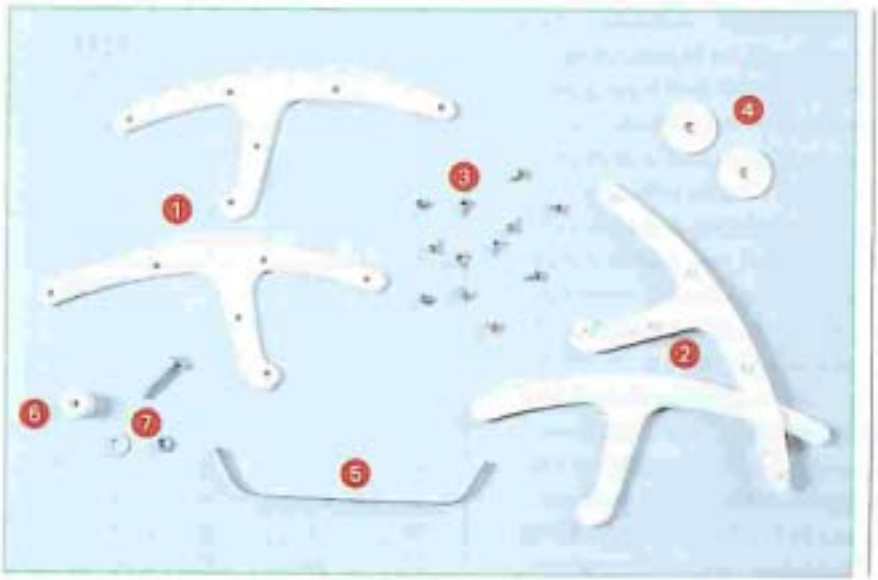
Il secondo kit per i sensori di contatto

Con il kit allegato, identico a quello illustrato a pagina 53, potrai disporre di altri due baffi. Montati sul circuito posteriore (come vedremo nel prossimo fascicolo) e sostituiti alla pinza, raddoppieranno le possibilità di percezione tattile del tuo robot. La copertura sensoriale offerta dalla doppia coppia di baffi, infatti, costituirà una sorta di scudo perimetrale.



A destra. **Gli allegati a questo fascicolo: le parti inferiori 1 e superiori 2 dei baffi, le viti 3 e le rondelle 4, la lamina 5, il distanziale 6 e la vite di fissaggio (con la relativa rondella e il dado M3) 7.**

Le fasi di programmazione

Riprendendo il discorso sulla possibilità di telecomandare il robot (iniziato a pagina 65), abbiamo già avuto modo di vedere (alle pagine 137-140 e 143-144) come e perché sia necessario anzitutto decodificare gli impulsi emessi dal LED del telecomando. Ricorderai che, allora, eravamo ricorsi a otto comandi di pulsin (il primo per il riconoscimento dell'inizio di una sequenza utile e i successivi sette contenenti l'informazione relativa al codice del tasto premuto sul telecomando). Dal momento che in fase di programmazione ogni semplificazione si traduce in un 'risparmio' di memoria e di tempo, vediamo come sia possibile migliorare ulteriormente l'uso del telecomando ottimizzando la scrittura del corrispondente programma.

La navigazione del tuo robot, garantita dai servomotori, richiede un frequente invio del segnale da parte del microcontrollore: ricorderai infatti che una pausa troppo lunga tra l'invio di un impulso e quello del successivo produce un movimento a scatti, per niente fluido. I tempi di decodifica del segnale richiesti da una routine di rilevazione e decodifica come quella di pag. 138 (costituita da otto pulsin e otto assegnamenti), ad esempio, rischiano facilmente di essere eccessivi. Per ridurre tali tempi e ottenere dal robot una migliore performance, bisogna però fare i conti con le caratteristiche specifiche del telecomando e con la sintassi delle istruzioni messe a disposizione dal linguaggio PBASIC: come vedremo nelle prossime pagine, infatti, una possibile ottimizzazione dipende dall'opportuna combinazione di una nuova istruzione (lookdown) e dei codici numerici relativi ai tasti del telecomando. Partiamo da quest'ultimo. Una prima osservazione utile è che, data la relativa semplicità del movimento di un robot su una superficie, raramente un programma di navigazione riguarderà l'uso di tutti i tasti presenti sul telecomando; molto più facilmente, invece, si 'concentrerà' esclusivamente su un ridotto sottoinsieme di essi, la cui scelta, però, non è irrilevante. A partire dalla tabella dei ventinove tasti del telecomando vista a pag. 140, facciamo ora un passo avanti, ordinandoli in base al valore crescente del codice numerico decimale a essi associato (vedi la tabella nella pagina seguente).

LE FASI DI PROGRAMMAZIONE

Questa nuova tabella indica, per ogni tasto, il relativo codice in valore decimale (DEC) e il corrispettivo binario (BIN): il primo determina l'ordine dei tasti (posizione), il secondo specifica (in rosso) il numero minimo di bit necessari a rappresentare univocamente quel tasto. In generale, come sai, usando *n* bit si possono rappresentare i numeri da 0 a (2^n-1) il che significa che usando 1 bit si possono rappresentare solo i numeri 0 e 1 (2^1-1), con 2 bit i numeri da 0 a 3 (2^2-1) e via di seguito. La tabella, confermando quanto detto, individua (con le linee rosse) più gruppi di tasti che, per essere rappresentati, richiedono lo stesso numero minimo di bit. In particolare:

- per rappresentare i tasti in posizione compresa tra 0 e 1 è sufficiente 1 bit;
- per rappresentare i tasti in posizione compresa tra 0 e 3 sono sufficienti 2 bit;
- per rappresentare i tasti in posizione compresa tra 0 e 7 sono sufficienti 3 bit;
- per rappresentare i tasti in posizione compresa tra 0 e 9 sono sufficienti 4 bit;
- per rappresentare i tasti in posizione compresa tra 0 e 17 sono sufficienti 5 bit
- per rappresentare i tasti in posizione compresa tra 0 e 24 sono sufficienti 6 bit
- per rappresentare i tasti in posizione compresa tra 0 e 28 sono sufficienti 7 bit.

Come vedremo tra poco, è vantaggioso scegliere nello stesso insieme quali tasti utilizzare in un programma, badando inoltre che siano quelli che necessitano il minor numero di bit possibile.

Riprendiamo, ad esempio, il programma di pag. 144. La scelta di utilizzare solo i primi sei tasti della tabella (illustrati in basso), risulta doppiamente felice: intanto i loro codici decimali costituiscono una successione completa a partire da 0 (il che permetteva di usare l'istruzione branch); inoltre, la codifica binaria di tali tasti richiede unicamente 3 bit. In termini di programmazione, questo secondo aspetto si traduce nella possibilità di ridurre a 3 sia i pulsin necessari alla decodifica del segnale (più uno iniziale) sia gli assegnamenti. Opportunamente modificato (come emerge dal listato nella pagina accanto), dunque, quel programma risulterà semplificato e le operazioni corrispondenti saranno più veloci. Analizzandolo, notiamo che le modifiche sostanziali che hanno permesso una drastica riduzione della memoria occupata sono due e riguardano la dichiarazione delle variabili e la subroutine per la decodifica (riquadrate in rosso). In particolare, nella dichiarazione sono state eliminate quattro variabili di tipo word (inputIR3, inputIR4, inputIR5, inputIR6); inoltre la dimensione della variabile *codicePulsante* è stata dimezzata: precedentemente di tipo byte, è diventata di tipo nibble. Infatti, dati i possibili valori che essa potrà assumere (da 0 a 5, come i codici decimali dei sei pulsanti utilizzati), basterà decodificare solo i primi 3 bit del codice binario (il quarto, infatti, sarà comunque 0). Nella subroutine di decodifica, dunque, sono rimaste solo le rilevazioni dei segnali associati ai 3 bit meno significativi del codice del tasto premuto. Il programma così scritto, però, parte dal presupposto che non vengano premuti altri tasti per errore: paga infatti la sua estrema semplicità e ottimizzazione con un'insufficiente resistenza agli errori. Se ad esempio venisse premuto il tasto in posizione 9, 11 o 25, il programma non potrebbe in alcun modo distinguerlo dal tasto in posizione 1: i 3 bit meno significativi, infatti, sono uguali (001).

Posizione	TASTO	Codice DEC	Codice BIN
0	ms(1)	0	0000000
1	M(1)	1	0000001
2	md(1)	2	0000010
3	ms(2)	3	0000011
4	M(2)	4	0000100
5	md(2)	5	0000101
6	P(1)	6	0000110
7	P(2)	7	0000111
8	P(3)	8	0001000
9	P(4)	9	0001001
10	14	16	0010000
11	17	17	0010001
12	13	18	0010010
13	16	19	0010011
14	1	20	0010100
15	stop	21	0010101
16	9	23	0010111
17	prg	29	0011101
18	4	37	0100101
19	6	43	0101011
20	15	54	0110110
21	5	56	0111000
22	2	58	0111010
23	12	60	0111100
24	3	63	0111111
25	11	97	1100001
26	7	116	1110100
27	10	117	1110101
28	8	124	1111100

Un altro aspetto legato alla decodifica dei tasti del telecomando riguarda l'ordine dei codici decimali che, a partire dal tasto in posizione 10, cessa di essere crescente e a incremento unitario (dopo la serie completa da 0 a 9, si 'salta' infatti a 16, per giungere poi, con il tasto in posizione 28, al codice decimale 124). Dunque, come si diceva a pag. 144, volendo utilizzare un ampio set di tasti, si dovrebbe ricorrere a una lunga serie di istruzioni if... then, rinunciando ai vantaggi offerti da un'unica istruzione branch.

In realtà, esiste un modo molto elegante per ovviare a questo inconveniente, che consiste nell'uso delle cosiddette tabelle di *lookup* (letteralmente, di 'ricerca'). Spesso, in ambito informatico, risulta molto utile rimappare un set di valori in un insieme più 'ordinato' e, dunque, più facilmente gestibile. Questa procedura, però, è lecita solo a patto che il rapporto tra i due insiemi (quello disordinato di partenza e quello ordinato di arrivo) sia biunivoco: a ciascun elemento del primo, cioè, deve corrisponderne uno e uno solo del secondo, e viceversa.

```
prova12modificata.bs2

'{$STAMP BS2}

'---------------- Dichiarazione Variabili ----------------

contatore         var        nib

codicePulsante    var        nib

movDx             var        word
movSx             var        word
segnaleInizio     var        word
inputIR0          var        word
inputIR1          var        word
inputIR2          var        word

'---------------- Dichiarazione Costanti ----------------

IR_SX             con        8
BASSO             con        0

SERVO_DX          con        13
SERVO_SX          con        12

ANTIORARIO_DX     con        1000
ANTIORARIO_SX     con        1000
ORARIO_DX         con        500
ORARIO_SX         con        500

'---------------- Programma Principale ----------------

low SERVO_DX
low SERVO_SX

main:

      gosub decodifica
      branch codicePulsante, [AvantiSx, Avanti, AvantiDx, IndietroSx, Indietro, IndietroDx]

goto main

'---------- Routine di navigazione ----------------

AvantiSx:
      movDx = ANTIORARIO_DX   : movSx = ANTIORARIO_SX   : goto esegui

Avanti:
      movDx = ORARIO_DX       : movSx = ANTIORARIO_SX   : goto esegui

AvantiDx:
      movDx = ORARIO_DX       : movSx = ORARIO_SX       : goto esegui

IndietroSx:
      movDx = ORARIO_DX       : movSx = ORARIO_SX       : goto esegui

Indietro:
      movDx = ANTIORARIO_DX   : movSx = ORARIO_SX       : goto esegui

IndietroDx:
      movDx = ANTIORARIO_DX   : movSx = ANTIORARIO_SX   : goto esegui

esegui:
for contatore = 1 to 10
            pulsout SERVO_DX, movDx
            pulsout SERVO_SX, movSx
            pause 3
      next
goto main

'---------- Subroutine decodifica impulsi IR ----------

decodifica:
  rileva:
     pulsin IR_SX, BASSO, segnaleInizio
     pulsin IR_SX, BASSO, inputIR0
     pulsin IR_SX, BASSO, inputIR1
     pulsin IR_SX, BASSO, inputIR2

  if segnaleInizio < 1000 then rileva

     codicePulsante.bit0 = inputIR0.bit9
     codicePulsante.bit1 = inputIR1.bit9
     codicePulsante.bit2 = inputIR2.bit9

return
```

Considerando ad esempio la nostra tabella (nella pagina a sinistra), noterai che tutte le colonne godono di tale corrispondenza biunivoca: la posizione 12, ad esempio, corrisponde solo al codice decimale 18 e viceversa. Tali colonne, essendo la rimappatura dei codici decimali dei 29 tasti con le 29 posizioni (numerate da 0 a 28), costituiscono di fatto una tabella di LookUp. Questo tipo di tabella, alla base anche di alcuni semplici metodi di cifratura (come il cifrario di Cesare, visto a pag. 68 della sezione Tecno), può essere gestito in PBASIC tramite due istruzioni complementari: lookup e *lookdown* (letteralmente, 'guardare dall'alto in basso'). Il comando lookup consente di estrarre da una lista il valore corrispondente a una posizione data; viceversa, lookdown memorizza la posizione in cui un valore di confronto dato si trova rispetto alla lista di partenza. Nella sintassi della prima istruzione, a lookup deve seguire anzitutto la *posizione* scelta (cioè una variabile, una costante o un'espressione numerica il cui valore sia compreso tra 0 e 255, ossia 2^8-1), seguita da una virgola (,); quindi, tra parentesi quadre ([]) seguite anch'esse dalla virgola, la serie dei *valori* da considerare (possono essere variabili, costanti o espressioni numeriche, separate da virgole, aventi valore massimo pari a 65 535, cioè 2^{16}-1); e, infine, la *variabile* in cui memorizzare il risultato della ricerca. Consideriamo, ad esempio, l'istruzione lookup *posizione*, [12, 14, 5, 88, 45, 0], *risultato*. Assegnando alla variabile *posizione* (di tipo nibble) il valore 1, lookup memorizza nella variabile *risultato* (di tipo byte) il valore 14, che nella lista è l'elemento in posizione 1 (l'ordine delle posizioni nella lista, infatti, parte dallo 0 compreso, che qui corrisponde al 12). Qualora il valore di *posizione* fosse superiore al numero di elementi contenuti nella lista, la variabile *risultato* manterrebbe il proprio (precedente) valore. La sintassi dell'istruzione complementare, invece, prevede che a lookdown segua il valore da confrontare (una variabile, una costante o un'espressione numerica) seguito dalla virgola; un operatore di confronto (=, >, >=, <=, < oppure <>) che, se assente, è =; la serie di valori (tra 0 e 65 535, separati da virgole) entro cui eseguire il confronto (tra [] seguite dalla virgola); la variabile *posizione* in cui memorizzarne l'esito (purché compreso tra 0 e 255). Dunque, dato un valore di confronto *valcfr*, lookdown individua il primo elemento di una lista data che soddisfi la condizione di confronto, memorizzandone poi la *posizione* in una variabile.

LE FASI DI PROGRAMMAZIONE

Nella lista, dunque, non devono esserci elementi uguali: il secondo, infatti, non verrebbe mai raggiunto. Supponiamo di avere l'istruzione **lookdown *valcfr*, [12, 14, 5, 88, 45, 0], *posizione***. Assegnato a *valcfr* (di tipo byte) il valore 88, lookdown memorizza nella variabile *posizione* (di tipo nib) il valore 3 (88, infatti, è il quarto elemento di una lista ordinata a partire da 0). Anche inserendo un altro operatore di confronto prima della lista (ad esempio <, invece di =, sottointeso), la ricerca effettuata da lookdown comporta il confronto del valore di *valcfr* (ad esempio, 13) con quelli della lista: trovato il primo (14) che soddisfi la condizione imposta dall'operatore (13<14), ne memorizza la *posizione* (1). Se anche altri elementi della lista verificassero la condizione (13<88 e 13<45), essi non sarebbero considerati, perché lookdown restituisce sempre e solo la posizione del primo elemento che soddisfa il confronto. Come per lookup, inoltre, se nessun valore della lista verifica il confronto, *posizione* rimane invariata. Detto questo, torniamo al problema relativo ai codici inviati dai tasti del telecomando: grazie all'istruzione lookdown (che ordina un qualunque set di valori secondo la loro posizione, ossia una serie di per sé crescente e consecutiva), è possibile eseguire l'istruzione di branch anche con valori disparati. Vediamo ad esempio un programma di navigazione (a destra) che utilizzi sette tasti: i sei relativi al movimento e lo stop. Poiché al tasto stop corrisponde il codice decimale 21 (cioè 0010101, con 5 bit significativi), per gestirne la ricezione, servono 5 bit, memorizzati in *codicePulsante*. Tuttavia, sarà poi la variabile *posizione* (dichiarata di tipo nib) a essere testata dall'istruzione branch: a differenza di *codicePulsante* infatti, che rientrerebbe in una lista non solo ampia (tra 0 e 21), ma anche incompleta (da 6 a 20), la sua corrispondente posizione, memorizzata con lookdown, risulta invece compresa in una serie completa a partire da 0 (tra 0 e 6), adatta cioè all'uso di branch. Infine, il fatto che posizione sia stata inizializzata (=) a 7 preserva il programma dalla pressione di tasti estranei (a patto, però, che i bit testati non corrispondano a quelli di uno dei tasti scelti): senza questa accortezza, infatti, *posizione* manterrebbe l'ultimo valore valido acquisito, rimandando perciò a una routine di navigazione indesiderata; così, invece, il branch va eseguito sul valore iniziale (7) che, non comparendo nella lista, 'neutralizza' lookdown e rimanda semplicemente all'inizio del main.

```
provaIR3bs2

'{$STAMP BS2}

'---------------- Dichiarazione Variabili ----------------

contatore          var        nib

codicePulsante     var        byte
posizione          var        nib

movDx              var        word
movSx              var        word
segnaleInizio      var        word
inputIR0           var        word
inputIR1           var        word
inputIR2           var        word
inputIR3           var        word
inputIR4           var        word

'---------------- Dichiarazione Costanti ----------------

IR_SX              con        6
BASSO              con        0

SERVO_DX           con        13
SERVO_SX           con        12

ANTIORARIO_DX      con        1000
ANTIORARIO_SX      con        1000
ORARIO_DX          con        500
ORARIO_SX          con        500

'---------------- Programma Principale ----------------

low SERVO_DX
low SERVO_SX

main:

      gosub decodifica
      posizione = 7
      lookdown codicePulsante, [0,1,2,3,4,5,21], posizione
      branch posizione, [AvantiSx, Avanti, AvantiDx, IndietroSx, Indietro, IndietroDx, Fine]

goto main

'---------- Routine di navigazione ----------------------

AvantiSx:
      movDx = ANTIORARIO_DX   : movSx = ANTIORARIO_SX   : goto esegui

Avanti:
      movDx = ORARIO_DX       : movSx = ANTIORARIO_SX   : goto esegui

AvantiDx:
      movDx = ORARIO_DX       : movSx = ORARIO_SX       : goto esegui

IndietroSx:
      movDx = ORARIO_DX       : movSx = ORARIO_SX       : goto esegui

Indietro:
      movDx = ANTIORARIO_DX   : movSx = ORARIO_SX       : goto esegui

IndietroDx:
      movDx = ANTIORARIO_DX   : movSx = ANTIORARIO_SX   : goto esegui

esegui:
for contatore = 1 to 10
            pulsout SERVO_DX, movDx
            pulsout SERVO_SX, movSx
            pause 3
      next
goto main

'---------- Subroutine decodifica impulsi IR ----------

decodifica:
   rileva:
      pulsin IR_SX, BASSO, segnaleInizio
      pulsin IR_SX, BASSO, inputIR0
      pulsin IR_SX, BASSO, inputIR1
      pulsin IR_SX, BASSO, inputIR2
      pulsin IR_SX, BASSO, inputIR3
      pulsin IR_SX, BASSO, inputIR4

   if segnaleInizio < 1000 then rileva

      codicePulsante.bit0 = inputIR0.bit9
      codicePulsante.bit1 = inputIR1.bit9
      codicePulsante.bit2 = inputIR2.bit9
      codicePulsante.bit3 = inputIR3.bit9
      codicePulsante.bit4 = inputIR4.bit9

return

Fine: end
```