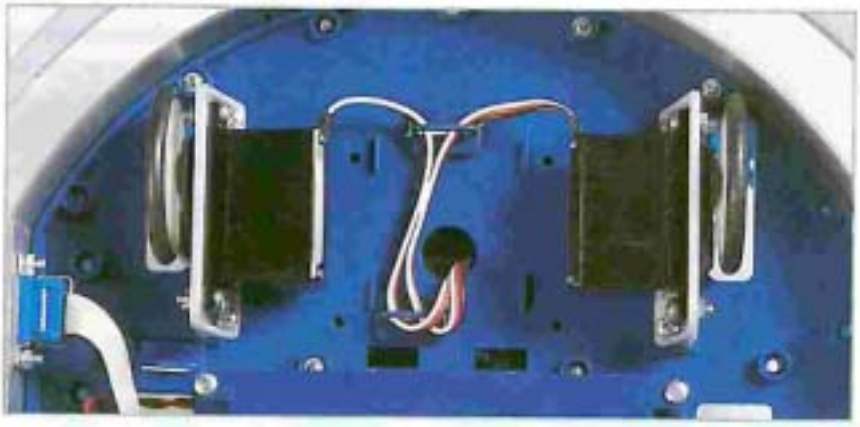
# Una ruota nuova per il tuo robot

Come preannunciato fin dall'inizio, eccoci giunti alle soglie di una nuova fase di espansione del robot dal punto di vista hardware. Con gli allegati di questo e dei prossimi tre fascicoli, infatti, potrai migliorare notevolmente le sue prestazioni locomotorie grazie alla sostituzione dei motori a spazzola con i servomotori. Inoltre, con gli allegati successivi, inizierai a montare il manipolatore (una pinza), azionato proprio dai motori a spazzola che prima facevano muovere il robot. Ovviamente, per continuare a fare esperimenti, ti converrà operare la sostituzione solo dopo aver ricevuto entrambe le ruote e i servomotori, anche perché, come vedremo presto, dovrai modificare anche il cablaggio per l'alimentazione. Come sai, i motori a spazzola necessitano dell'intermediazione dei driver e, dunque, dipendono dalla scheda di controllo motori; invece i servomotori saranno collegati direttamente alla scheda madre. Infatti un *servo-motore* (oppure motore asservito) come, più in generale, un servosistema, comporta per definizione l'idea di un dispositivo dotato di regolazione automatica ed esclude perciò la necessità di un'intermediazione rispetto al microcontrollore.


**Sopra. Gli allegati a questo fascicolo: una ruota in plastica 1 con il copertone in gomma piena 2 e il supporto 3 per il primo servomotore.**

**A sinistra. La ruota completa: il copertone va applicato con una leggera pressione.**

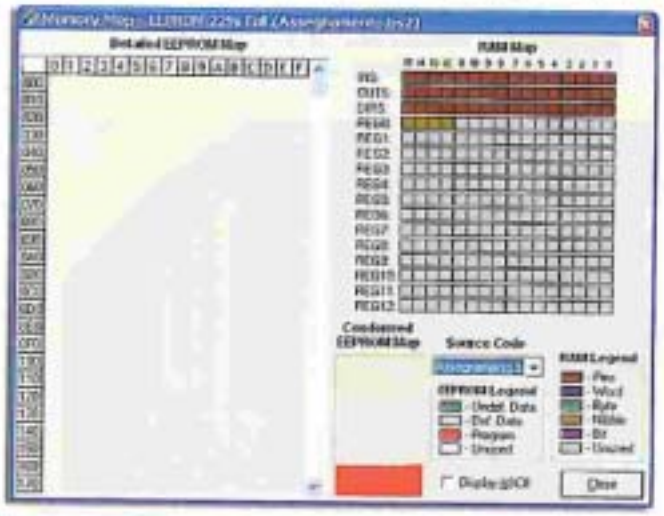
**Sotto. Visione d'insieme delle due ruote e dei servomotori sostituiti agli attuali motori a spazzola, come risulterà dalle fasi di montaggio degli allegati ai prossimi tre fascicoli.**

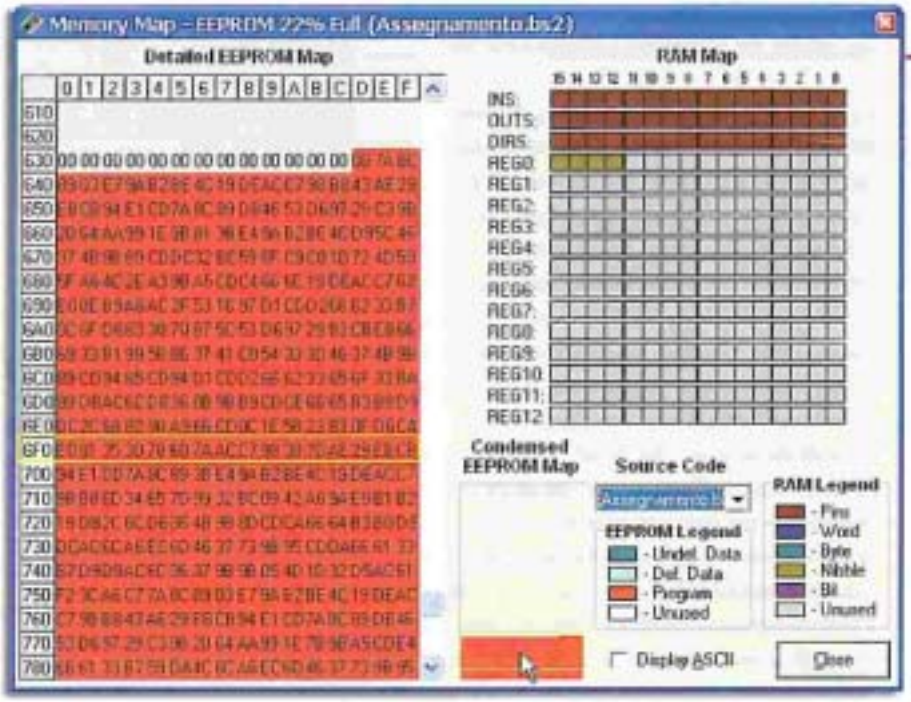


## LE FASI DI PROGRAMMAZIONE

*Abbiamo già visto come, all'interno di un programma, si dichiarano e funzionano le subroutine (a pag.68). Sappiamo anche che il loro utilizzo fa risparmiare spazio in memoria. Vediamo ora esattamente come e perché. Grazie alla Memory Map (già vista a pag. 63), possiamo infatti visualizzare lo spazio occupato da un programma nella memoria EEPROM e confrontarlo con quello occupato dallo stesso programma, scritto però con l'introduzione di una subroutine.*


**Per semplicità, riprendiamo il programma di pag. 80 (riproposto a destra), che riassume i metodi di assegnamento e la visualizzazione delle variabili. Ricarica o riscrivi il programma nell'area di editing, quindi premi il pulsante Memory Map della barra degli strumenti (indicato dal puntatore del mouse): apparirà la finestra Memory Map (in basso). Come sai, lo schema (*map*) a destra riguarda la memoria RAM del microcontrollore: in questo caso, oltre ai registri relativi alle porte logiche (INS, OUTS e DIRS), è rappresentata (in REG0) una sola variabile (relativa alla *x* dichiarata nel programma) di tipo nibble (identificato dal colore dei quattro quadratini, in base alla RAM Legend). Lo schema a sinistra (*Detailed EEPROM Map*, cioè 'Mappa Dettagliata della EEPROM'), riguarda invece la memoria EEPROM. Come ricorderai, mentre la RAM è una memoria cosiddetta 'volatile' (che, cioè, cancella i dati ogni volta che manca l'alimentazione), la EEPROM mantiene invece i dati anche in mancanza di alimentazione. Per questo, il codice di un programma viene sempre memorizzato nella EEPROM, mentre i valori delle variabili vengono, di volta in volta, caricati nella RAM. La memoria EEPROM di cui è dotato il tuo robot ha una capacità di 2048 byte (2KByte) e memorizza le istruzioni del listato (scritte in PBASIC) 'tradotte' in codice binario, cioè l'unico linguaggio comprensibile al microcontrollore. Poiché la mappa, di fatto, rappresenta lo schema della memoria byte per byte, può essere utile consultarla per verificare lo spazio che il programma occupa all'interno della memoria. Vediamo dunque come è organizzata la mappa della EEPROM. Innanzitutto, la finestra (a sinistra) ne fornisce due versioni: quella dettagliata, appunto, e la cosiddetta *Condensed EEPROM Map* (visibile in basso, al centro della finestra). Quest'ultima è una sorta di sintesi (o 'condensato') dell'altra e mostra immediatamente lo spazio occupato dal programma (porzione in rosso), in proporzione a quello totale a disposizione (porzione chiara); inoltre, funziona anche come barra di scorrimento (analogamente a quella posta a lato della versione *Detailed*): infatti, cliccandoci sopra con il puntatore del mouse (visibile nell'immagine della pagina a destra), si può scorrere il contenuto della mappa, fino a visualizzare (nella *Detailed Map*) la porzione di memoria effettivamente occupata dal programma (in rosso). Vediamo dunque la struttura dettagliata della mappa: ogni quadratino corrisponde a un byte che ha un indirizzo (posizione nella EEPROM) e un contenuto (tradotto in bit).**



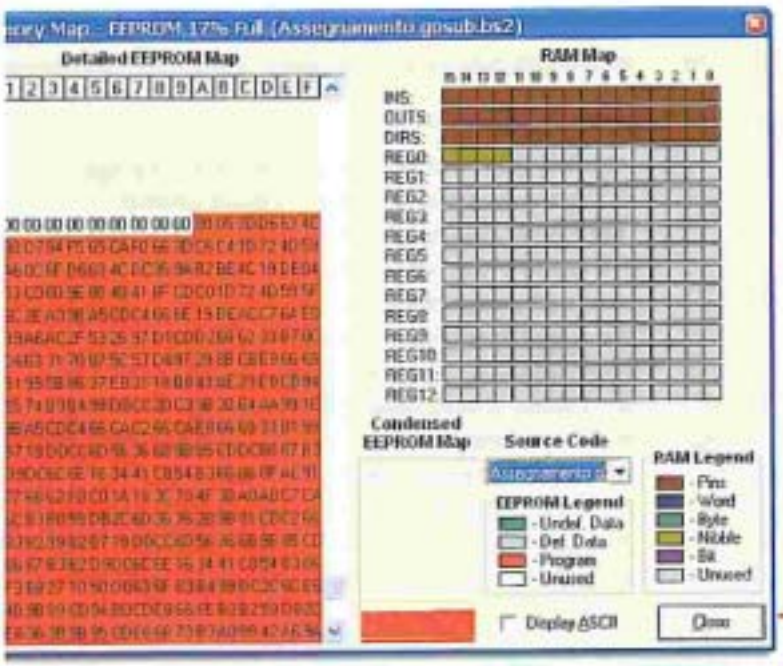



Ogni riga della mappa dettagliata rappresenta un blocco di 16 byte i cui indirizzi dipendono dalla combinazione dei numeri esadecimali posti nella prima colonna a sinistra e di quelli della prima riga in alto: il numero della prima colonna corrisponde all'indirizzo del primo byte della riga (incolonnato, infatti, sotto lo 0); gli indirizzi degli altri byte della stessa riga mantengono invariate le prime due cifre e sostituiscono all'ultima il numero progressivo sotto cui sono incolonnati (rispetto alla prima riga in alto: 0, 1, 2... F). Vediamo, ad esempio, la riga 6F0 (riquadrata in giallo): il primo byte della riga (il cui contenuto è E0) ha come indirizzo 6F0, il che significa che è il 1777° byte della memoria (applicando le regole di traduzione dal sistema esadecimale al decimale, viste a pag. 79, risulta che $6F0 = 6*16^2 + 15*16^1 + 0*16^0 = 1536 + 240 + 0 = 1776$; poiché lo 0 è il 1° byte, 1 è il 2°, 2 è il 3°... 1776 sarà il 1777° byte); i successivi 15 byte (contenenti 81, 35... CB), invece, avranno come indirizzo rispettivamente 6F1, 6F2... 6FF. Il contenuto di ogni byte è visualizzato con un numero esadecimale a due cifre (E0, 81, 35... CB, ad esempio) che, pur potendo codificare i numeri da 0 a 255 esattamente come gli 8 bit di ciascun byte, è però più adatto a rappresentarli in modo compatto. Come avrai notato, il programma occupa solo una parte della memoria, il 22% (indicato nella barra dei titoli della finestra). Come vedremo in seguito, la parte di memoria che rimane libera (nell'esempio, dalla riga 620 in su) potrà essere utilizzata per memorizzare altri dati che debbano essere conservati anche in mancanza di alimentazione. Durante il processo di memorizzazione, la EEPROM viene scritta a blocchi di 16 byte (un'intera riga alla volta), a partire dal fondo: nel caso in cui alcuni byte di una riga siano superflui, essi vengono comunque settati, ma al valore 0. Nella riga 630, ad esempio, solo gli ultimi 3 byte a destra (in rosso) contengono parte del programma; gli altri 13, invece, sono inutilizzati e settati a 0 (00). Ora puoi chiudere la finestra Memory Map, premendo il pulsante Close, posto in basso a destra. Tornando invece al listato del programma (nella pagina a sinistra), possiamo notare che, nella sua struttura, compaiono delle ripetizioni: dopo ciascuno tipo di assegnamento, infatti, ricorre la stessa istruzione di debug (debug DEC ? x, BIN ? x, HEX ? x, CR) per la visualizzazione del nome (?) e del valore della variabile (x), secondo i diversi formati (decimale, binario ed esadecimale). Ripetendosi più volte sempre identiche, queste istruzioni di debug possono essere 'tradotte' in un'unica subroutine e, di conseguenza, 'alleggerire' il programma.

*Vogliamo soffermarci ora sui vantaggi che derivano dall'utilizzo di una o più subroutine all'interno di un programma, in particolare per quanto riguarda la sua modificabilità, vale a dire la possibilità di intervenire sul codice dopo averlo scritto, apportando cambiamenti quali, ad esempio, la correzione di un eventuale errore rilevato in fase di esecuzione o, addirittura, la variazione della struttura stessa del programma. Un compito più o meno facile, a seconda di come il codice è stato scritto: un conto, infatti, è doverlo stravolgere completamente o doverne riscrivere una gran parte, un altro è invece poter intervenire in precise e ben riconoscibili porzioni. Quest'ultima possibilità dipende dall'aver applicato le buone norme di scrittura e dall'aver saputo scegliere l'impiego migliore delle varie istruzioni messe a disposizione dal linguaggio di programmazione. L'uso delle subroutine, in particolare, consente di ridurre, anche notevolmente, il codice scritto e di risparmiare spazio nella memoria EEPROM: un aspetto determinante quando si ha a che fare con programmi complessi, laddove quanto più spazio si risparmia in fase di scrittura, tanto più la memoria sarà disponibile ad accogliere altri dati.*

## LE FASI DI PROGRAMMAZIONE

La nuova versione (a destra) del programma precedente, prevede una subroutine (scritta alla fine del listato, dopo il programma principale o main) contenente l'istruzione di debug. Come ricorderai, il corpo della subroutine è delimitato dall'etichetta iniziale o dall'istruzione finale return, che consente di riprendere l'esecuzione del main dal punto in cui era stata 'invocata' (cioè chiamata) la subroutine stessa. Nel listato, la subroutine è invocata di volta in volta dall'istruzione gosub Visualizza (riquadrata in rosso) che, di fatto, ha sostituito tutte le occorrenze dell'istruzione di debug della versione precedente del listato. Al termine del main, prima della dichiarazione della subroutine, è stata inserita l'istruzione end, necessaria per terminare l'esecuzione del programma. Come già anticipato a pag. 68 (a proposito del loop), l'esecuzione delle istruzioni di un programma è prevalentemente sequenziale (salvo nei casi di istruzioni di salto). Le subroutine, sebbene scritte dopo il main, vengono di fatto invocate durante la sua esecuzione (con il gosub): dunque è necessario marcare esplicitamente la fine del main perché non interpreti in modo errato il corpo della subroutine come parte integrante della normale sequenza di istruzioni. Senza l'end, infatti, una volta eseguita la subroutine invocata dall'ultimo comando gosub Visualizza, il programma sarebbe incappato nuovamente nella subroutine, con la differenza che, stavolta, giunto all'istruzione return (ultima riga), non avrebbe più avuto un punto


preciso a cui tornare: la subroutine eseguita in sequenza al main, infatti, non risulterebbe invocata da alcun comando gosub. Non potendo decidere in modo univoco che cosa fare, il programma si azzererebbe, ricominciando da capo. Per evitare tale comportamento indesiderato, dunque, è fondamentale isolare (in questo caso con end) la parte dichiarativa delle subroutine dal resto del codice. Se ora riapri la finestra della Memory Map (a sinistra), puoi verificare (nella barra dei titoli) che questa versione del programma occupa solo il 17% dell'intera memoria, cioè molto meno del precedente. Immagina, dunque, lo spazio che verrebbe risparmiato in situazioni in cui la subroutine contenga più di una istruzione. Verifica cosa accade, ad esempio, inserendo un'opportuna subroutine nel programma per la fusione sensoriale (a pag. 86), al posto delle istruzioni ricorrenti.