Arduino Quick Reference

Di Luca Panebianco per Automazione Open Source

Versione 1.0

www.xploreautomation.com

Indice dei contenuti

La lista di strutture,dati e funzioni è stata direttamente tratta dal sito ufficiale di Arduino alla pagina
http://arduino.cc/it/Reference/HomePage

1 Strutture

1.1 Strutture fondamentali del programma

setup(){...}	Inizializzazione del programma. All'interno vengono settati gli ingressi/uscite, le variabili e fatti partire alcuni servizi(come la seriale).
Loop(){...}	Questo è in mail loop del programma. Il codice all'interno verrà ripetuto all'infinito.

1.2 Strutture di controllo di base

If(cond){...}	Se viene verificata la condizione(*cond*) dell'*if* viene eseguito il codice all'interno del blocco.
	Es:
	if(i<0){
	… //codice eseguito se i<0
	}

If(cond) else{...}	Se in una struttura *if* immediatamente precedente la condizione(*cond*) non è stata rispettata allora si entra nel blocco dell' *else*.
	Es:
	if(i<0){
	…
	}
	else{
	… //questo codice viene eseguito se non vale la condizione dell'if
	}

for(asseg; cond; inc){...}	Il ciclo *for* serve per ripetere ciclicamente un blocco. Assegnato un riferimento numerico(*asseg*) finchè viene rispettata una certa condizione(*cond*) si entra nel blocco e alla fine di esso viene applicato l'incremento(*inc*). Es: for(int i=0; i<10;i++){ … //codice eseguito ad ogni ciclo, la variabile *i* può

```
                    essere modificata nel ciclo
}
```

`switch(var){`
`case val1:`
`break;`
`}`

Lo *switch* funziona come un interruttore. Passata una variabile a questa struttura(*var*) vengono applicate diverse azioni a differenza del suo valore attraverso i ***case***. Alla fine delle istruzioni per lo specifico caso si esce dalla struttura con un ***break.***

Es:

```
int var=1;
switch(var){
case 0:
…       //codice che non verrà eseguito poiché var!=1
break;
case 1:
…       //codice che verrà eseguito fino al break
break;
...
}
```

`while(cond){...}`

Se viene verificata la condizione(*cond*) si entra nel blocco. Un ciclo del tipo *while(true)* almeno ché non ci sia un'istruzione di *break* all'interno genera un ciclo infinito.

Es:

```
while(i!=0){
…       //loop che verrà eseguito finchè vale la
            condizione
}
```

`do{...}while(cond)`

Simile al *while()* con l'unica differenza che il blocco viene eseguito almeno una volta.

`break;`

Istruzione che permette di uscire dal ciclo attuale(*for*, *while* o *do...while*) o uscire da uno *switch*.

`continue;`

Il continue serve a saltare le istruzioni successive in un loop riportandoci all'inizio dello stesso.

`return var;`

Le funzioni spesso hanno una variabile di **ritorno**. Con questa istruzione facciamo proprio questo: impostiamo come ritorno il valore di una variabile.

goto(label);	Trasferisce il flusso del programma nel punto in cui è definito *label.* Solitamente non consigliato nella programmazione C ma utilizzabile solo quando si hanno cicli *for* o *if* pesantemente nidificati.
	Es:
	`for(byte r = 0; r < 255; r++){`
	`  for(byte g = 255; g > -1; g--){`
	`  for(byte b = 0; b < 255; b++){`
	`  if (analogRead(0) > 250){ goto erroreConGoto;}`
	`  }`
	`  }`
	`}`
	`erroreConGoto:`
	
	Se si verifica la condizione di errore si va direttamente fuori dal ciclo *for*.

1.3 Strutture di sintassi

… ;	Carattere di fine istruzione.
{...}	Le parentesi graffe circondano un blocco di istruzioni.
//...	Questi due caratteri identificano un commento su un'unica riga.
/* ... */	Questi caratteri identificano un commento che si può scrivere su più righe.
#define	Serve per definire costanti utilizzate all'interno del programma. Devono essere definite all'inizio del programma stesso.
#include	Serve ad includere altri file nel nostro programma come librerie e altri file che magari hanno solo costanti.

1.4 Operatori aritmetici

=	Operatore di assegnamento: alla variabile sulla sinistra viene dato uno specifico valore. Es. *int a=5;*
+	Operatore di somma.
-	Operatore di sottrazione.
*	Operatore di moltiplicazione.
/	Operatore di divisione.
%	Operatore di resto di una divisione.

1.5 Operatori di comparazione

==	Operatore di uguaglianza: riporta vero solo se gli elementi a destra e sinistra hanno lo stesso valore.
!=	Operatore di disuguaglianza: riporta vero solo se l'operatore di uguaglianza riporta falso.
<	Operatore di minoranza stretta: riporta vero se e solo se l'elemento a sinistra è minore dell'elemento a destra.
>	Operatore di maggioranza stretta: riporta vero se e solo se l'elemento a sinistra è maggiore dell'elemento a destra.
<=	Operatore di minoranza non stretta: riporta vero se e solo se l'elemento a sinistra è minore o al massimo uguale dell'elemento a destra.
>=	Operatore di maggioranza non stretta: riporta vero se e solo se l'elemento a sinistra è maggiore o al massimo uguale dell'elemento a destra.

1.6 Operatori booleani

&&	Operatore *AND* Es. *a && b* riporterà vero solo se entrambe le condizioni sono vere.
||	Operatore OR Es *a || b* riporterà vero se almeno una delle due condizioni è vera
!	Operatore NOT Es. *! a* riporterà vero sono se la condizione *a* è falsa

1.7 Operatori per l'accesso a puntatore

* (puntatore) & (riferimento)	Puntatori e riferimenti sono alcune tra le features più complicate per chi inizia a programmare in C e non è semplice spiegarlo qui in poche parole . Comunque per scrivere la maggior parte degli sketch non ci sarà bisogno di queste conoscenze

1.8 Operatori bit a bit(bitwise)

bin1 & bin2	Operatore che fa l'AND bit a bit Es: 1 0 0 1 1 & 1 0 0 0 1 ---------- 1 0 0 0 1
bin1 | bin2	Operatore che fa l'OR bit a bit Es: 1 0 0 0 1 & 1 1 0 0 0 ---------- 1 1 0 0 1
^bin1	Operatore che fa lo XOR(eXclusive OR) bit a bit Es: 1 0 0 1 1 & 1 1 0 1 0 ---------- 0 1 0 0 1

~bin1	Operatore di NOT bit a bit Es: ~ 1 0 0 1 1 ---------- 0 1 1 0 0
var << n_bit	Operatore che consente lo shift(scorrimento) a sinistra per un certo valore di bit(*n_bit*)
var >> n_bit	Operatore che consente lo shift(scorrimento) a destra per un certo valore di bit(*n_bit*)

1.9 Operatori composti

var++/++var	Operatore di incremento: aumenta il valore della variabile di 1. esiste preincremento(*++i*) che ritorna il nuovo valore di i, e postincremento(*i++*) che riporta il vecchio valore di i.
	Es:
	int i,h=5;
	int a=++i; // a vale 6
	int a=i++; // a vale 5

var- -/- -var	Operatore di decremento: diminuisce il valore della variabile di 1.esiste predecremento(*- -i*) che ritorna il nuovo valore di i, e postdecremento(*i- -*) che riporta il vecchio valore di i.
var+=num	Operatore di somma composta: sostituisce il valore della variabile a sinistra con la somma della stessa con il valore a destra. x+=y equivale a x=x+y
var-=num	Operatore di sottrazione composta: sostituisce il valore della variabile a sinistra con la differenza della stessa con il valore a destra. x-=y equivale a x=x-y

var*=num	Operatore di moltiplicazione composta: sostituisce il valore della variabile a sinistra con il prodotto della stessa per il valore a destra. x*=y equivale a x=x*y
var/= num	Operatore di divisione composta: sostituisce il valore della variabile a sinistra con la divisione della stessa per il valore a destra. x/=y equivale a x=x/y
bin1&=bin2	Operatore di *AND* bit a bit composto: sostituisce il valore della variabile a sinistra con l'*AND* bit a bit della stessa per il valore a destra. x &= y equivale a x=x & y
bin1|=bin2	Operatore di *OR* bit a bit composto: sostituisce il valore della variabile a sinistra con l'*OR* bit a bit della stessa per il valore a destra. x |= y equivale a x=x | y

2 Variabili

2.1 Costanti

HIGH | LOW	Queste sono costanti atte a indicare lo stato del livello di un pin, sia di INPUT che di OUTPIT. Un input vale HIGH se viene rilevata una tensione maggiore di 3 volt, LOW altrimenti. Un output che viene settato ad HIGH avrà in uscita una tensione di 5 volt, 0 altrimenti.
INPUT | OUTPUT	Queste due costanti vengono utilizzate nella funzione *pinMode* e servono a identificare se un pin è utilizzato come uscita o come ingresso.
true | false	Questi sono i due valori logici delle operazioni booleane. False,inteso come valore numerico, è equivalente allo 0, true altrimenti(nel senso che è un valore diverso da 0, anche -200 equivale true). A differenza delle altre costanti, queste vengono scritte con le lettere minuscole.

Costanti intere	Sono valori interi inseriti direttamente in uno sketch(nell'assegnamento ad esempio). Possono essere scritti in diverse basi o avere altri modificatori.
	Base 10(decimale): senza alcuna formattazione.
	Es: int a =10;
	Base 2(binaria): si prepone una **'B'** al valore e si accettano solo valori tra 0 e 1 e per valori tra 0 e 255.
	Es: int a =B0100;
	base 8(ottale): si prepone **'0'** al valore e si accettano solo valori tra 0 e 7. NB: Attenzione nel mettere uno 0 in altri valori che non si intendono come ottali!
	int a =023432;
	base 16(esadecimale): si prepone **0x** e si accettano valori tra 0 e 9 e in più i valori 'A'(10), 'B'(11), 'C'(12), 'D'(13), 'E'(14), 'F'(15). Valgono anche le lettere minuscole.
	int a =0x12AaB;
	I modificatori sono due e si pospongono alla costante:
	'u' o 'U' : per imporre la costante come **unsigned**
	'l' o 'L' : per imporre la costante come **long**
	su può usare anche 'ul' o 'UL' per unire entrambe le proprietà.
	int a =1123UL;

Costanti a virgola mobile	Le costanti a virgola mobile sono costanti non intere. Per indicare la virgola che suddivide interi e decimali si usa il punto('.') e per indicare l'esponente la lettera 'e' o 'E' Es: 2.34E2 rappresenta 2,34*10^2

2.2 Tipi di dato

void	Tipo utilizzato solo come ritorno di una funzione. Significa che la funzione non riporterà alcun dato come ad esempio la funzione loop().
boolean	Una variabile booleana ha due possibili valori *true*(vero) o *false*(falso) e occupa un byte di memoria
char	Il tipo *char* identifica un singolo carattere ASCII e occupa un byte di memoria(255 valori) ed è di tipo signed(con segno). E' equivalente scrivere: char c = 'a'; char c =65;

unsigned char	L'*unsigned char* è uguale al *char*, però è di tipo unsigned. Comunque, per riferirsi ad una variabile unsigned di 8 bit è preferibile utilizzare byte .
byte	Il *byte* serve a memorizzare un numero a 8 bit unsigned
int	L'*int* è un tipo di variabile intera scritta in 2 byte signed. Il range va da -32,768 a 32,767.
unsigned int	Semplicemente un *int* senza segno con un range tra 0 e 65,535.
word	Il tipo *word* ha lo stesso significato di un'insigned int.
long	Il tipo *long* identifica degli *int* estesi a 4 byte. Il loro range va da -2,147,483,640 a 2,147,483,639
unsigned long	Il tipo *unsigned long* è un *long* senza segno. Il suo range va da 0 a 4,294,967,295.
float	Il tipo *float* identifica dei numeri scritti in virgola mobile signed su 32 bit(4 byte). Il loro range va da 3.4028235E+38 a -3.4028235E+38.
double	Il tipo *double* a differenza di altri linguaggi è praticamente un *float*(quindi su 32 bit invece di 64) senza guadagni in precisione.
string (array di char) String (Oggetto)	Il tipo *string* si può definire in due modi: o come array di caratteri o come un oggetto. Considerandolo come array di caratteri, la stringa è un array di caratteri con l'aggiunta di un terminatore '\0'(facoltativo, verrà aggiunto dal compilatore). Es: chat str1[8]={'a', 'r', 'd', 'u', 'i', 'n', 'o', '\0'} chat str2[8]={'a', 'r', 'd', 'u', 'i', 'n', 'o'} chat str3[]=”arduino” Questi sono tre modi per poter creare una stringa.

	Considerandolo come un oggetto, i suoi metodi consentono la sua manipolazione. Questi verranno affrontate più avanti(cap 4.1).

array	Gli *array* sono collezioni di variabili dello stesso tipo con accesso indicizzato(attraverso la loro posizione dall'inizio dell'array stesso).
	Es: `int array1[]={1,2,3,4,5}`
	`int array2[5]={1,2,3}`
	`int array3[5];`
	`int val = array1[3];`
	Nel primo caso la grandezza dell'*array* viene gestita dal compilatore, nel secondo caso la inseriamo noi, e non dobbiamo per forza riempire l'*array* subito. Nel caso non vengano definiti gli elementi dell'*array*(caso 1) si deve per forza indicare la grandezza dell'*array*.
	Per prelevare un valore dall'*array* si utilizza l'esempio 4.
	NB: il compilatore non guarda se si sta accedendo ad un'area esterna dell'*array*!

2.3 Conversione

char(val)	Converte un valore in *char* dato un valore(val). Ritorna un *char*.
byte(val)	Converte un valore in *byte* dato un valore(val). Ritorna un *byte*.
int(val)	Converte un valore in *int* dato un valore(val). Ritorna un *int*.
word(val) word(byte1,byte2)	Converte un valore in *word* dato un valore(val), o utilizza due *byte* per creare un *word*. ritorna un *word*.
long(val)	Converte un valore in *long* dato un valore val. Ritorna un *long*.
float(val)	Converte un valore in *float* dato un valore val. Ritorna un *float*.

2.4 Funzioni di utilità

SizeOf(var)	Questo operatore riporta il numero di byte di una variabile.

3 Funzioni

3.1 I/O Digitali

pinMode(pin,mode)	Serve a configurare un pin. Necessita il suo numero(pin) e la modalità(mode), rappresentate dai due valori costanti **INPUT** e **OUTPUT**
digitalWrite(pin,value)	Serve a scrivere un uscita ad un determinato pin(pin)di OUTPUT. Il valore(value) può essere **HIGH** o **LOW**. NB L'unica accortezza segnalata è di non usare il pin13 come uscita digitale perchè a quella uscita è collegato un Led e una resistenza che porteranno il livello di uscita sempre a **LOW**
digitalRead(pin)	Legge il valore ad un dato pin. Ritorna **HIGH** o **LOW**.

3.2 I/O Analogici

analogReference(tipo)	Configura il voltaggio di riferimento per gli input analogici. I tipi disponibili sono: *DEFAULT*: il valore di defaulti di 5V(per che schede da 5V) o 3.3V(per le schede da 3.3V) *INTERNAL*: è un riferimento del microcontrollore. E' di 1.1V per l'ATmega168 o 328 e di 2,56 per l'ATmega8(non disponibile per l'Arduino Mega) *INTERNAL1V1*: riferimento di 1.1V(solo per Arduino Mega) *INTERNAL2V56*: riferimento di 2.56V(solo per Arduino Mega) *EXTERNAL*: prende come riferimento il voltaggio presente nel pin AREF(tra 0 e 5V)
analogRead(pin)	Legge il valore della tensione al pin richiesto e riposta un valore intero positivo tra 0 e 1023
analogWrite(pin,valore)	L'analogWrite funziona come una **PWM** e il valore che passiamo deve essere un valore intero compreso tra **0 e 255** che rappresenta il duty cicle dell'onda quadra.

3.3 I/O Avanzati(audio,shift,pulse)

Tone(pin, freq) Tone(pin, freq, durata)	Genera un onda quadra che rappresenta il tono di una nota ad un dato pin per una certa durata(facoltativa). Senza la durata la nota verrà suonata finchè non viene chiamato noTone(). Nel caso si vogliano utilizzare più pin bisogna per forza richiamare noTone() prima di richiedere il suono ad un altro pin.
noTone(pin)	Disattiva la generazione dell'onda quadra attivata da Tone() al dato pin.
ShiftOut(pin, clockpin, ordine, val))	Fa lo scorrimento seriale di un byte(val) un bit alla volta utilizzando un certo pin come uscita. Lo scorrimento(ordine) può essere per il bit più significativo per primo(MSBFIRST) o per quello meno significativo per primo(LSBFIRST). Quando l'operazione è finita viene fatto il toggle del pin clickpin.
shiftIn(pin, clockpin,ordine)	Operazione inversa di shiftOut. Dato un pin dove leggere, un clockPin nel quale notificare la lettura e un modo per leggerlo(MSBFIRST o LSBFIRST) viene riportato il byte letto.
pulseIn(pin, val) pulseIn(pin, val,timeout)	legge un impulso da un dato pin. Se il valore(val) è **HIGH** si aspetta che il pin passi a **HIGH**, inizia il timer, aspetta che torni **LOW** e smette il timer. Ritorna la lunghezza dell'impulso in microsecondi. Con il valore **LOW** il procedimento è inverso e si aspetta un passaggio ha **HIGH** a **LOW** prima di far partire il timer. Si può aggiungere un timeout che interrompe l'attesa dell'impulso se non arriva entro quel tempo.

3.4 Temporali

Millis()	Riporta un intero che rappresenta i millisecondi da quando il programma è stato avviato. Utile se si vogliono misurare intervalli di tempo.
micros()	Riporta un intero che rappresenta i microsecondi da quando il programma è stato avviato. Utile se si vogliono misurare intervalli di tempo.

Delay(time)	Blocca temporaneamente il programma per il numero di millisecondi passato come parametro.
delayMicroseconds(time)	Blocca temporaneamente il programma per il numero di microsecondi passato come parametro.

3.5 Matematiche

min(x,y)	Ritorna il minimo tra due numeri x e y NB:non utilizzare funzioni all'interno degli argomenti! Es: non fare min(x++,y)
max(x,y)	Ritorna il massimo tra due numeri x e y NB:non utilizzare funzioni all'interno degli argomenti!
abs(x)	Ritorna il valore assoluto di x
constrain(x, a, b)	Limita il valore x tra a(minimo) e b(massimo) e lo ritorna Es: se x è compreso tra a e b ritorna x se x è minore di a ritorna a se x è maggiore di b ritorna b
map(val, a1, b1, a2, b2)	Ritorna il valore mappato invece che nell'intervallo a1/b1 in quello a2/b2. Praticamente fa una proporzione.
pow(base, exp)	Ritorna la base elevata all' esponente. Base e esponente possono essere anche *float*.
sqrt(val)	Ritorna la radice quadrata di un numero.

3.6 Trigonometriche

sin(rad)	Ritorna il valore del seno di un angolo(in radianti). Il valore è compreso tra -1 e 1.
cos(rad)	Ritorna il valore del coseno di un angolo(in radianti). Il valore è compreso tra -1 e 1.
tan(rad)	Ritorna il valore della tangente di un angolo(in radianti). Il valore è compreso tra meno infinito e infinito

3.7 Generatrici di numeri casuali

randomSeed(val)	Inizializza il generatore pseudo-casuale facendo in modo che esso inizi da un dato valore. Questa sequenza anche se molto lunga, sarà sempre la stessa. Un consiglio è utilizzare come valore iniziale un valore analogico preso da un pin non utilizzato.
random(max) random(min,max)	Genera un valore casuale intero o dato il suo limite massimo o sia dando entrambi i limiti(minimo e massimo).

3.8 Riguardanti Bit e Bytes

LowByte(val)	Ritorna il byte di ordine più basso(quello più a destra) da una variabile.
highByte(val)	Ritorna il byte di ordine più alto(quello più a sinistra) da una variabile.
bitRead(x, n)	Ritorna un bit dalla variabile *x* alla posizione *n*.
bitWrite(x, n, b)	Con questa funzione si può scrivere sulla variabile *x* alla posizione del bit *n* il valore di b(0 o 1). Non ritorna nulla.
bitSet(x, n)	Setta a 1 il valore del bit *n* della variabile *x*.
bitClear(x, n)	Setta a 0 il valore del bit *n* della variabile *x*.

bit(n)	Ritorna il valore di un numero espresso in bit(n).

3.9 Interrupt esterni

attachInterrupt (int, function,mode)	Permette di specificare una funzione da chiamare(routine di interrupt) quando un evento di una certa modalità avviene al pin di interrupt. Modalità: **LOW**: si attiva quando il pin è low **CHANGE**:si attiva quando il pin cambia valore **RISING**:si attiva quando il pin passa da low a high **FALLING**:si attiva quando il pin passa da high a low.
detachInterrupt(pin)	Disattiva l'interrupt ad un pin impostato con attachInterrupt().

3.10 Interrupt

noInterrupts()	Disabilita gli interrupt, abilitati di default. Alcune funzioni sono disabilitate senza interrupt e le comunicazioni in ingresso potrebbero essere ignorate.
interrupts()	Riabilita gli interrupt disabilitati con noInterrupts().

3.11 Comunicazione

Serial	Utilizzato per comunicare tra l'Arduino e il PC o altre device. L'Arduino ha due pin digitali dedicati a questo scopo: 0(Rx) e 1(Tx). Da ricordare che questi due canali comunicano in TTL quindi se bisogna comunicare col pc(che utilizza il protocolo RS232) serve un integrato tipo il MAX232 che faccia la conversione tra i due protocolli. I suoi metodi verranno affrontate più avanti(cap 4.2).
Stream	Questa è la classe base per poter gestire stream basati su bit e bytes. Non viene chiamata direttamente ma invocata ogni volta che viene richiamato un suo metodo.

4 Alcune classi standard

4.1 Funzioni della classe String

Operatori

[]	Con questo operatore si può accedere ad un preciso carattere della stringa.
+	Operatore di concatenazione, unisce più stringhe tra loro Es: String s =”hello”+” “+”world”;
==	Operatore di uguaglianza tra stringhe. Ha la stessa valenza di *equals()*.

Funzioni

Questi metodi devono essere applicati ad un oggetto del tipo stringa.

Se abbiamo creato una stringa in questo modo:

```
str1= String(“Arduino”);
```

Per poter vedere la sua lunghezza dovremo scrivere

```
int len = str1.length();
```

String(val) String(val, base)	Questo è il costruttore della classe String e quindi ritorna un'istanza di quella classe. Si può passare direttamente un valore che può essere di tipo *string*, *char*, *byte, int*, *long*, *unsigned int* e *unsigned long*. Questo valore viene tradotto in una stringa di caratteri che lo rappresenta. Per i tipi interi si può aggiungere la base che vogliamo utilizzare: quelle disponibili sono: **BIN**, **OCT**, **DEN**, **HEX**.
charAt(n)	Questa funzione permette di riportare direttamente il carattere *n* di una stringa,
compareTo(str2)	Questa funzione permette di comparare due stringhe e dire se una viene prima dell'altra. Il valore riportato sarà negativo se la nostra stringa viene prima di quella di confronto, 0 se sono

	uguali mentre un numero positivo se la nostra stringa viene dopo di quella di confronto.
concat(str1, str2)	Combina o concatena due stringhe in una nuova. La seconda stringa viene appesa alla prima e il risultato messo un una nuova String.
endsWith(str2)	Testa(riportando *true* o *false*) se la stringa di prova si trova in fondo alla nostra stringa.
equals(str2)	Testa due stringhe per vedere se sono uguali. Il confronto è case sensitive.
equalsIgnoreCase(str2)	Testa due stringhe per vedere se sono uguali. Il confronto non è case sensitive.
getBytes(buf, len)	Copia i singoli bytes di una stringa in un buffer(buf) di una data lunghezza(len).
indexOf(val) indexOf(val, from)	Cerca un carattere o stringa(val) all'interno della nostra stringa, riportando un intero che identifica la posizione della prima occorrenza trovata. Restituisce -1 se non vengono trovate occorrenze. In aggiunta si può aggiungere un parametro(from) che identifica da che carattere fare la ricerca.
lastIndexOf() lastIndexOf(val, len)	Cerca un carattere o stringa(val) all'interno della nostra stringa, riportando un intero che identifica la posizione dell' ultima occorrenza trovata. Restituisce -1 se non vengono trovate occorrenze. In aggiunta si può aggiungere un parametro(from) che identifica da che carattere fare la ricerca.
length()	Ritorna un valore intero che identifica la lunghezza della stringa(senza il terminatore di stringa).
replace(val1, val2)	Serve a sostituire tutte le istanze di una carattere(val1) con un altro(val2). Si può anche usare per sostituire una sottostringa della nostra stringa con un altra.

setCharAt(ind, c)	Serve a settare un carattere(c) della stringa ad una certa posizione(ind).
startsWith(str)	Testa se una stringa inizia con i caratteri di un'altra(str).
substring(from) substring(from, to)	Ritorna una sottostringa della nostra stringa a partire da una certa posizione(from) in aggiunta si può definire anche fino a che carattere prelevare la sottotringa attraverso un' altra posizione(to).
toCharArray(buf, len)	Trasferisce il contenuto di una stringa in un buffer di char(buf) di una certa lunghezza(len).
toLowerCase()	Sostituisce la stringa con una con caratteri minuscoli. Prima della versione 1.0 veniva riportata una stringa.
toUpperCase()	Sostituisce la stringa con una con caratteri maiuscoli. Prima della versione 1.0 veniva riportata una stringa.
trim()	Sostituisce la stringa senza alcun carattere di spazio. Prima della versione 1.0 veniva riportata una stringa.

4.2 Funzioni della classe Serial

begin(bRate)	Si inizia la comunicazione seriale potendone settare anche la velocità (Baud Rate) in bit per secondo.
end()	Disabilita la comunicazione seriale.
available()	Riporta il numero di *byte* disponibili per essere letti dalla porta seriali. Questi dati sono già arrivati e salvati in un buffer(di 128 bytes).
read()	Riporta il prossimo *byte* disponibile della seriale. Riporta -1 se non ci sono dati da leggere.

peek()	Riporta il prossimo dato disponibile della seriale ma non lo rimuove dal buffer interno. Più chiamate di peek() successive riporterà lo stesso carattere.
flush()	Aspetta che tutti i dati in uscita dalla seriale siano inviati. Prima della versione 1.0 questo metodo cancellava ogni dato presente nel buffer di lettura.
print(num,) print(num, valore)	Scrive dati sulla porta seriale come un testo leggibile ASCII. I numeri interi vengono scritti utilizzando un carattere ASCII per ogni cifra mentre i float vengono approssimati alla seconda cifra dopo la virgola e poi trasmessi. Caratteri e stringhe vengono trasmessi così come sono. Si può aggiungere una valore per definire un formato in cui verrà tradotto il valore numerico prima che sia inviato. Per gli interi valgono i valori **BIN**(conversione binaria), **OCT**(conversione ottale), **DEC**(conversione decimale), **HEX**(conversione esadecimale). Mentre per i numeri in virgola mobile il valore identifica il numero di cifre dopo la virgola che si possono inviare. Ritorna il numero di *byte* scritti nella seriale.
println(num) println(num, valore)	Fa la stessa azione di print() però aggiunge in fondo alla stringa inserita come parametro alla funzione il carattere di nuova linea('\n': codice ASCII 10).
write(val) write(str) write(array,lun)	Funziona come print, ma senza cercare di rendere i caratteri leggibili col codice ASCII(un valore numerico non viene tradotto come la serie dei caratteri delle singole cifre). Le stringhe vengono mandate invece proprio come la funzione print(), mentre passando come parametri un buffer(array) e un valore(lun) vengono mandati i primi n valori del buffer. Anche questa funzione ritorna il numero di *byte* scritti.
SerialEvent()	Questa funzione viene chiamata quando ci sono dei dati disponibili sul buffer di lettura.