Usare Arduino

1 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:
pinMode (13, OUTPUT);

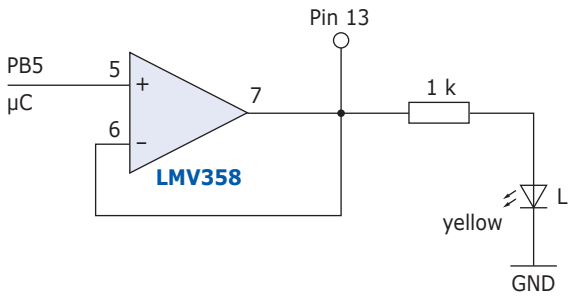

Fig. 1. *Hardware relativo al LED L.*

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.

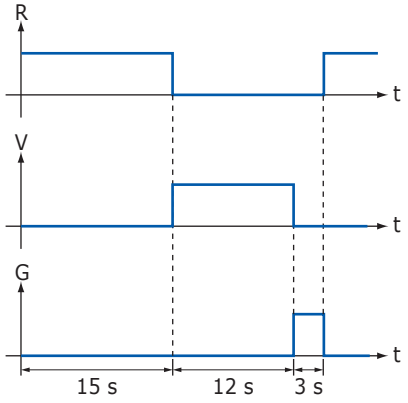

Fig. 2. *Temporizzazione di un semaforo.*

Utilizzando il LED giallo sulla scheda, connesso al pin 13, basta aggiungere due LED esterni sui pin 12 (Rosso) e 11 (Verde).

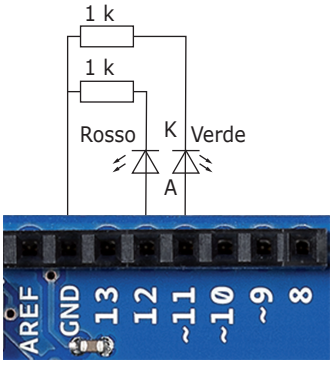

Fig. 3. *Connessione dei LED esterni, rosso e verde.*

```
/* 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
}
```

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

2 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**);

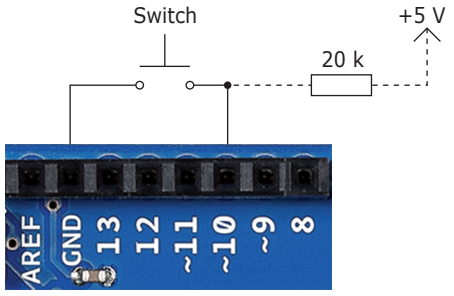

Fig. 4. *Connessione di uno switch attivo basso.*

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.

Fig. 5. *Tilt sensor.*

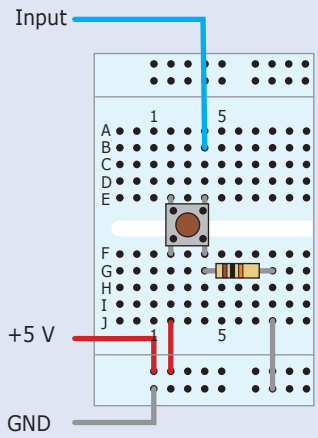

Fig. 6. *Input da switch, attivo alto, con resistenza di pull-down.*

3 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).

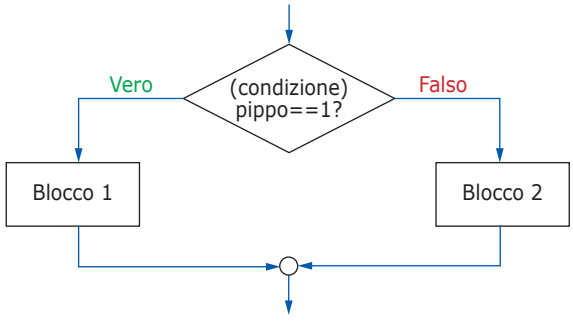

Fig. 7. *Selezione completa.*

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

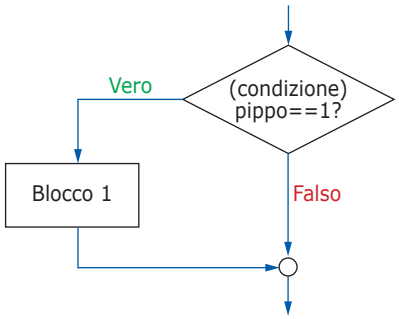

Fig. 8. *Selezione semplice.*

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
}
```

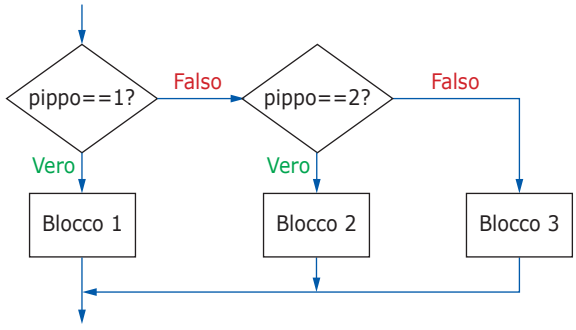

Fig. 9. *Selezione multipla.*

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

3.1 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)
```

3.2 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) e (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 ;
```

equivalente alla struttura:

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

Per esempio:

```
#define min (x,y)
 ((x) < (y)) ? (x) : (y) // definisce la fun-
 zione min che restituisce il numero minore
#define max (x,y)
 ((x) > (y)) ? (x) : (y) // definisce la fun-
 zione max che restituisce il numero maggiore
#define Dec (x)
 ((x > '9') ? (x - 'A' + 10) : (x - '0')) //
 converte una cifra esadecimale ASCII in de-
 cimale
#define Asc (x)
 ((x > 9) ? (x - 10 + 'A') : (x + '0')) // con-
 verte una cifra esadecimale in ASCII
```

4 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 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. 1 - 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 low/high.
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 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**, corrispondente a 257 in decimale.

5 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).

5.1 Operatori logici relazionali

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 23 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.
```

6 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 distinguere 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;
```

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.

7 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*.

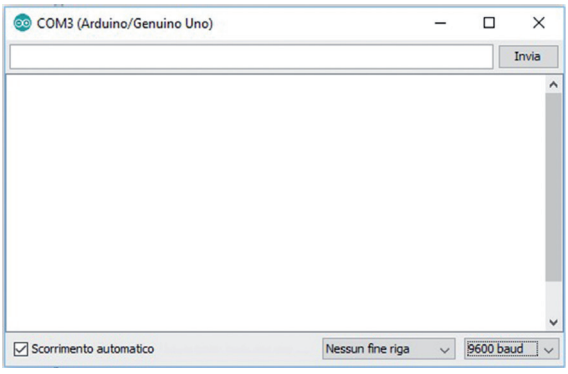

Fig. 10. *Finestra di comunicazione.*

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', il cui valore decimale è 48, il ciclo riparte da questo valore.

7.1 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.

ESEMPIO 1

```
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.

ESEMPIO 2

```
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)
```

8 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.

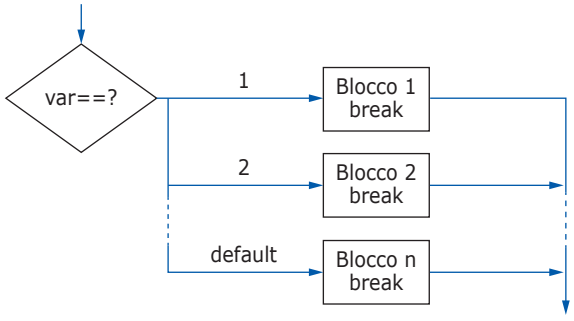

Fig. 11. *Struttura della selezione multipla switch/case.*

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 che segue, 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.

Si ricorda che il codice ASCII del numero 0 è 0x30, 1 = 0x31, ecc., pertanto 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;
 }
}
```

9 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 incremento.

```
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.
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.

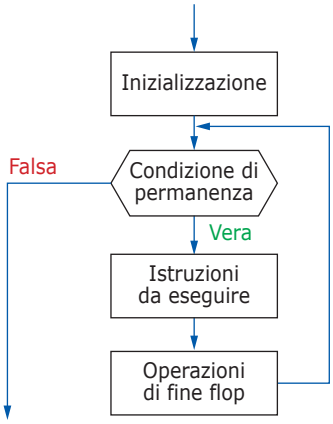

Fig. 12. *Struttura di un ciclo for.*

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 (ma non devono mancare i punto 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 ( ; ; ) { }
}
```

10 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;
}
```

Pertanto, se la condizione di ingresso non diventa mai falsa, il loop risulta **infinito**.
Al contrario, se la condizione è falsa già al primo impatto, 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++;
}
```

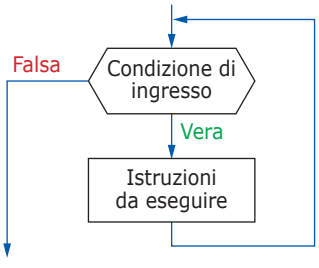

Fig. 13. *Schema a blocchi di un ciclo while.*

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);
```

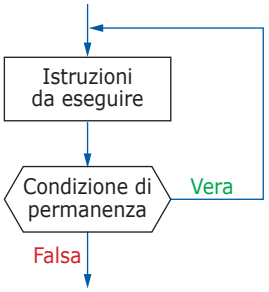

Fig. 14. *Schema a blocchi del ciclo do-while.*

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);
```

11 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, numerati 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 1, aumentando e diminuendo gradualmente il PWM che lo governa.

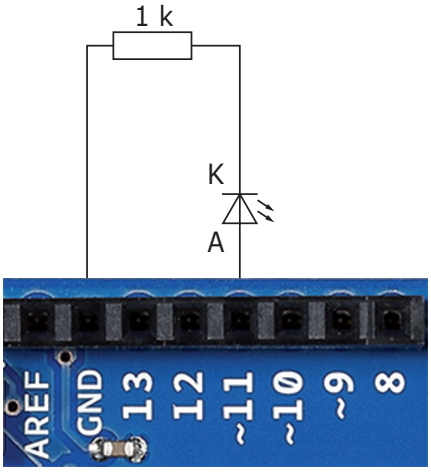

Fig. 15. *LED esterno connesso sul pin 11.*

```
/* 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);
 }
}
```

12 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);
/ ...
}
```

Regolazione luce LED

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

```
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 valore 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).

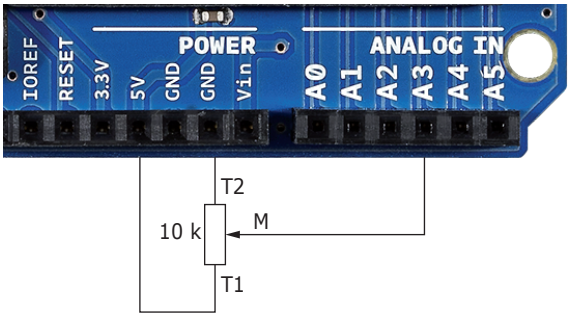

Fig. 16. *Potenziometro connesso sul pin A3.*