici automatismi.

1 ARDUINO E LA SUA PROGRAMMAZIONE DI BASE

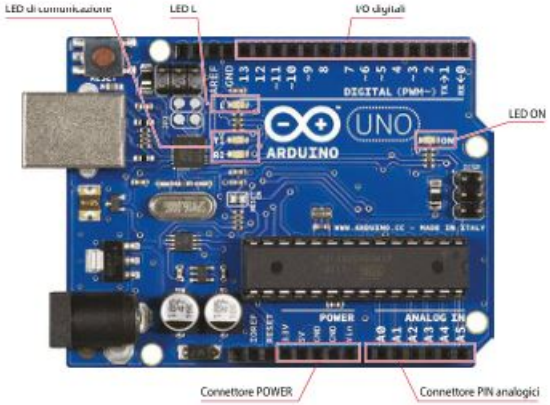
1 INTRODUZIONE AD ARDUINO

• *Logo di Arduino.*

Arduino è il nome di una serie di schede elettroniche di piccole dimensioni e basso costo, molto diffuse sia in ambito hobbistico, sia in contesti didattico-professionali, sviluppate da alcuni membri dell'*Interaction Design Institute* di Ivrea. Tali schede costituiscono una delle modalità più semplici per avvicinarsi al mondo dei microcontrollori e della programmazione; sono ottime, infatti, per sviluppare rapidamente piccole applicazioni.
L'ambiente per lo sviluppo del software è libero e ben documentato, così come tutti gli schemi circuitali.

1.1 SCHEDA

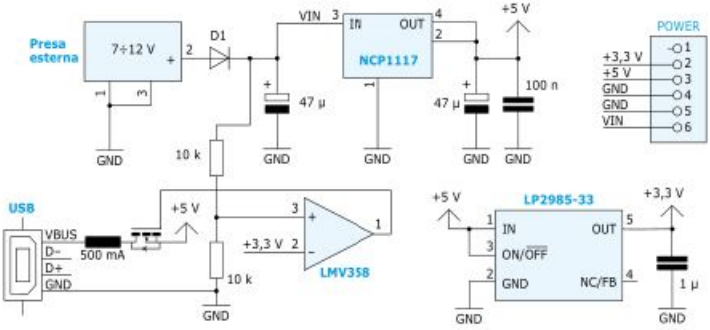
La scheda contiene un microcontrollore Atmel a 8 bit, un'interfaccia per la comunicazione USB, un oscillatore al cristallo da 16 MHz, una presa per l'eventuale alimentazione 7÷12 V, un tasto di reset e alcuni connettori per l'input e l'output dei segnali digitali e analogici.



• *Scheda Arduino UNO.*

In particolare, la scheda Arduino UNO contiene il microcontrollore ATmega328 che dispone di 32 kByte di memoria Flash per il programma, 2 kB SRAM, 1 kB EEPROM e 14 pin digitali di ingresso/uscita.
La **memoria Flash** non perde l'informazione anche togliendo l'alimentazione (non è volatile) ed è pre-programmata con un software per il caricamento dei programmi (*bootloader*).

La **memoria EEPROM** non è volatile e serve per la memorizzazione di eventuali parametri fissi, mentre la **SRAM** è volatile ed è usata per depositare i dati intermedi elaborati dal programma.
Le uscite digitali operano a 5 V e ogni pin può fornire o ricevere un massimo di 40 mA.
L'**alimentazione** della scheda è derivata dalla presa USB collegata al PC.
L'alimentazione esterna è richiesta solo se la scheda è utilizzata nell'applicazione finale (*target*), scollegata dal computer. In questo caso, la tensione di 7÷12 V fornita tramite la presa jack esterna, è convertita nei 5 V necessari al funzionamento interno e nei successivi 3,3 V (max. 50 mA), tramite due integrati stabilizzatori di tensione.



• *Hardware di alimentazione.*

Un transistor (di tipo MOSFET), posto in serie alla tensione VBUS proveniente dalla presa USB e governato da un amplificatore operazionale comparatore (LMV358), impedisce il conflitto tra la tensione VBUS e i 5 V generati dall'eventuale alimentazione esterna. Tutte le tensioni sono disponibili sui pin del connettore *Power*.

1.2 AMBIENTE DI SVILUPPO

L'ambiente di sviluppo (IDE, *Integrated Development Environment*) è scaricabile gratuitamente dal sito web http://www.arduino.cc.
IDE permette di comporre un programma in un linguaggio molto simile al C standard, controllarne la sintassi (*editor*), convertirlo nel codice macchina specifico del microcontrollore in uso (*compilatore*), caricarlo nella memoria del microcontrollore (*loader*), controllarne la corretta esecuzione e correggere eventuali errori (*debugger*).
Tutti i programmi (detti *sketch*) sono costituiti da due blocchi (o funzioni):

- il blocco iniziale **setup ()**, eseguito una volta sola all'accensione, nel quale vanno inserite tutte le istruzioni di configurazione del componente e di pre-set delle variabili da utilizzare nel programma;
- il blocco **loop ()**, eseguito in continuazione, nel quale è inserito l'algoritmo vero e proprio che gestisce l'applicazione desiderata.

• *Struttura di uno sketch.*

La scritta **void** che precede entrambi i blocchi sta a indicare che l'esecuzione della funzione non restituisce alcun valore.
L'inizio e il termine di ciascun blocco sono individuati dalle parentesi graffe, rispettivamente aperta ({) e chiusa (}).
Le istruzioni operative terminano con il **punto e virgola**.
I commenti che occupano una sola riga vanno scritti dopo la **doppia barra** (//), mentre i commenti di più righe vanno aperti con la coppia **/*** e chiusi con ***/**.

```
// commento breve, una sola riga
/* commento esteso
Occupa più righe
*/
```

I pulsanti della barra degli strumenti (tabella 1) consentono di velocizzare le operazioni.

TAB. 1 – PULSANTI VELOCI

VERIFICA	Controlla il codice per la ricerca di errori e lo compila
CARICA	Controlla il codice, lo compila e lo invia alla scheda
NUOVO	Crea un nuovo sketch
APRI	Apre il menu degli sketch memorizzati in precedenza, tra i quali è possibile selezionarne uno
SALVA	Salva sul PC lo sketch corrente
SERIAL MONITOR	Apre il serial monitor per il debug

2 PRIMI PASSI

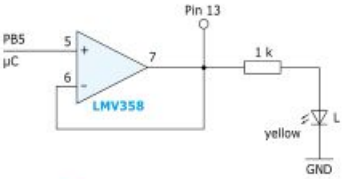
Per iniziare a utilizzare Arduino sono sufficienti pochi passi:
1. scaricare dal sito web http://www.arduino.cc il software Arduino e installarlo sul PC;
2. collegare la scheda in dotazione a una presa USB (il LED verde ON si accende);
3. aprire l'ambiente di sviluppo e controllare che in *Strumenti* → *Scheda* sia selezionato il tipo di scheda in dotazione (per esempio "Arduino/Genuino Uno");
4. selezionare da *File* → *Esempi* (oppure attivando il tasto ↑, Apri) il programma d'esempio *01.Basics* → *Blink*, che accende e spegne ripetutamente il LED L presente sulla scheda;
5. compilare il programma, attivando il comando *Sketch* → *Verifica/Compila*;
6. verificare che in *Strumenti* → *Porta* sia selezionata la porta seriale di comunicazione con la scheda;
7. caricare il programma (*uploading*) nella scheda utilizzando il comando *Sketch* → *Carica* (i LED TX e RX sulla scheda lampeggiano per qualche istante).

Cliccando sull'apposito tasto freccia orizzontale → il sistema compila e carica automaticamente il programma in lavorazione.
Terminato il caricamento, il programma entra in esecuzione, facendo lampeggiare il LED L, con intervalli di circa un secondo.

3 USCITE DIGITALI

Arduino UNO dispone di 14 terminali (pin) utilizzabili individualmente come ingressi o uscite digitali, numerati da 0 a 13.
In conformità alle richieste dell'applicazione, un pin può essere utilizzato come **ingresso**, se si vuole acquisire il livello logico del segnale digitale che gli è stato applicato, oppure come **uscita**, per emettere una tensione logica di controllo, di livello alto o basso, che azioni il dispositivo attuatore collegato al pin stesso.
Affinché Arduino sia in grado di utilizzare un determinato pin nella modalità richiesta

dal programmatore (in conformità all'hardware esterno collegato), questo deve essere preventivamente impostato in modalità input o output.
L'istruzione da utilizzare è **pinMode**, da inserire nel blocco setup ().
Per esempio, considerato che sulla scheda è presente un LED giallo (L) connesso con il pin 13 degli I/O digitali, per impostare il pin 13 in modalità output e governare il LED L, si può utilizzare l'istruzione:



• *Hardware relativo al LED L.*

```
pinMode (13, OUTPUT);
```

Questo significa che dal pin 13 potrà uscire un segnale digitale con tensione 5 V (livello logico alto, LED acceso) oppure 0 V (livello logico basso, LED spento).
Il livello logico da porre sull'uscita è da impostare tramite l'istruzione **digitalWrite**.
Per esempio:

```
digitalWrite (13, HIGH);
```

emette un livello logico alto sul pin 13, ovvero 5 V, accendendo il LED L, mentre

```
digitalWrite (13, LOW);
```

emette un livello logico basso, ovvero 0 V, spegnendo il LED.
Come programma di esercitazione si può riscrivere l'esempio *01.Basics → Blink*, compilarlo, caricarlo, controllarne l'esecuzione e salvarlo in una propria cartella.

```
/*Blink
accende e spegne il LED L ad intervalli di 1 s
*/
void setup() {
 pinMode (13, OUTPUT); // inizializza il pin 13 come uscita digitale
}
void loop() {
 digitalWrite (13, HIGH); // accende il LED L
 delay (1000); // attesa di 1 s
 digitalWrite (13, LOW); // spegne il LED L
 delay (1000); // attesa di 1 s
}
```

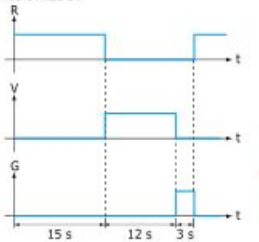
La funzione **delay (1000)** innesca l'esecuzione di un programma che impegna il processore per 1.000 ms; il loop principale si ripete quindi ogni 2 s.
Per ritardi con risoluzione in microsecondi è disponibile la funzione:

```
delayMicroseconds (n° µs);
```

Per entrambe le funzioni di ritardo, l'argomento è un numero (di tipo **unsigned long**) che può arrivare fino a 4.294.967.295.

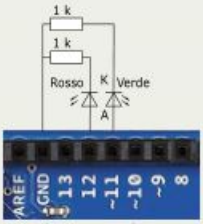
Esempio applicativo

Un esempio tipico di utilizzo della funzione **delay** è l'accensione in sequenza periodica delle luci di un semaforo.
Utilizzando il LED giallo sulla scheda, connesso al pin 13, basta aggiungere due LED esterni sui pin 12 (rosso) e 11 (verde).



• *Temporizzazione di un semaforo.*

```
/* Semaforo */
#define Giallo 13
#define Rosso 12
#define Verde 11
void setup() {
 pinMode (Giallo, OUTPUT);
 pinMode (Rosso, OUTPUT);
 pinMode (Verde, OUTPUT);
}
void loop() {
 digitalWrite (Giallo, LOW);
 digitalWrite (Rosso, HIGH);
 delay (15000); // attesa di 15 s
 digitalWrite (Rosso, LOW);
 digitalWrite (Verde, HIGH);
 delay (12000); // attesa di 12 s
 digitalWrite (Verde, LOW);
 digitalWrite (Giallo, HIGH);
 delay (3000); // attesa di 3 s
}
```



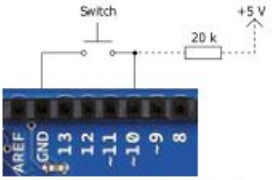
• *Connessione dei LED esterni, rosso e verde.*

La direttiva **#define** consente di sostituire nella scrittura del programma le costanti numeriche con espressioni più facili da ricordare.

4 INGRESSI DIGITALI

Un pin di I/O digitale può essere predisposto ad acquisire il livello logico di un comando digitale, impostandolo come ingresso mediante l'istruzione **pinMode (*n° pin*, INPUT)**, da inserire nel blocco setup ().
Per esempio, volendo acquisire lo stato logico di un interruttore (**switch**) connesso tra il pin 10 e massa, prima si imposta il pin 10 in modalità ingresso, utilizzando l'istruzione:

```
pinMode (10, INPUT);
```



• *Connessione di uno switch attivo basso.*

e, successivamente, l'istruzione **digitalRead** (n° pin)

```
digitalRead (10);
```

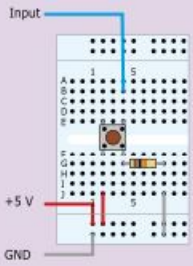
per ottenere il livello logico, alto o basso, presente sul pin 10.
Poiché lo switch può solo chiudere a massa l'ingresso, serve una **resistenza** perennemente connessa con il positivo (**pull-up**), in modo che l'ingresso risulti sicuramente a livello logico alto quando lo switch è aperto. Senza tale resistenza, il livello logico acquisito in condizioni di tasto aperto risulterebbe incerto, a causa della presenza del rumore ambientale.
L'inserimento automatico di una resistenza di pull-up (sconnessa di default) del valore di 20÷50 kΩ su un pin configurato come ingresso digitale è realizzato con il comando **digitalWrite (n° pin, HIGH)**.

Di seguito è proposto, a titolo di esempio, un programma che accende e spegne il LED L (pin 13) in base alla pressione di un pulsante collegato al pin 10.

```
/* Ingresso */
#define Giallo 13
#define Tasto 10
boolean pippo;
void setup() {
 pinMode (Giallo, OUTPUT);
 pinMode (Tasto, INPUT);
 digitalWrite (Tasto, HIGH); // pull-up
}
void loop() {
 pippo = digitalRead(Tasto);
 digitalWrite (Giallo, pippo);
}
```

Tilt sensor

Un sensore di pendenza è un componente in grado di rilevare la pendenza dell'oggetto su cui è montato.
Nella sua forma più semplice, il dispositivo presenta due pin e contiene una biglia metallica che, quando è in una determinata posizione, chiude il contatto elettrico tra i due terminali, mentre quando risulta inclinata per più di un certo angolo, lo apre.
Dal punto di vista elettrico, equivale a uno switch digitale, da acquisire tramite un ingresso, con opportuna resistenza di pull-up o di pull-down.



• *Tilt sensor.*

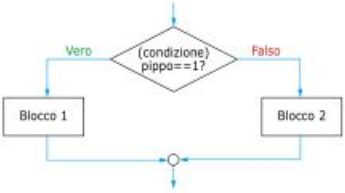
• *Input da switch, attivo alto, con resistenza di pull-down.*

5 ISTRUZIONI DI SELEZIONE IF-ELSE

La programmazione di Arduino rispetta la grammatica del linguaggio C, in particolare per tutto ciò che riguarda istruzioni, tipi di dati, operatori aritmetici e logici.
È importante, innanzitutto, distinguere tra sequenze e selezioni, perché, mentre le **sequenze** sono strutture caratterizzate da un numero finito di operazioni da eseguire in un ordine preciso, le **selezioni** consentono, al verificarsi o meno di una certa condizione, di eseguire una sequenza di istruzioni invece di un'altra.
Le istruzioni di selezione permettono, infatti, di eseguire un blocco di codice (sequenza) solo se è verificata una determinata condizione logica chiusa tra parentesi tonde.
L'istruzione più nota è **if-else**.

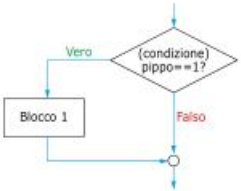
```
if (condizione == TRUE) {
 blocco 1
}
else {
 blocco 2
}
```

Nel caso in cui la condizione sia vera viene eseguito solo il primo blocco di istruzioni (blocco 1), mentre se è falsa viene eseguito solo il secondo blocco (blocco 2). Si parla di **selezione completa**.



• *Selezione completa.*

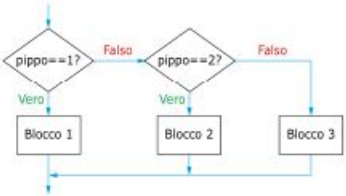
Se esiste solo un blocco da eseguire in modo condizionato (**selezione semplice**), si utilizza la selezione con la sola *if*, senza *else*.



• *Selezione semplice.*

In caso di **selezione multipla** si può utilizzare la codifica *if, else if, else if, ... , else*.

```
if (pippo == 1) {
 blocco 1
}
else if (pippo == 2) {
 blocco 2
}
else {
 blocco 3
}
```



• *Selezione multipla.*

Se il blocco da eseguire è costituito da una sola istruzione, si possono evitare le parentesi graffe.

5.1 OPERATORI DI COMPARAZIONE

Dire che una condizione logica è vera equivale a dire che assume un valore binario pari a 1, ma in senso booleano è vera qualsiasi condizione diversa da zero, per cui –50, –7, –1, 3, 9 sono tutti valori veri. Nel codice della **condizione di selezione**, per confrontare due elementi si usa il **doppio uguale** (==), mentre il singolo carattere di uguale (=) è utilizzato per assegnare un valore ad una variabile. Pertanto

```
if (pippo = 1)
```

non esprime un confronto, ma assegna il valore 1 alla variabile pippo, determinando una condizione sempre vera.
Oltre al doppio uguale, ci sono anche altri operatori di comparazione:

```
x != y // x diverso da y
x < y // x minore di y
x > y // x maggiore di y
x <= y // x minore o uguale a y
x >= y // x maggiore o uguale a y
```

5.2 OPERATORI LOGICI RELAZIONALI

Laddove la condizione di selezione è composta dal verificarsi di più comparazioni, queste vanno poste in combinazione logica tra loro, utilizzando gli operatori logici relazionali **&&** (doppio and), **||** (doppio or) e **!** (not), il cui risultato è solo **vero/falso**.
Si osservino i seguenti esempi.

- La selezione

```
if ( (A > B) && (C < D) )
```

è vera se entrambe le condizioni (A > B) e (C < D) sono vere.

- La selezione

```
if ( (A > B) || (C < D) )
```

è vera se almeno una delle condizioni (A > B) oppure (C < D) è vera.

- La selezione

```
if ( !(A > B) )
```

è vera se A <= B.

È possibile ricorrere anche a una forma di **selezione contratta**, meglio comprensibile con un esempio:

```
a = b > 9 ? 5 : 0 ;
```

equivale, infatti, alla struttura:

```
if ( b > 9) a= 5;
else a= 0;
```

Per esempio:

```
#define min (x,y) ((x) < (y)) ? (x) : (y)
```

definisce la funzione *min* che restituisce il numero minore;

```
#define max (x,y) ((x) > (y)) ? (x) : (y)
```

definisce la funzione *max* che restituisce il numero maggiore;

```
#define Dec (x) ((x > '9') ? (x - 'A' + 10) : (x - '0'))
```

converte una cifra esadecimale ASCII in decimale;

```
#define Asc (x) ((x > 9) ? (x - 10 + 'A') : (x + '0'))
```

converte una cifra esadecimale in ASCII.

6 TIPI DI DATI

Una **variabile** è un contenitore in grado di ospitare un valore modificabile, mentre una **costante** è un numero fisso e non modificabile, tanto che il tentativo di cambiarne il valore nel corso del programma viene segnalato come errore.
Il valore numerico relativo a una variabile o a una costante può essere espresso in una qualsiasi delle forme numeriche più note.
Le variabili utilizzate dal programma devono essere preventivamente dichiarate, specificandone il **tipo** tra le opzioni indicate in tabella 2 (pagina seguente) ed eventualmente settandone il valore.
Per esempio, l'istruzione

```
int Pippo = 0;
```

dichiara che la variabile *Pippo* è un intero e che il suo valore iniziale è *0*.

Tab. 2 – Tipi di variabili dichiarabili in Arduino

Tipo	Dicitura	N° di byte occupati	Valore	
			minimo	massimo
Boolean	Var. Binaria	1	LOW, false	HIGH, true
Char	Carattere	1	−128	+127
Byte	Intero da 8 bit senza segno	1	0	255
Int	Intero con segno	2	−32.768	+32.767
Unsigned Int	Intero senza segno	2	0	65.535
Long	Intero lungo	4	−2.147.483.648	+2.147.483.647
Unsigned Long	Intero lungo senza segno	4	0	+4.294.967.295
Float	Numero reale	4	$-3{,}4 \cdot 10^{38}$	$+3{,}4 \cdot 10^{38}$
Double	Numero reale lungo	8	$-1{,}797 \cdot 10^{308}$	$+1{,}797 \cdot 10^{308}$

Il tipo determina implicitamente l'**occupazione di memoria** riservata alla variabile e quindi gli estremi minimo e massimo che questa può assumere.
Il tipo **boolean** specifica una variabile binaria, che può assumere solo i valori true/false oppure high/low.
Il tipo **char** contiene un singolo carattere alfanumerico (lettera o numero), memorizzato sotto forma di numero a 8 bit, secondo la **codifica ASCII**.
Per esempio, l'istruzione:

```
char pippo = 'A';
```

dichiara una variabile ad 8 bit, di nome pippo e gli assegna il valore decimale 65.
Eseguendo

```
pippo = pippo + 1;
```

il valore della variabile pippo diventa 66, corrispondente alla lettera 'B'.
Il tipo **byte** specifica un intero senza segno a 8 bit, i cui valori possono andare da 0 a 255.
Perciò, eseguendo le due istruzioni

```
byte pluto = 255;
pluto = pluto + 1;
```

la variabile pluto contiene al termine il valore 0.
Analogamente, eseguendo

```
byte pluto = 0;
pluto = pluto - 1;
```

la variabile pluto contiene al termine il valore 255.
I tipi **int** sono utilizzati per dichiarare variabili che possono contenere solo valori interi, positivi o negativi, senza decimali. Il bit più significativo assume il ruolo di segno, perciò il valore numerico non può superare i 15 bit senza entrare in errore di overflow (+32.767 + 1 = −32.768) o underflow (−32.768 − 1 = +32.767), come già visto per il tipo byte.
Se un intero è dichiarato **unsigned**, tutti i 16 bit sono considerati cifre binarie e la variabile può assumere valori da 0 a 65.567.
Il tipo **long** è un intero con capacità maggiori.
I tipi **float** e **double** sono utilizzati per dichiarare le variabili reali, con la virgola. Sono da utilizzare con parsimonia, perché la loro elaborazione matematica impegna parecchie risorse in termini di tempo di esecuzione.
Per la dichiarazione delle **costanti** valgono gli stessi tipi di dato validi per le variabili. Una **costante** scritta senza prefissi è considerata in forma decimale. È considerata, invece, espressa in binario se al numero è anteposta la B, in ottale se è anteposto lo 0 e in esadecimale se è anteposta la coppia 0x.

Per esempio:
- 101 vale 101 **decimale**;
- B101 è un numero **binario**, corrispondente a 5 decimale;
- 0101 è un numero **ottale**, corrispondente a 65 decimale;
- 0x101 è un numero **esadecimale**, corrisponcente a 257 in decimale.

7 OPERATORI ARITMETICI

La programmazione in Arduino permette l'utilizzo di alcuni operatori aritmetici semplici:
- = (assegnazione);
- + (somma);
- – (sottrazione);
- * (prodotto);
- / (quoziente),
- % (resto della divisione tra interi).

Tra operandi con uguale numero di bit, è possibile eseguire anche operazioni logiche bit a bit (**bitwise**) utilizzando gli operatori **&** (and), **|** (or), **^** (xor), **~** (not, complementa ogni singolo bit).

7.1 OPERATORI DI SCORRIMENTO

È possibile eseguire, inoltre, operazioni di scorrimento di una posizione (**bitshift**), a destra o a sinistra, del gruppo di bit che costituiscono il valore di una variabile di tipo intero mediante l'utilizzo degli **operatori di shift**:
- << (bitshift left);
- >> (bitshift right);

Per esempio, se pippo = 6 = B00000110, eseguendo

```
pluto = pippo << 2; // shift a sinistra di due posizioni
```

al termine pluto = B00011000 = 24.
Eseguendo invece

```
paperino = pippo >> 1; // shift a destra di una posizione
```

al termine paperino = B00000011 = 3.
Gli operatori di scorrimento sono frequentemente utilizzati per facilitare le operazioni su singoli bit di una variabile, come nel caso delle #define che seguono:

```
#define setbit (x, y) y = y | ( 1<< x)
#define resbit (x, y) y = y & ~ ( 1<< x)
#define testbit (x, y) ( y & ( 1<< x)
```

Negli esempi proposti, la maschera con la quale eseguire l'operazione logica bit a bit con la variabile y è ottenuta facendo scorrere l'unità (1 = B00000001) a sinistra di x posizioni (1 << x). Per esempio, 1 << 3 = B00001000
Tali #define sono da utilizzarsi, per esempio, come:

```
setbit (3, pippo);
```

porta a 1 il bit di peso 2^3 della variabile pippo;

```
resbit (3, pippo);
```

porta a 0 il bit di peso 2^3 della variabile pippo;

```
if (testbit (3, pippo)) { ... }
```

se il bit di peso 2^3 della variabile pippo è diverso da 0, esegui ...;

```
while (!testbit (3, pippo));
```

mentre il bit di peso 2^3 della variabile pippo è uguale a 0, attendi.

8 OPERATORI COMPOSTI

Sono dette **operatori composti** (*compound*) alcune forme di scrittura contratta che velocizzano la stesura del software.
I principali sono **++** (**increment**) e **--** (**decrement**) che, rispettivamente, incrementano e decrementano di un'unità una variabile intera.

- `x++` equivale a scrivere `x = x + 1`
- `x--` equivale a scrivere `x = x - 1`

Nelle assegnazioni, bisogna distirguere tra **post-increment** e **pre-increment**.
Per esempio, supponendo ogni volta x = 5:

- `y = x++`
  prima assegna x a y (y = 5) e poi incrementa x (x = 6);
- `y = ++x`
  prima incrementa x (x = 6) e poi lo assegna a y (y = 6);
- `y = x--`
  prima assegna a y il valore di x (y = 5) e poi decrementa x (x = 4);
- `y = --x`
  prima decrementa x (x = 4) e poi lo assegna a y (y = 4).

Altre forme matematiche contratte, valide per variabili di ogni tipo, sono per esempio:

- `x += y` equivalente all'espressione `x = x + y;`
- `x -= y` equivalente all'espressione `x = x - y;`
- `x *= y` equivalente all'espressione `x = x * y;`
- `x /= y` equivalente all'espressione `x = x / y;`

Per esempio, supponendo ogni volta x = 2:

- `x += 4` dà come risultato x = 6;
- `x -= 3` dà come risultato x = –1;
- `x *= 10` dà come risultato x = 20;
- `x /= 2` dà come risultato x = 1.

Un esempio di utilizzo della forma contratta è

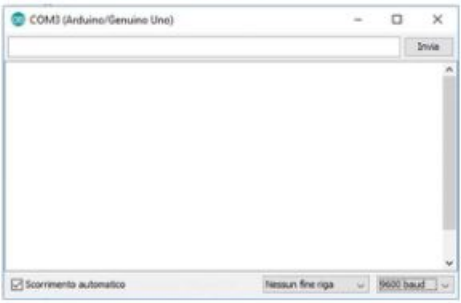
```
PITR &= ~7;
```

forma contratta dell'equivalente

```
PITR =PITR & ~7;
```

che azzera i 3 bit meno significativi di PITR, in quanto, presa la costante numerica 7 (=B00000111) e negata (~7=B11111000), esegue l'AND logico bit a bit tra questa e la variabile PITR e lo assegna alla variabile stessa.

9 DEBUG DEL PROGRAMMA



Per il debug del programma, è disponibile una libreria di funzioni per lo scambio seriale di informazioni tra la scheda e l'ambiente di sviluppo (o altri dispositivi).
La comunicazione è seriale e, se attivata, impegna i pin 0 (RX) e 1 (TX), che in tal caso non sono più disponibili come I/O.
La finestra di comunicazione si apre selezionando *Strumenti → Monitor seriale*.

- *Finestra di comunicazione.*

Le istruzioni disponibili sono:

```
Serial.begin (9600);
```

imposta la velocità della comunicazione a 9.600 bit/s;

```
Serial.flush ();
```

svuota l'eventuale coda di dati già presenti in ricezione;

```
Serial.print ("stringa");
```

invia una stringa di caratteri ASCII;

```
Serial.println (variabile);
```

invia il valore della variabile e aggiunge il carattere di ritorno a capo (ASCII 13, oppure '\r') e di salta riga (ASCII 10, oppure '\n');

```
Serial.write (variabile);
```

invia il carattere ASCII corrispondente al valore della variabile;

```
if (Serial.available ( ) > 0) {
 pippo = Serial.read ( );
}
```

se esistono dati in ricezione, acquisisce il byte ricevuto.
Per facilitarne la comprensione, la maggior parte delle istruzioni di comunicazione è stata inserita nello sketch d'esempio proposto di seguito.

```
/* debug seriale */
byte pippo;
void setup() {
 Serial.begin (9600);
 Serial.flush ();
 byte pippo = 0;
}
void loop() {
 Serial.print ("ciclo num. ");
 Serial.println (pippo, DEC);
 pippo = pippo +1;
 if (Serial.available () > 0) {
  pippo = Serial.read ();
 }
 delay (2000); // attesa di 2 s
}
```

Se, durante il funzionamento del programma, dalla finestra di comunicazione si invia, per esempio, il carattere ASCII '0' (valore decimale 48), il ciclo riparte dal valore indicato.

9.1 SERIAL.WRITE E SERIAL.PRINT

Le funzioni Serial.write e Serial.print hanno funzioni differenti.
La funzione **Serial.write( )** invia un singolo byte come singolo carattere ASCII.
Per esempio:

```
Serial.write (byte(78)) // invia il carattere N (il cui valore ASCII è 78)
```

La funzione **Serial.print( )**, invece, invia i numeri interi utilizzando un carattere ASCII per ogni cifra, i numeri reali (float) utilizzando un carattere ASCII per ogni cifra ma con solo due decimali, mentre i singoli caratteri ASCII e le stringhe di caratteri sono trasmessi senza variazioni.

Per esempio:

```
Serial.print (78); // invia 78
```

```
Serial.print (1.23456); // invia 1.23
```

```
Serial.print ('N'); // invia N
```

```
Serial.print ("Hello world."); // invia Hello world
```

La funzione *Serial.print( )* può avere, inoltre, un secondo **parametro opzionale** per specificare il formato di invio della variabile da trasmettere.
Il formato può essere: BIN (binario), DEC (decimale), HEX (esadecimale).
Per i numeri in *floating point* il secondo parametro indica il numero di decimali da prendere in considerazione.
Per esempio:

```
Serial.print (78, BIN); // invia 1001110
```

```
Serial.print (78, DEC); // invia 78
```

```
Serial.print (78, HEX); // invia 4E
```

```
Serial.print (1.23456, 0); // invia 1
```

```
Serial.print (1.23456, 2); // invia 1.23
```

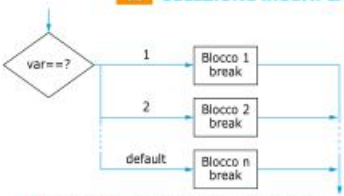
```
Serial.print (1.23456, 4); // invia 1.2346
```

Disponendo di una informazione a 10 bit (0÷1023)

```
Serial.print (value >> 8, BIN);
// invia i 2 bit di peso maggiore
```

```
Serial.print (value % 256, BIN);
// invia gli 8 bit di peso minore (tranne eventuali zeri significativi)
```

10 SELEZIONE MULTIPLA SWITCH/CASE



• *Struttura della selezione multipla switch/case.*

La gestione di algoritmi complessi richiede strutture di programmazione più performanti rispetto alla semplice selezione.
L'istruzione di selezione multipla **switch/case** permette di definire in modo semplice l'esecuzione di blocchi di codice mutualmente esclusivi, evitando di ricorrere a lunghe catene di tipo if-else.
La porzione di codice da eseguire è scelta in funzione del valore di una variabile sotto test.
La forma scritta della selezione è la seguente:

```
switch (var) {
 case 1:
  // istruzioni da eseguire se var == 1
  break; // uscita dallo switch
  case 2:
  // istruzioni da eseguire se var == 2
  break;
  ..
 default: // eventuale
  // istruzioni da eseguire solo se nessun case ha avuto successo
  break;
}
```

Il programma verifica ciascun *case* e se il risultato è positivo esegue tutte le istruzioni (eventualmente anche quelle dei *case* successivi) fino a quando non incontra un *break*.
Nell'esempio a fianco, se la variabile vale V1 oppure V2, vengono eseguite sia le istruzioni 2 sia le istruzioni 5.
Il comando **break**, dove presente, interrompe il controllo dei *case* successivi.

```
switch (variabile) {
 case V1:
 case V2:
  // istruzioni 2
 case V5:
  // istruzioni 5
  break;
 case V3:
  // istruzioni 3
 case V9:
 case V8:
  // istruzioni 8
  break;
}
```

Programma d'esempio

È riportato di seguito, a titolo di esempio, un programma che compone un numero a tre cifre, acquisendolo da tre caratteri ASCII inviati tramite la linea seriale.
Noti i codici ASCII (0 = 0x30, 1 = 0x31, ecc.), il valore decimale di ciascuna cifra si ottiene sottraendo 0x30 al codice ASCII corrispondente.

```
/* Acq. n° a 3 cifre */
int num_acq, conta, num_finale;
void setup() {
 Serial.begin (9600);
 Serial.flush ();
 conta = 0;
 num_acq = 0;
 num_finale = 0;
}
void loop() {
 if (Serial.available() > 0) {
  num_acq = Serial.read () - 0x30;
  switch (conta) {
   case 0:
    num_acq *= 100;
    break;
   case 1:
    num_acq *= 10;
    break;
   case 2:
    break;
  }
  num_finale = num_finale + num_acq;
  conta++;
 }
 if (conta == 3){
  Serial.println(num_finale);
  num_finale = 0;
  conta = 0;
 }
}
```

11 CICLO FOR

A differenza della selezione, una struttura di **iterazione** (ciclo for) consente di ripetere ciclicamente una sequenza di istruzioni (anche una sola), racchiusa tra parentesi graffe, fino a che risulta verificata una determinata condizione.
L'intestazione di un ciclo **for** prevede tre campi: inizializzazione, condizione di permanenza e operazioni di fine loop.

```
for (inizializzazione; condizione di permanenza; operazioni di fine loop) {
 // istruzioni da eseguire;
}
```

L'**inizializzazione** è l'insieme di una o più istruzioni da eseguire una volta sola prima di entrare nel loop.

• *Struttura di un ciclo for.*

La **condizione di permanenza** è testata prima di ogni ciclo e, se risulta vera, vengono eseguite tutte le istruzioni presenti tra le graffe e le **operazioni di fine loop**.
Quando la condizione risulta falsa, il loop termina.
Il **ciclo for** utilizza normalmente l'incremento di un contatore per determinare la permanenza nel loop.
Per esempio il ciclo

```
for (i = 0; i < 10; i ++) (// loop)
```

esegue 10 volte le istruzioni racchiuse tra le graffe, mentre il ciclo

```
for (i = 0, ch = 'a'; i <= 6; i ++) pippo ();
```

esegue, invece, una sola volta le due assegnazioni iniziali e 7 volte la funzione pippo ().

Il ciclo for è **molto flessibile**, per esempio ammette anche che alcuni o tutti i tre elementi di configurazione possano mancare (non devono mancare, però, i punti e virgola separatori).
Potrebbe, quindi, esistere un ciclo come

```
for ( ; ; )
```

Sarebbe un ciclo infinito, che non termina mai.

Programma d'esempio

L'esempio che segue emette i 94 caratteri ASCII con codice dal 33 (carattere "!") al 126 (carattere "~") utilizzando un ciclo for.

```
/* for */
byte num;
void setup ( ) {
 Serial.begin (9600);
 Serial.flush ( );
}
void loop( ) {
 Serial.println ("");
 for (num = 33; num < 127; num++) {
  Serial.write (num);
  if ((num -32)% 10 == 0)
   Serial.println ("");
 }
 for ( ; ; ) { }
}
```

12 CICLI WHILE E DO-WHILE

Il ciclo **while** esegue in continuazione le istruzioni contenute tra le graffe, fintanto che l'**espressione** tra parentesi è vera (o diversa da 0). Tale espressione è perciò detta **condizione di ingresso** nel ciclo.

```
while (espressione) {
 // istruzioni da eseguire;
}
```

• *Schema a blocchi di un ciclo while.*

Pertanto, se la condizione di ingresso non diventa mai falsa, il loop risulta **infinito**.
Al contrario, se la condizione è falsa già al primo mpatto, il loop non viene eseguito **nemmeno una volta**.
L'esempio che segue è un loop *while* che si ripete 200 volte.

```
var = 0;
while (var < 200) {
 // istruzioni da ripetere 200 volte;
 var++;
}
```

Il ciclo **do-while** lavora in modo analogo al ciclo while, con la differenza che la condizione è testata al termine del ciclo e funziona, perciò, come **condizione di permanenza** nel ciclo.

```
do {
 // istruzioni da eseguire;
} while (condizione di permanenza);
```

• *Schema a blocchi del ciclo do-while.*

La differenza è, quindi, che il ciclo *do-while* è sempre eseguito almeno una volta.
Affinché il ciclo possa ripetersi, la condizione di permanenza deve risultare vera (o diversa da 0).
Nell'esempio proposto di seguito, il loop *do-while* attende che il valore acquisito dal sensore raggiunga almeno il valore 100.

```
do {
 x = readSensors ( );
} while (x < 100);
```

Break e continue

L'istruzione **break**, necessaria per uscire da una selezione switch, può essere utilizzata anche per uscire immediatamente da un loop **for**, **while** o **do-while**, bypassando il test della condizione propria del loop. Per esempio:

```
for (x = 0; x < 255; x ++) {
 digitalWrite (PWMpin, x);
 sens = analogRead (sensorPin);
 if (sens > threshold){
  // bail out on sensor detect
  x = 0;
  break;
 }
 delay (50);
}
```

L'istruzione **continue**, invece, è da utilizzare all'interno di un loop **for**, **while** o **do-while** quando si vuole bypassare la porzione di codice rimanente, successiva all'istruzione stessa, e saltare direttamente al test della condizione di permanenza o di ingresso del ciclo stesso. L'istruzione continue perciò non interrompe il loop. Nell'esempio che segue, l'istruzione continue impedisce l'emissione dei valori compresi tra 41 e 119.

```
for (x = 0; x < 255; x ++) {
 if (x > 40 && x < 120) continue;
 digitalWrite (PWMpin, x);
 delay (50);
}
```

13 USCITE PWM

Arduino dispone anche di alcuni pin per l'acquisizione di segnali provenienti dai sensori posizionati sull'automatismo e di pin facilmente programmabili per controllare la potenza da inviare agli attuatori.
Un **segnale PWM** (*Pulse Width Modulation*) è un segnale rettangolare con larghezza d'impulso variabile. Un'uscita PWM può risultare utile, quindi, per controllare la potenza da inviare ad attuatori quali motori, lampade, riscaldatori o servomotori. Arduino dispone di 6 pin digitali, rumerati rispettivamente 3, 5, 6, 9, 10 e 11, che, se definiti come uscite, possono produrre segnali PWM a circa 490 Hz, con duty-cycle regolabile dallo 0% al 100%.
Sebbene discreta, l'informazione contenuta in un segnale PWM può essere considerata di tipo analogico, pertanto, nonostante i pin dedicati siano tra quelli definiti digitali, per impostare il valore del duty cycle da emettere si utilizza l'istruzione

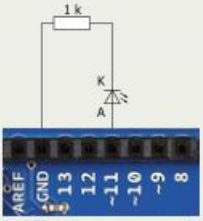
```
analogWrite (n° pin, valore);
```

con il valore della grandezza modulante che può variare tra 0 e 255 (risoluzione 8 bit).

Programma d'esempio

Il programma che segue controlla l'illuminazione prodotta dal LED connesso sul pin 11, aumentando e diminuendo gradualmente il PWM che lo governa.

```
/* PWM */
#define PWMpin 11
void setup() {
 pinMode (PWMpin, OUTPUT);
}
void loop ( ) {
 for (int i =0 ; i <= 255; i++) {
  analogWrite (PWMpin, i);
  delay (10);
 }
 for (int i = 255; i >= 0; i--) {
  analogWrite (PWMpin, i);
  delay (10);
 }
}
```



• *LED esterno connesso sul pin 11.*

14 INGRESSI ANALOGICI

Un **ingresso analogico** è una risorsa che permette di acquisire il valore di una tensione rispetto al valore di fondo scala, esprimendo il risultato della conversione sotto forma di numero binario. Arduino dispone di 6 ingressi analogici (*analog in*), etichettati da A0 ad A5, per l'acquisizione di tensioni da 0 a 5 V con 10 bit di risoluzione (2^{10} = 1.024 valori possibili, da 0 a 1.023).
L'istruzione di acquisizione è:

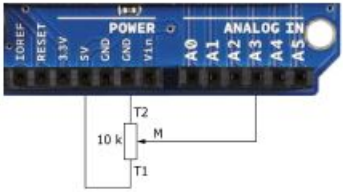
```
analogRead (n° pin);
```

Per esempio, il ciclo che segue rileva il valore di un ingresso analogico ogni 10 escursioni di loop.

```
void loop ( ) {
 i++;
 if ((i % 10) == 0)
  x = analogRead (sensPin);
 / ...
}
```

Programma d'esempio

L'esempio che segue attiva un PWM sul LED connesso al pin 11, proporzionale al valore analogico letto dal pin A3.



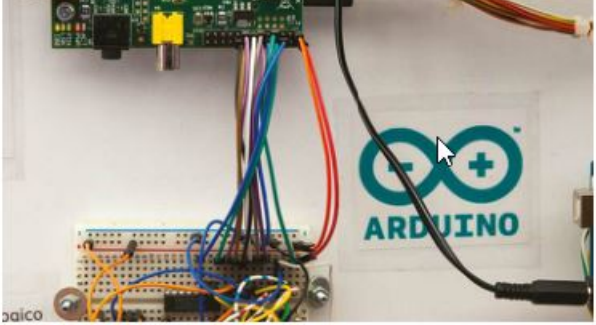
• *Potenziometro connesso sul pin A3.*

• *Potenziometro.*

```
int ledPin = 11;
int analogPin = 3;
int val = 0;
void setup ( ) {
 pinMode (ledPin, OUTPUT); }
void loop ( ) {
 val = analogRead (analogPin);
 analogWrite (ledPin, val >> 2);
}
```

Poiché la funzione *analogRead* restituisce un valcre compreso tra 0 e 1.023, questo va shiftato a destra di due bit (diviso per 4) per renderlo compatibile con il range da 0 a 255 ammesso dalla funzione *analogWrite*.
Sebbene l'impostazione predefinita preveda l'acquisizione di una tensione fino a 5 V, è possibile cambiare il fondo scala utilizzando il pin AREF e la funzione *analogReference ( )*.
Le opzioni sono:

- DEFAULT, per un fondo scala di 5 V;
- EXTERNAL, per impiegare come fondo scala la tensione applicata al pin AREF (con una resistenza da 5 kΩ in serie).



2 FUNZIONI E GESTIONE IN TEMPO REALE

1 LE FUNZIONI

Per semplificare la struttura di un programma, è spesso utile raggruppare **istruzioni** richiamate più volte o che realizzano una determinata operazione complessa, all'interno di un contenitore da richiamare con il solo nome.
Tale contenitore, definito **funzione**, è racchiuso tra due parentesi graffe, con anteposto un nome.
La struttura di una funzione prevede 3 campi:

- <**tipo** del valore restituito>
- <**nome** della funzione>
- (<elenco dei **parametri** da passare alla funzione>)

Se la funzione non restituisce alcun valore, il tipo indicato è **void** (come setup () e loop ()). Se la funzione non necessita di parametri, le parentesi tonde sono vuote.
Per esempio, la funzione *MulFunc*, proposta di seguito, si aspetta due argomenti interi (a e b) e restituisce un intero (il risultato del prodotto).

```
int MulFunc (int a, int b) {
 int ris = a * b;
 return ris;
}
```

La funzione che segue, invece, non richiede alcun parametro di chiamata.
Compara, infatti, l'input acquisito dall'ingresso analogico 0 con un livello di soglia (*threshold*) pari a 400, restituendo il valore 1 in caso l'input sia maggiore della soglia o il valore 0 se minore o uguale.

```
int checkSensor () {
 if (analogRead (0) > 400) return 1;
 else return 0;
}
```

L'istruzione **return**, seguita da un valore costante o da una variabile, restituisce tale valore alla funzione chiamante.
Per esempio, nell'istruzione

```
if (checkSensor ( )){…}
```

l'argomento stesso del test è costituito dal valore restituito dalla funzione *checkSensor*.
Come ulteriore esempio di utilizzo dell'istruzione *return*, si analizzi il funzionamento della funzione *RSens* che acquisisce 5 campioni a 10 bit da un sensore analogico connesso sul pin A0 e ne restituisce la media, scalandola a 8 bit (0÷255).

```
int RSens ( ){
 int i;
 int sval = 0;
 for (i = 0; i < 5; i++){
  sval = sval + analogRead(0);
 }
 sval = sval / 5; // media
 sval = sval >> 2; // scala da 10 a 8 bit
 return sval;
}
```

Poiché la funzione restituisce un intero, la sua chiamata va assegnata a una variabile int, per esempio:

```
int sens;
sens = RSens ( );
```

È anche possibile utilizzare l'istruzione *return* senza argomenti, in qualunque punto di una funzione di tipo void, per uscire dalla funzione stessa senza restituire nulla. Sono valide pertanto entrambe le forme:

```
return;
return value;
```

L'istruzione *return* senza argomenti è utile per esempio per testare una sezione di codice all'interno di una funzione, senza dover commentare il resto del codice.

```
void loop ( ) {
 // zona di codice da testare
 return;
 // resto del codice che potrebbe contenere altri errori
}
```

Più in generale, è utile ricorrere alle funzioni per isolare una parte di codice che può essere richiamata più volte nel programma principale, in modo da evitare la scrittura ripetuta del codice e rendere il programma più compatto e meglio leggibile.

2 FUNZIONI DI LIBRERIA

Arduino dispone di una libreria base ricca di funzioni, in particolare di funzioni matematiche, funzioni trigonometriche e funzioni avanzate di ingresso/uscita (I/O).

2.1 FUNZIONI MATEMATICHE

Le principali funzioni matematiche disponibili nella libreria sono proposte di seguito.

```
min(x, y);
```

restituisce il minore tra i due numeri passati, qualsiasi sia il tipo, per esempio
`pippo = min (pippo, 100);`
garantisce che la variabile pippo sia al massimo 100;

```
max(x, y);
```

restituisce il maggiore tra i due numeri passati, cualsiasi sia il tipo, per esempio
`pippo = max (pippo, 20);`
garantisce che pippo non sia mai minore di 20;

```
abs(x);
```

restituisce il valore assoluto di x, per esempio, se x = –5
`pippo= abs(x);`
assegna a pippo il valore 5;

```
constrain(x, a, b);
```

restituisce x se è compresa tra i limiti a e b, altrimenti uno dei limiti, per esempio
`pippo = constrain(pippo, 10, 150);`
limita il range di pippo tra 10 e 150;

```
map(value, fromLow, fromHigh, toLow, toHigh);
```

converte di scala un valore, per esempio
`temp = map (analogRead(0), 0, 1023, 0, 100);`
converte il valore letto dall'ADC nella scala da 0 a 100;

```
double pow (float base, float exponent);
```

esegue l'elevamento a potenza della base, per esempio

```
pluto = pow (5,2);
```

assegna alla variabile pluto, di tipo double, il valore $5^2 = 25$;

```
double sqrt(x);
```

restituisce la radice quadrata di x, per esempio, se x = 25

```
pluto = sqrt (x);
```

assegna alla variabile pluto, di tipo double, il valore 5.

2.2 FUNZIONI TRIGONOMETRICHE

Le principali funzioni trigonometriche disponibili nella libreria lavorano con l'argomento in radianti. Di seguito sono riportate le più comuni.

```
double sin (float xrad);
```

per esempio, se xrad = 1

```
pluto = sin (xrad);
```

assegna alla variabile pluto, di tipo double, il valore 0,84.

```
double cos (float rad);
```

per esempio, se xrad = 1

```
pluto = cos (xrad);
```

assegna alla variabile pluto, di tipo double, il valore 0,54.

```
double tan (float rad);
```

per esempio, se xrad = 1

```
pluto = tan (xrad);
```

assegna alla variabile pluto, di tipo double, il valore 1,557

2.3 FUNZIONI DI I/O

Le principali funzioni di ingresso/uscita disponibili nella libreria sono *pulseIn* e *shiftOut*.

```
unsigned long pulseIn (pin, value);
```

attende al massimo 1 s che il livello del segnale presente sul pin di ingresso indicato vada alto (*value* = *HIGH*) o basso (*value* = *LOW*) e ne misura la durata in µs (unsigned long).

Per esempio, se l'ingresso 7 è stato definito INPUT,

```
unsigned long duration = pulseIn (7, HIGH);
```

attende al massimo 1 s che il livello del segnale presente sul *pin* 7 vada alto, ne misura la durata in µs e la assegna alla variabile *duration*.

```
shiftOut (dataPin, clockPin, bitOrder, byte value);
```

emette gli 8 bit (byte) di *value* sul pin n° *dataPin*, un bit per volta, a partire dal bit meno significativo (*bitOrder* = *LSBFIRST*) o dal più significativo (*bitOrder* = *MSBFIRST*), accompagnati da altrettanti fronti sul pin n° *clockPin*, ad indicare ogni volta la disponibilità di un bit valido.

Poiché la funzione *shiftOut* emette solo 8 bit per volta, se la variabile è più lunga vanno utilizzate più chiamate.

Per esempio, se il dato è un numero a 16 bit da emettere a partire dal bit meno significativo, basta chiamare

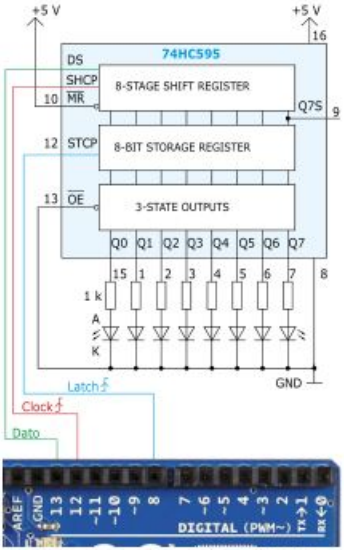
```
shiftOut (pindata, pinclock, LSBFIRST, data);
shiftOut (pindata, pinclock, LSBFIRST, (data >> 8));
```

Programma d'esempio - Shift register 74HC595

L'algoritmo che segue emette, impegnando soli 3 pin, un numero binario a 8 bit, di valore progressivo, sugli 8 LED in uscita a un integrato esterno shift register 74HC595.

• *Disposizione dei pin dell'integrato 74HC595.*

Nel programma che segue, il valore progressivo a 8 bit da emettere è contenuto nella variabile *j*, incrementata ogni 1 s all'interno di un ciclo for. Mediante la funzione *shiftOut* si emettono gli 8 bit del dato sul pin 13 (*dataPin*) accompagnati da altrettanti colpi di clock sul pin 12 (*clockPin*), riempiendo lo shift register interno all'integrato 74HC595. Successivamente, il valore acquisito dallo shift va trasferito al registro che pilota le uscite (*Storage Register*), mediante un fronte di salita sul pin 8 (*latchPin*).



• *Interfacciamento tra Arduino e 74HC595.*

```
#define latchPin 8 // STCP latch
#define clockPin 12 // SHCP clock
#define dataPin 13 // DS data serial
void setup ( ) {
 pinMode (latchPin, OUTPUT);
 pinMode (clockPin, OUTPUT);
 pinMode (dataPin, OUTPUT);
}
void loop ( ) {
 for (int j = 0; j < 256; j++) {
  digitalWrite (latchPin, LOW); // abbassa latch
  shiftOut (dataPin, clockPin, LSBFIRST, j);
  digitalWrite (latchPin, HIGH); // rialza latch
  delay (1000);
 }
}
```

Il pin OE (*Output Enable*) dell'integrato, attivo basso, governa, a sua volta, il collegamento tra il dato immagazzinato nello Storage Register e gli otto pin di uscita a cui sono connessi i LED; va perciò posto a massa.

3 FUNZIONE MILLIS

La chiamata della funzione di libreria

```
unsigned long millis ( )
```

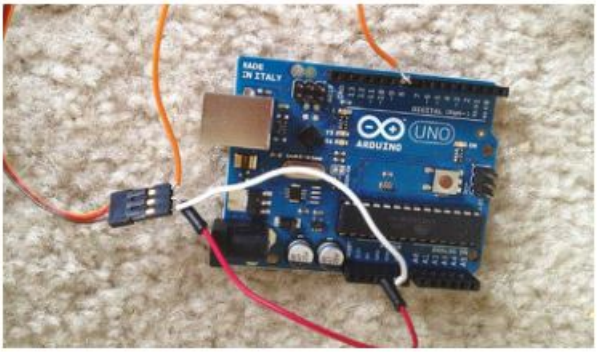
restituisce il numero di millisecondi trascorsi dall'avvio del programma, sotto forma

di intero a 32 bit senza segno. Secondo le specifiche di Arduino, tale numero trabocca e riparte da zero circa ogni 9 ore e 32 minuti.
Trascurando il ritardo di esecuzione delle istruzioni, il programma che segue emette ogni secondo il valore del tempo trascorso a partire dall'avvio.

```
/* prova millisec()*/
unsigned long time;
void setup(){
 Serial.begin(9600);}
void loop(){
 Serial.print("Time: ");
 time = millis();
 Serial.println(time);
 delay(1000);
}
```

La funzione *millis()* può risultare perciò una funzione utile per valutare la **durata di esecuzione** di pezzi del programma.
Il programma che segue permette di confrontare la durata di esecuzione di 10.000 somme tra interi rispetto a una pari elaborazione tra float.

```
/* confronto durata int float */
unsigned int y, pippo;
unsigned long startTime;
float pluto;
void setup(){Serial.begin(9600);}
void loop() {
 startTime = millis();
 for (y=1; y<10000; y++) {
  pippo +=20;}
 Serial.print("Int: ");
 Serial.println(millis() - startTime);
 startTime = millis();
 for (y = 1; y < 10000; y++) {
  pluto +=20;}
 Serial.print("float: ");
 Serial.println(millis() - startTime);
 delay(1000);
}
```



4 FUNZIONE RANDOM

Un generatore di numeri casuali (*random*) è un algoritmo in grado di produrre una sequenza di numeri casuali, ogni volta diversa dalla precedente.
Un generatore di numeri pseudo-casuali, invece, produce una sequenza che è sempre la medesima ad ogni chiamata. In Arduino, la funzione di libreria

```
long random (max)
```

restituisce un numero pseudo-casuale compreso tra 0 e max – 1, mentre la funzione

```
long random (min, max)
```

restituisce un numero pseudo-casuale compreso tra gli estremi min e max – 1; il valore minimo, quindi, è incluso tra i risultati della funzione, mentre il valore massimo non è compreso.
Se è importante che una sequenza di valori prodotta dalla funzione *random ()* differisca ogni volta, occorre anteporre alla funzione *random ()* la chiamata della funzione

```
randomSeed (long int seed)
```

con un argomento abbastanza imprevedibile, come, per esempio, il valore del rumore acquisito da un ingresso analogico sconnesso, rilevato mediante una *analogRead ()*.
Per esempio, il programma che segue emette una sequenza di numeri casuali compresi tra 0 e 299, utilizzando, ad ogni avvio, la lettura dell'ingresso analogico A0 (supposto sconnesso) come seme (*seed*) di generazione.

```
long randNumber;
void setup( ){
 Serial.begin (9600);
 randomSeed (analogRead(0));
}
void loop( ){
 randNumber = random (300);
 Serial.println (randNumber);
 delay (50);
}
```

Se si vuole, invece, che la sequenza si ripeta ogni volta esattamente identica, basta anteporre alla funzione *random ()* la chiamata della *randomSeed ()* con un numero fisso.

5 VARIABILI PUBBLICHE, PRIVATE E STATICHE

Le **variabili** definite in testa al programma principale (esternamente al loop) sono dette **pubbliche**, perché risultano visibili e modificabili da tutte le funzioni presenti nel programma.
Le variabili definite all'interno di una funzione sono, invece, **private**, cioè sono visibili solo all'interno della funzione stessa e da nessun altro; sono create e distrutte a ogni chiamata, perciò due funzioni possono avere variabili private con il medesimo nome, senza che queste interferiscano tra loro.
Se si vuole che una variabile conservi il suo valore tra una chiamata e l'altra, va dichiarata **static**. Una variabile statica è creata e inizializzata solo quando la funzione è chiamata per la prima volta, conserva ogni volta il suo valore tra due chiamate successive e non è né visibile, né leggibile, né alterabile dalle altre funzioni.
L'algoritmo che segue contiene la funzione *sposta_random* che genera una sequenza pseudo-casuale di numeri all'interno di due estremi (*LowRange*, *HighRange*) e con passo massimo definito (*Stepsize*).

Per non perdere l'ultimo valore generato, la funzione utilizza la variabile statica **current**.

```
/* random */
#define LowRange -20
#define HighRange 20
#define Stepsize 5
int valore;
void setup ( ) {
 Serial.begin(9600); }
void loop ( ) {
 valore = sposta_random (Stepsize);
 Serial.println (valore);
 delay(1000);
}
int sposta_random (int moveSize){
 static int current; // variable statica
 current = current + (random(-moveSize, moveSize + 1));
 if (current < LowRange){ // controlla i limiti
  current = current + (LowRange - current); // recupera in +
 }
 else if (current > HighRange){
  current = current - (current - HighRange); // recupera in -
 }
 return current;
}
```

L'attributo **const** (costante) definisce, infine, un'entità di sola lettura (*read-only*), quindi non modificabile. Ciò significa che la variabile può essere usata come qualunque altra variabile dello stesso tipo, ma non può essere modificata.
Se erroneamente si tenta di assegnarle un valore, il compilatore segnala l'errore.
Equivale, in pratica, alla direttiva #define.
Si veda lo spezzone di programma proposto di seguito.

```
const float pi = 3.14;
float x;
// ....
x = pi * 2;
pi = 7;
/* istruzione illegale perché si tenta di modificare il valore di
una costante */
```

6 STRINGHE E ARRAY

Molto spesso i dati da elaborare sono organizzati in insiemi ordinati, formati da più elementi disposti in sequenza, a una o più dimensioni, individuabili mediante puntatori numerici.
Gli insiemi principali sono le stringhe e gli array.
Una **stringa** è un insieme ordinato di elementi char, terminato con il carattere **null** (\0).
Ciascun elemento occupa un byte di memoria.
Il terminatore *null* è indispensabile per interagire con le funzioni di manipolazione delle stringhe, come, per esempio, *Serial.print ()*.
Tutte le dichiarazioni sottostanti creano stringhe valide.

```
char Str1 [15]; // stringa non inizializzata
```

```
char Str2 [8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o'};
/* il carattere null è aggiunto dal compilatore */
```

```
char Str3 [8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o', '\0'};

char Str4 [ ] = "arduino"; // stringa di 8 caratteri, compreso il null

char Str5 [8] = "arduino"; // stringa di 8 caratteri

char Str6 [15] = "arduino"; // stringa con spazio extra riservato
```

Un **array** è un insieme ordinato di elementi omogenei (del medesimo tipo), con dimensione (*size*) specificata.
Gli array possono essere a una o più dimensioni.
Per esempio, l'istruzione

```
int a [10] [5]
```

definisce un array di interi a due dimensioni, formato da 10 righe e 5 colonne, nel quale possono essere ospitati fino a 50 numeri irteri.
I più utilizzati sono gli array a una dimensione, detti anche **vettori**.
Tutte le dichiarazioni sottostanti, per esempio, creano vettori validi di 6 elementi.

```
int myInts [6]; // vettore di 6 elementi, non inizializzato

int myPins[ ] = {2, 4, 8, 3, 6, 7};
// la dimensione dell'array è calcolata dal compilatore

int myVals [6] = {2, 4, -8, 3, 2, 7}; // vettore inizializzato

char message[6] = "hello";
// trattandosi di char esiste anche il carattere null di chiusura
```

Altri esempi di definizioni di array monodimensionali, sono le seguenti:

```
char bufftxa [ ];
// array vuoto di caratteri, di lunghezza definita successivamente

int num [10];
/* array vuoto di 10 elementi interi, da num [0] a num [9], occupa
 10 · 2 = 20 byte di RAM */

int num [3] = { 45, 12, 1257}; // array inizializzato
```

Gli elementi interni di un array sono accessibili mediante un **indice numerico** che inizia da 0 e termina a size – 1.
Per esempio, osservando l'array *myVals* precedente si ha che myVals [0] contiene 2, myVals [1] contiene 4, e così via.
È possibile anche modificare il valore contenuto negli elementi del vettore, attraverso una **nuova assegnazione**. Per esempio, l'assegnazione myVals [0] = 9 modifica il valore contenuto nel primo elemento (da 2 diventa 9).
L'indice numerico non è, però, da usare in modo improprio. L'assegnazione x = myVals [6], per esempio, pur essendo impropria perché myVals ha solo 6 elementi con indice da 0 a 5, non è considerata errata dal compilatore, ma è assolutamente da evitare perché restituisce un valore del tutto ignoto, ovvero l'informazione contenuta nella cella di memoria successiva al vettore myVals stesso.
Gli array sono spesso elaborati mediante **cicli for**, che utilizzano il contatore come indice stesso dell'array.
Il ciclo for proposto, per esempio, emette in modo ordinato gli elementi di un array sulla porta seriale.

```
int i;
for (i = 0; i < 5; i ++) {
 Serial.println (myPins[i]);
}
```

L'esempio successivo, invece, aggiorna ogni 1 s il valor medio (*average*) relativo agli ultimi 5 campioni di un sensore, utilizzando un ciclo for per elaborare l'array dei campioni.

```
/* media */
#define sensPin A3
int sensVal [5]; // crea un array per i campioni
int i, j; // variabili per i contatori
long average; // variabile per memorizzare la media
void setup ( ) {
 Serial.begin(9600); }
void loop( ){
 // acquisisce il nuovo campione
 sensVal [(i++) % 5] = analogRead(sensPin);
 average = 0;
 for (j=0; j<5; j++){
  average += sensVal[j]; // sommare i campioni
 }
 average = average / 5; // dividere per il totale
 Serial.println(average);
 delay(1000);
}
```

7 PUNTATORI

I puntatori sono variabili utilissime, perché, soprattutto nella manipolazione di grosse quantità di dati, consentono di semplificare il codice e di ridurre l'occupazione di memoria. A differenza di un indice, che è una variabile numerica, un puntatore è una variabile che **conserva l'indirizzo** dell'elemento puntato.
Per esempio

```
char *p
```

definisce un puntatore (indirizzatore) vuoto, di nome *p*, in grado di indirizzare variabili o array contenenti caratteri.
È obbligatorio specificare il **tipo di puntatore**, perché in caso di tipo char, eseguendo p++ l'indirizzo contenuto in *p* avanzerebbe di 1 byte (1 char occupa 1 sola cella di memoria), mentre in caso di tipo float (float *p), p++ farebbe avanzare l'indirizzo contenuto in *p* di 4 byte di memoria.
È essenziale comprendere questo aspetto per indirizzare correttamente i dati di memoria, anche se tutto ciò avviene senza che il programmatore se ne accorga.
Se *p* è un puntatore a elementi char, l'assegnazione

```
x = *p;
```

assegna alla variabile x **il contenuto dell'elemento** (char) **puntato** da *p*.
Altri esempi:

```
*PITR = 0x00;
```

azzera il contenuto della variabile puntata da PITR;

```
*(PITR+1) = 0x00;
```

azzera il contenuto della variabile successiva.
È importante sottolineare che tutti i **nomi di array** sono implicitamente dei puntatori, perché contengono l'indirizzo di memoria del primo elemento dell'array.
Per esempio in

```
char *str = "stringa";
```

il puntatore *str* punta il primo elemento dell'array di 8 elementi *'s' 't' 'r' 'i' 'n' 'g' 'a' '\0'* memorizzato in Flash.

Nell'algoritmo che segue, per esempio, si usano due puntatori (*p*, *s*) per calcolare il numero di caratteri validi presenti in una stringa.
Il puntatore *s* viene posizionato sul primo carattere dell'array *stringa* e vi rimane.
Il puntatore *p*, invece, parte allineato sul primo carattere all'inizio del ciclo for e si incrementa ad ogni ciclo.
Poiché il ciclo for termina quando il carattere purtato da *p* è nullo, la distanza tra i due puntatori al termine, ovvero p-s, sarà pari alla lunghezza della stringa.

```
char stringa [80] = "Pippo cammina random";
char *p; char *s; // p ed s sono due puntatori a carattere
s = stringa; // allinea s sul primo carattere di stringa
for (p=stringa; *p; p++)
Serial.println (p-s);
```

In alcune applicazioni, per esempio quando si deve interagire con un LCD, è conveniente dimensionare un **array di stringhe** o un **array di puntatori**.
Poiché tutti i nomi di array sono puntatori, nell'istruzione

```
char *giose[7] = { "Lu.","Ma.","Me.","Gi.","Ve.","Sa.","Do." };
```

l'asterisco che segue la definizione di tipo char (char*) indica che si tratta di un array di puntatori.
L'algoritmo che segue emette diverse stringhe utilizzando un array di puntatori per selezionarle in memoria.

```
char *myStrings[ ]={"string 1", "string 2", "string 3", "string 4",
"string 5"};
void setup ( ){
 Serial.begin (9600); }
void loop ( ){
 for (int i = 0; i < 5; i++){
  Serial.println(myStrings[i]);
  delay(500);
 }
}
```

7.1 DEREFERENCE E REFERENCE

L'operatore * (*dereference*), anteposto a un indirizzo, restituisce il **contenuto** della variabile indirizzata, mentre l'operatore & (*reference*), anteposto a una variabile, restituisce l'**indirizzo** di memoria di quella variabile.
Per esempio:

```
char cc;
```

indica che cc è una variabile di tipo char creata in RAM;

```
char *puntcc;
```

indica che puntcc è un puntatore vuoto di elementi char;

```
puntcc = &cc;
```

indica che puntcc contiene l'indirizzo di memoria della variabile cc;

```
xx = *puntcc;
```

indica che xx vale il contenuto della variabile puntata da puntcc, equivalente alla `xx = cc;`.

8 STRUTTURE

Per organizzare al meglio un archivio di dati anche non omogenei tra loro, il programmatore può crearsi una **struttura** di variabili su misura, utilizzando la parola chiave **struct**.

Per esempio, la definizione che segue

```
struct tiprec {
 // definisce il nome del tipo di struttura
 int giorno; // progressivo del giorno dalla schiusa delle uova
 float presenti; // n° polli presenti
 float grammicapo; // consumo unitario di mangime
 int ltacqua; // consumo d'acqua giornaliero
};
```

è stata utilizzata da un programmatore per definire un tipo di elemento (*tiprec*) composto da 4 variabili (2 intere e 2 float), che gli era necessario per registrare in modo ordinato la progressione dei consumi e della mortalità in un allevamento di polli. Un elemento di tipo *tiprec* impegna quindi 2+4+4+2 = 12 byte.
La definizione

```
struct tiprec archivio[100];
```

crea quindi un archivio di 100 elementi di tipo *tiprec* (1.200 byte), di nome *archivio*.
Per interagire con la singola variabile di un elemento di una struttura bisogna specificarne il percorso, utilizzando il punto di separazione.
Per esempio:

```
oggipresenti = archivio[num_giorni].presenti;
oggirichiesto = archivio[num_giorni].grammicapo * oggipresenti / 1000.0;
oggiacqua = archivio[num_giorni].ltacqua * coefacq;
```

Anche con le strutture, è possibile operare tramite puntatori.
Per esempio:

```
struct tiprec *puntarch;
```

crea un puntatore a strutture di tipo *tiprec*;

```
puntarch= &archivio;
```

allinea il puntatore sulla struttura archivio.
La singola variabile di una struttura è quindi raggiungibile o direttamente

```
oggipresenti = archivio[num_giorni].presenti;
```

oppure tramite il puntatore

```
oggipresenti = (puntarch + num_giorni) -> presenti
```

9 PROGMEN

Di norma, tutte le variabili sono create in RAM ma, essendo tale risorsa limitata, è preferibile dichiarare le variabili fisse ingombranti, quali gli array di stringhe per un LCD, nella memoria programma (Flash), che è ben più estesa.
Per creare una variabile o un array in Flash anziché in RAM, basta aggiungere la dichiarazione **PROGMEM** accanto alla definizione di tipo. Poiché PROGMEM appartiene alla libreria **pgmspace.h**, questa va inclusa nel programma utilizzando la direttiva

```
#include <avr/pgmspace.h>
```

da porre in testa al programma stesso.

TAB. 3 - TIPI DI VARIABILI PROGMEM DICHIARABILI IN ARDUINO

TIPO	BYTE OCCUPATI
char, byte, int8_t, uint8_t	1
int16_t, uint16_t	2
int32_t, uint32_t	4
Float	4

Per evitare confusioni ed errori, i tipi ammessi da PROGMEM hanno definizioni specifiche, leggermente differenti dalle variabili tradizionali (tabella 3). Anche l'operazione di lettura di una variabile dalla Flash richiede una funzione specifica, compresa nella libreria pgmspace.h:

```
pgm_read_byte (indirizzo); // legge 1 byte
pgm_read_word (indirizzo); // legge 2 byte
pgm_read_dword (indirizzo); // legge 4 byte
pgm_read_float (indirizzo); // legge un float
pgm_read_ptr (indirizzo); // legge un puntatore, 2 byte
```

Il codice che segue, per esempio, legge interi a 16 bit e byte residenti in Flash e li assegna ad altrettante variabili in RAM da stampare.

```
#include <avr/pgmspace.h>
PROGMEM const uint16_t Set[ ] = { 65000, 32796, 16843, 10, 11234};
PROGMEM const char Message[ ]= "I AM PREDATOR";
unsigned int displayInt;
int k;
char myChar;
void setup() {
 Serial.begin(9600);
 k = 0;
}
void loop() {
 for (k=0;k<5;k++){
  displayInt = pgm_read_word (Set + k); // legge un int a 2 byte
  Serial.println( displayInt );
  delay( 500 );
 }
 for (k=0;k<13;k++){
  myChar = pgm_read_byte (Message + k); // legge 1 byte
  Serial.print( myChar );
 }
 Serial.println ();
 delay( 500 );
}
```

L'esempio successivo, invece, memorizza in Flash un array di stringhe, tiene in RAM la relativa tabella degli indirizzi e usa la funzione **strcpy_P** per copiare una stringa dalla memoria Flash ad un vettore di nome *buffer* presente in RAM.

```
#include <avr/pgmspace.h>
PROGMEM const char string_0 [ ] = "String 0";
PROGMEM const char string_1 [ ] = "String 1";
PROGMEM const char string_2 [ ] = "String 2";
PROGMEM const char string_3 [ ] = "String 3";
PROGMEM const char string_4 [ ] = "String 4";
PROGMEM const char string_5 [ ] = "String 5";
const char* string_table[ ] = {string_0,string_1,string_2,string_3,
string_4,string_5};
char buffer[30];
/* buffer RAM più grande della stringa più lunga da memorizzare */
void setup ( ) {
 Serial.begin(9600);}
void loop ( ){
 for (int i = 0; i < 6; i++) {
  strcpy_P (buffer, (char*) string_table[i]);
  Serial.println( buffer );
  delay( 1000 );
 }
}
```

10 ALGORITMI PER AUTOMATISMI

La maggior parte degli automatismi esegue lavorazioni che ripetono cicli di lavoro composti da fasi, elaborate in sequenza o in alternativa tra loro.
Il passaggio da una fase alla successiva avviene solo se sono soddisfatte specifiche condizioni di transizione legate al verificarsi di un tempo (macchina funzione del tempo) o di un evento meccanico, come il raggiungimento di un finecorsa o la pressione di un tasto (macchina funzione della corsa).
Quasi sempre, accanto alle **sequenze cicliche di lavorazione**, l'algoritmo di gestione deve elaborare anche alcune **funzioni combinatorie sempre attive**, quali la lettura dello **stop immediato** e l'acquisizione del comando di **stop a fine ciclo**, la cui attivazione va memorizzata per essere testata alla fine del ciclo allo scopo di bloccarne la ripresa automatica.
Le funzioni combinatorie sono realizzate impiegando solo istruzioni di selezione, quali *if-else*, e sono di esecuzione veloce.
Per esempio:

```
void pressostato ( ) {
 if ( ... ) { ... }
 else ...;
}
```

Per fare in modo che si possano gestire in contemporanea sia le funzioni combinatorie sia le cicliche, l'algoritmo sequenziale non ricorre a istruzioni di attesa delle condizioni di transizione, dalla durata imprevedibile, come

```
while (sensore == 0);
```

ma utilizza una gestione entra-esci veloce, che memorizza la fase in corso ed elabora la sequenza delle fasi, sfruttando la struttura *switch-case*, come schematizzato nella funzione *ciclo1* che segue.

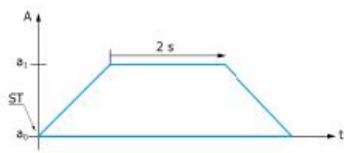
```
void ciclo1 ( ) {
 switch ( faseCiclo1) {
  case 0 : Vs=1;
   // azioni associate alla fase 0
   if (condiz. di transiz. alla fase X) {faseCiclo1 = X; break;}
   if (condiz. di transiz. alla fase Y) {faseCiclo1 = Y; break;}
   else break; // uscita con rientro successivo nella medesima fase
  case 1 : …
   // azioni associate alla fase n° 1
   if ( ... ) { faseCiclo1 = ... ; break;}
   else break; // uscita con rientro successivo nella medesima fase
  case 2 : ...
  case X: ...
  case Y: ...
 }
}
```

Operando in questo modo, il loop principale può gestire in contemporanea e alla massima velocità più funzioni combinatorie e più cicliche, come schematizzato nel programma che segue.

```
void setup( ) {
 faseCiclo1 = 0; faseCiclo2 = 0;
}
void loop ( ) {
 pressostato ( ); // 2 funzioni combinatorie sempre attive
 tast_vis ( );
 ciclo1 ( ); // 2 cicliche
 ciclo2 ( );
 ...
}
```

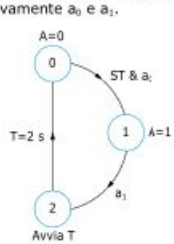
Esempio di automatismo

Si supponga di dover azionare un cilindro pneumatico (A), secondo il diagramma proposto.



• *Azionamento temporizzato.*

Premendo il pulsante di avvio (ST), il cilindro A esce, completa la sua corsa, attende 2 s e rientra.
Il cilindro ha due stati: lo stato con l'asta che rientra (A = 0) e lo stato con l'asta in uscita (A = 1).
Dispone, inoltre, di una coppia di finecorsa, attivi alti, allocati nelle due posizioni estreme della corsa, rispettivamente a_0 e a_1.



• *Diagramma degli stati dell'azionamento.*

Rifacendosi al diagramma degli stati, la sequenza può essere scomposta, come segue:

- a riposo, il cilindro è in posizione rientrata (A = 0);
- con il finecorsa a_0 chiuso (a_0 = 1), premendo il pulsante ST il cilindro A esce (A = 1);
- quando si chiude il finecorsa a_1 (a_1 = 1), inizia il conteggio dei 2 s;
- allo scadere dei 2 s, il cilindro A rientra (A = 0).

Il sistema richiede, quindi, 3 ingressi (ST, a_0, a_1) e 1 uscita (A), da simulare con 3 tasti e 1 LED.

Il programma di gestione del ciclo può essere strutturato nel modo seguente:

```
#define A 13
…
long time;
int fase = 0;
void setup ( ) {
 pinMode (A, OUTPUT);
 …
}
void loop ( ) {
 switch ( fase) {
  case 0 : digitalWrite (A, LOW);
   if (ST==1 && a0==1) {
   fase = 1; break;}
   else break;
  case 1 : digitalWrite (A, HIGH);
   if (a1==1) {
    fase = 1; time = millis();
    break;}
   else break;
  case 2 :
   if ((abs(millis() - time) >= 2000) {
    fase = 0; break;}
   else break;
 }
}
```

Se il sistema prevede anche la presenza di un carter di sicurezza, che faccia rientrare immediatamente il cilindro in caso di apertura, il test dell'ingresso relativo deve essere sempre attivo (attivo a ogni ciclo).

```
void loop ( ) {
 if (Carter ==0) fase =0;
 switch ( fase) {
  ...
 }
}
```

11 INTERRUPT

Interrupt è una modalità operativa che permette ai dispositivi esterni di interrompere momentaneamente l'esecuzione del programma principale, per eseguire uno specifico sottoprogramma di servizio all'interruzione stessa.
Una volta terminata tale routine, il sistema riprende l'esecuzione del programma dal punto esatto in cui si trovava al momento dell'interruzione.
I dispositivi esterni abilitati a operare in interrupt vengono quindi serviti senza l'intervento del programma principale.
Per definizione, un sistema **in tempo reale** è un sistema che non perde informazioni. Poiché un ingresso gestito dal programma principale viene testato con periodicità pari al tempo di scansione del loop, un evento di durata minore rischierebbe di non essere avvertito e quindi verrebbe perso.
Abilitando, invece, un ingresso a operare in interrupt, il servizio relativo agisce immediatamente, garantendo al sistema di non perdere informazioni, e di essere cioè un sistema in tempo reale. L'evento che scatena un interrupt può essere semplicemente il fronte di un segnale, di salita, di discesa o entrambi (interrupt su evento); in altri casi può essere lo scadere di un timer (interrupt a tempo).

Il conteggio degli impulsi di un encoder rotativo, l'acquisizione di impulsi ottici nell'infrarosso e la gestione degli allarmi immediati, quale la caduta dell'alimentazione principale, sono funzioni normalmente gestite in interrupt.
Arduino mette a disposizione i digital pin 2 e 3 per innescare rispettivamente gli interrupt numero 0 e 1. L'istruzione da utilizzare è

```
attachInterrupt (n° interrupt, funzione, evento)
```

i cui parametri sono nell'ordine:

- il numero di interrupt (0 o 1);
- il nome della funzione da eseguire ogni volta che si verifica un evento sul pin corrispondente, detta **interrupt service routine**;
- il tipo di evento scatenante la chiamata, scegliendo tra le quattro costanti predefinite indicate in tabella 4.

La funzione da chiamare in interrupt non deve contenere parametri e non deve ritornare alcun valore, cioè deve essere del tipo *void function ()*. Inoltre, se non si vuole rischiare di perdere eventuali dati in ricezione sulla linea seriale, l'esecuzione della funzione deve durare il meno possibile (è vietato utilizzare la *delay ()*).
Ogni variabile modificata all'interno della funzione dovrebbe essere dichiarata **volatile**, una direttiva rivolta al compilatore che ne determini la creazione in RAM e non in un registro.
In questo modo si è sicuri che risulta accessibile senza alterazioni da più processi concorrenti, quali per esempio loop e interrupt.
L'esempio che segue conta il numero di rimbalzi che si verificano a ogni pressione o rilascio del tasto connesso sul pin 2 (*interrupt 0*).

TAB. 4 - TIPI DI EVENTI SCATENANTI UN INTERRUPT

TIPO DI EVENTO	MODALITÀ DI ATTIVAZIONE
LOW	Fintanto che il pin è basso
CHANGE	Ogni volta che il pin cambia il livello logico
RISING	Solo su fronte di salita
FALLING	Solo su fronte di discesa

• *Tasto connesso sul pin 2.*

```
/* interrupt */
#define tasto 2
volatile int conta = 0;
void setup ( ) {
 Serial.begin (9600);
 Serial.flush ( );
 pinMode (tasto, INPUT);
 digitalWrite (tasto, HIGH);
 attachInterrupt (0, blink, CHANGE);
}
void loop( ) {
 if (conta){
 delay (300);
 Serial.println (conta);
 conta = 0;
 }
}
void blink ( ) {
 conta++;
}
```

L'istruzione

```
detachInterrupt (n° interrupt)
```

disattiva la funzione interrupt indicata (0 o 1), precedentemente attivata.

Applicazione di interrupt

Per comprendere il ruolo delle funzioni di interrupt, si osservi l'esempio di un'applicazione che richiede una gestione in interrupt. Su un nastro trasportatore, che scorre alla velocità costante di 1 m/s, passano in continuazione pacchi dei quali si vuole misurare la lunghezza con precisione ±0,5 cm, mediante misura di tempo dopo l'interruzione della fotocellula. Le risorse da mettere in campo sono un ingresso di interrupt per la fotocellula, attivo su entrambi i fronti del segnale di buio, e un contatore azionato da un timer con cadenza

$$\Delta t = \frac{0,5\,\text{cm}}{1\,\text{m/s}} = 5\,\text{ms}$$

Il contatore è azzerato durante la routine di interrupt relativa al fronte iniziale della fotocellula e va letto nella routine relativa al fronte di fine-pacco.
Se N è il valore letto, la misura di lunghezza varrà:

$$L = N \cdot 0,5\,\text{cm} \pm 0,5\,\text{cm}$$

12 TIMER INTERRUPT

Nella gestione degli automatismi, è spesso richiesta l'attivazione di una funzione a **intervalli regolari**, senza l'intervento del main, o meglio, mentre il programma principale sta eseguendo regolarmente le sue funzioni.
Per fare ciò, è necessario disporre di un timer, abilitato a chiamare interrupt ad ogni *overflow* del valore impostato al suo interno.
Ciò è possibile, per esempio, estendendo la libreria standard di Arduino con MsTimer2 (http://www.arduino.cc/playground/Main/MsTimer2), da installare (copiare) nella cartella {arduino-path}/libraries/
MsTimer2 è una libreria che impegna il timer interno Timer2 in interrupt, con risoluzione 1 ms.
Le funzioni disponibili sono:

- `MsTimer2::set (unsigned long ms, void (*f)())`
  setta l'intervallo, espresso in ms, della chiamata periodica della funzione *f* indicata; trattandosi di una funzione di interrupt, non deve avere parametri di chiamata;
- `MsTimer2::start()`
  abilita il timer interrupt;
- `MsTimer2::stop()`
  disabilita il timer interrupt.

L'esempio che segue innesca ogni 0,5 s la funzione *flash* che rovescia lo stato del LED connesso al pin 13.

```
char vuoto; // per includere correttamente la libreria
#include <MsTimer2.h>
void flash() {
 static boolean output = HIGH;
 digitalWrite(13, output);
 output = !output;
}
void setup (){
 pinMode(13, OUTPUT);
 MsTimer2::set(500, flash); // periodo di chiamata 500ms
 MsTimer2::start();
}
void loop () {}
```