Programmare robot Lego usando NQC

(Versione 3.03, 2 Ottobre 1999)

di Mark Overmars

Department of Computer Science
Utrecht University
P.O. Box 80.089, 3508 TB Utrecht
the Netherlands

Prefazione

I set robotici Lego MindStorms e CyberMaster sono dei fantastici nuovi giochi con i quali si possono costruire una grande varietà di robot, che possono essere programmati per eseguire operazioni di ogni sorta. Sfortunatamente, il software fornito con questi set, benché visualmente accattivante, è piuttosto limitato per quanto riguarda la funzionalità. Perciò, può essere utilizzato solo per eseguire semplici operazioni. Per poter usufruire a piena potenza dei robot, hai bisogno di un ambiente di programmazione diverso. NQC è un linguaggio di programmazione, realizzato da Dave Baum, progettato specificamente per i robot Lego. Se non hai mai scritto un programma, non ti preoccupare. NQC è veramente facile da usare e questo tutorial ti insegnerà ogni cosa. Attualmente, programmare i robot in NQC è molto più facile che programmare un normale computer, quindi questa è una chance per diventare un programmatore nella maniera più facile e divertente.

Per poter scrivere programmi ancor più facilmente, esiste RCX Command Center. Questa utility ti aiuta a scrivere i codici, a spedirli al robot, a far partire e fermare il robot. RCX Command Center lavora più o meno come un elaboratore testi, ma con qualche funzionalità in più. Questo tutorial userà RCX Command Center (versione 3.0 o superiore) come ambiente di sviluppo. Puoi scaricarlo gratis dal web all'indirizzo

http://www.cs.uu.nl/people/markov/lego/

RCX Command Center gira su PC Windows ('95, '98, NT). Assicurati di eseguire il software fornito con il set Lego almeno una volta, prima di usare RCX Command Center. Il software Lego installa infatti certi componenti usati da RCX Command Center. Il linguaggio NQC può anche essere usato su altre piattaforme. Puoi scaricarlo dal web all'indirizzo

http://www.enteract.com/~dbaum/lego/nqc/

Gran parte di questo tutorial può anche essere applicato ad altre piattaforme (usando NQC versione 2.0 o superiore), ad eccezione del fatto che si perdono alcuni strumenti e la colorazione della sintassi.

In questo tutorial presumo che tu abbia un set MindStorms. Molti argomenti si possono applicare anche ai set CyberMaster sebbene qualche funzionalità non sia disponibile per questi robot. Per esempio, i nomi dei motori sono diversi, quindi dovresti cambiare un poco gli esempi per farli funzionare.

Riconoscimenti

Vorrei ringraziare Dave Baum per aver sviluppato NQC. Inoltre, mille grazie a Kevin Saddi per aver scritto una prima versione della prima parte di questo tutorial.

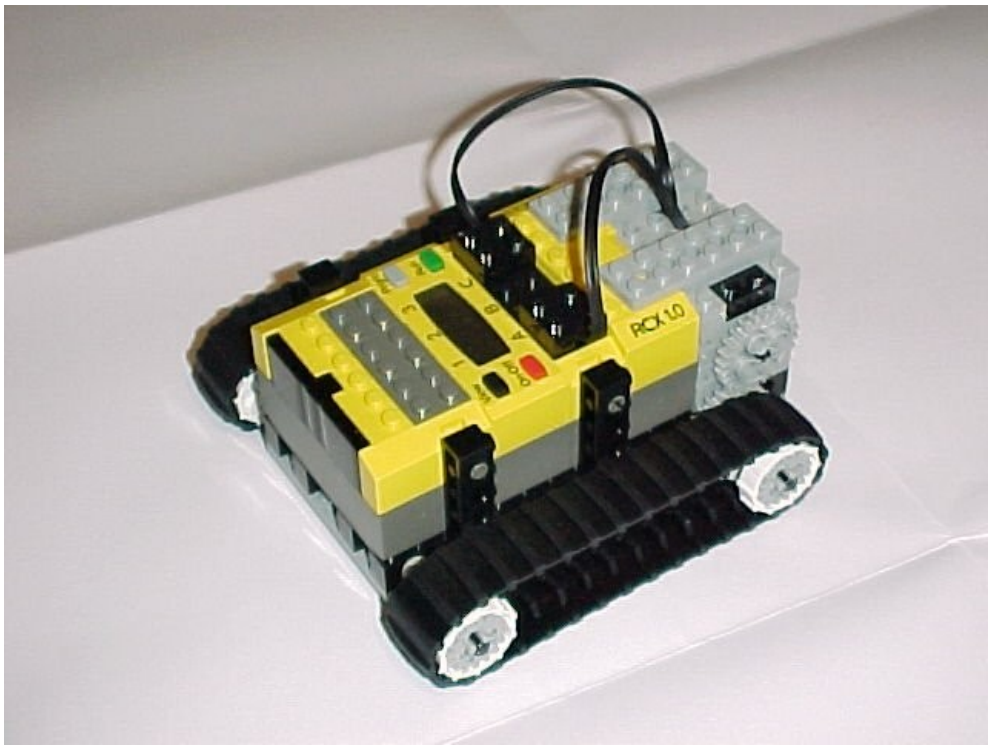
Indice

I. Scrivere il tuo primo programma

In questo capitolo ti mostrerò come scrivere un semplice programma. Vogliamo programmare un robot che si muova avanti per quattro secondi, quindi indietro per altri quattro secondi, e quindi si fermi. Non molto spettacolare, ma ti introdurrà nelle basi della programmazione e ti farà capire quanto ciò sia facile. Ma prima di poter scrivere un programma, abbiamo bisogno di un robot.

Costruire un robot

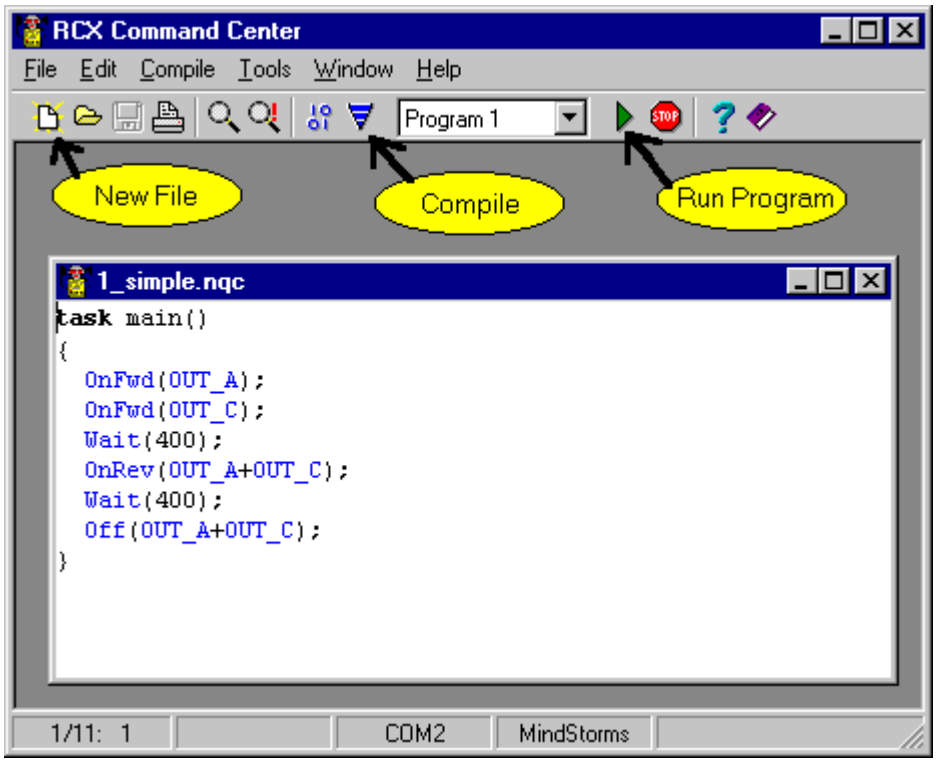
Il robot che useremo in questo tutorial è una versione del robot top secret descritto alle pagine 39-46 della constructopedia. Useremo solo il telaio base. Rimuoviamo tutta la parte frontale con le due braccia e il sensore al tatto. Inoltre, connettiamo i motori in maniera che i cavetti siano connessi all'RCX all'esterno. Questo è importante per il nostro robot affinché proceda nella giusta direzione. Ora dovrebbe essere più o meno così:



Fai anche attenzione che la porta a infrarossi sia connessa correttamente al tuo computer e che sia impostata sulle lunghe distanze. (Vorrai controllare infatti con il software del RIS che il robot stia funzionando nella maniera corretta.)

Eseguire RCX Command Center

Scriviamo i nostri programmi usando RCX Command Center, quindi eseguiamolo cliccando due volte sull'icona RcxCC. (Assumo che tu abbia già installato RCX Command Center. Se non è così, scaricalo dal sito web (vedi Prefazione), decomprimilo, e piazzalo in una directory a tuo piacere.) Il programma ti chiederà ora dove cercare il robot. Accendi quest'ultimo e premi **OK**. Il programma troverà (con tutta probabilità) automaticamente il robot. Apparirà ora l'interfaccia utente, come mostrato sotto (ma senza finestre).



L'interfaccia assomiglia a quella di un normale editor per testi, con i soliti menu e pulsanti per aprire file, salvarli, stamparli, modificarli, ecc. Ma ci sono anche alcuni comandi speciali per compilare e scaricare i programmi sul tuo robot e per ottenere informazioni da esso. Per il momento, questi possono anche essere ignorati.

Stiamo per scrivere un nuovo programma. Quindi premi il pulsante **New File** per far apparire una nuova finestra in bianco.

Scrivere il programma

Ora scrivi il seguente programma:

```
task main()
{
  OnFwd(OUT_A);
  OnFwd(OUT_C);
  Wait(400);
  OnRev(OUT_A+OUT_C);
  Wait(400);
  Off(OUT_A+OUT_C);
}
```

La prima volta potrebbe sembrare un poco complicato, quindi analizziamolo attentamente. I programmi in NQC consistono di task (compiti). Il nostro programma ha un solo task, chiamato `main`. Ogni programma deve avere un task chiamato `main`, che è quello direttamente eseguito dal robot. Approfondiremo i task nel capitolo VI. Un task consiste in un numero di istruzioni, anche chiamate statement. Prima e dopo questi statement vanno poste delle parentesi graffe che delimitano l'inizio e la fine del task. Ogni statement termina con un punto e virgola. In questa maniera è chiaro dove un'istruzione finisce e dove quella successiva inizia. Quindi, la struttura di un task dovrebbe apparire, in generale, così:

```
task main()
{
  statement1;
  statement2;
    …
}
```

Il nostro programma ha sei statement. Osserviamoli uno alla volta:

`OnFwd(OUT_A);`
Questa istruzione dice al robot di far partire la porta di output A, ovvero il motore connesso alla porta chiamata A sull'RCX, per muoversi avanti. Si muoverà alla velocità massima, a meno che questa non venga prima impostata. In seguito vedremo come fare ciò.

`OnFwd(OUT_C);`
Stessa istruzione, ma adesso facciamo partire il motore C. Dopo questi due statement, entrambi i motori stanno girando, ed il robot sta avanzando.

`Wait(400);`
Adesso dobbiamo aspettare per un po'. Questa istruzione ci dice di aspettare per quattro secondi. L'argomento, ovvero il numero tra parentesi, fornisce il numero di "battiti". Ogni battito corrisponde a 1/100 di secondo. Quindi puoi dire al programma quanto aspettare in maniera molto precisa. Ora per quattro secondi il programma non farà niente ed il robot continuerà ad avanzare.

`OnRev(OUT_A+OUT_C);`
Il robot è ora andato abbastanza lontano, quindi gli diciamo di andare nella direzione opposta, cioè all'indietro. Notare che possiamo settare entrambi i motori in una volta sola usando `OUT_A+OUT_C` come argomento. Potevamo combinare anche la prime due istruzioni alla stessa maniera.

`Wait(400);`
Aspettiamo ancora 4 secondi.

`Off(OUT_A+OUT_C);`
Ed infine spegniamo entrambi i motori.

Ecco il programma completo. Muove entrambi i motori in avanti per 4 secondi, quindi indietro per altri 4 secondi, e alla fine li spegne.

Avrai probabilmente notato la presenza dei colori durante la digitazione del programma. Appaiono automaticamente. Tutto ciò che è in blu è un comando per il robot, o un indicazione del motore o una qualche altra cosa che il robot conosce. La parola **`task`** è in grassetto poiché è una parola importante (riservata) in NQC. In seguito vedremo altre parole importanti che appaiono in grassetto. I colori sono utili per scoprire se sono stati commessi errori durante la digitazione.

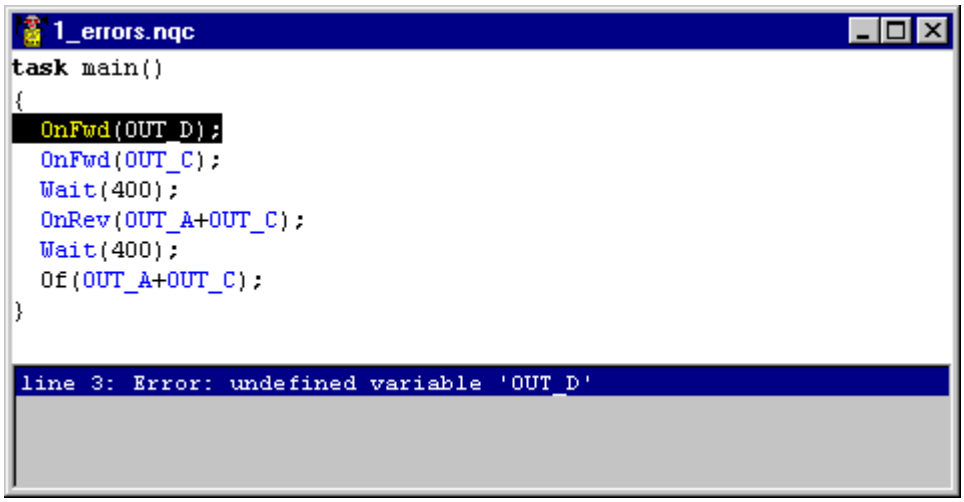
Esecuzione del programma

Una volta scritto il programma, Bisogna compilarlo (ovvero, cambiarlo in codice che il robot possa capire ed eseguire) ed inviarlo al robot usando il collegamento ad infrarossi (ciò viene chiamato "scaricare" il programma). C'è un pulsante che esegue tutto ciò (osserva la figura sopra). Premilo e, se non hai commesso errori durante la digitazione, il programma sarà correttamente compilato e scaricato al robot. (Se ci sono errori nel programma, ti verrà comunicato; vedi sotto.)

Ora puoi eseguire il programma. Per far ciò, premi il bottone verde sul tuo robot o, più facilmente, premi il pulsante run (vedi figura sopra). Il robot fa ciò che ti aspettavi? Se no, è probabile che i cavetti siano connessi in maniera errata.

Errori nel tuo programma

Durante la digitazione dei programmi c'è una buona possibilità che tu commetta qualche errore. Il compilatore nota gli errori e te li segnala nella parte bassa della finestra, come nella seguente figura:



Il primo errore viene selezionato automaticamente (abbiamo sbagliato il nome del motore). Quando sono presenti più errori, puoi cliccare sui messaggi per poterli vedere. Nota che spesso gli errori all'inizio del programma possono causare altri errori in seguito. Quindi è meglio iniziare a correggere i primi errori e cercare di compilare il programma di nuovo. Nota inoltre che la colorazione della sintassi aiuta a prevenire gli errori. Per esempio, sull'ultima riga abbiamo scritto `Of` invece di `Off`. Poiché questo è un comando sconosciuto, esso non viene colorato in blu.

Ci sono anche errori che il compilatore non può trovare. Se avessimo scritto `OUT_B` ciò non sarebbe stato notato poiché il motore esiste (anche se non viene usato nel robot). Quindi se il robot mostra comportamenti inaspettati, ci sarà con tutta probabilità qualcosa di sbagliato nel programma.

Cambio della velocità

Come avrai notato, il robot si è mosso piuttosto velocemente. Per default infatti, il robot si muove il più velocemente possibile. Per cambiare la velocità, puoi usare l'istruzione `SetPower()`. La potenza è un numero compreso tra 0 e 7. 7 è il più veloce, 0 il più lento (ma il robot continuerà a muoversi). Ecco una nuova versione del nostro programma nella quale il robot si muove lentamente:

```
task main()
{
  SetPower(OUT_A+OUT_C,2);
  OnFwd(OUT_A+OUT_C);
  Wait(400);
  OnRev(OUT_A+OUT_C);
  Wait(400);
  Off(OUT_A+OUT_C);
}
```

Ricapitolazione

In questo primo capitolo hai scritto il tuo primo programma in NQC, usando RCX Command Center. Dovresti ormai sapere come scrivere un programma, come scaricarlo sul robot e come far eseguire il programma al robot. RCX Command Center può fare molte altre cose. Per scoprirle, leggi la documentazione con esso fornita. Questo tutorial avrà soprattutto a che fare con il linguaggio NQC e menzionerà soltanto alcune caratteristiche di RCX Command Center.

Hai anche imparato alcuni importanti aspetti del linguaggio NQC. Prima di tutto, hai imparato che ogni programma ha un task chiamato `main` che è sempre eseguito dal robot. Inoltre hai imparato i quattro comandi più importanti dei motori: `OnFwd()`, `OnRev()`, `SetPower()` e `Off()`. Infine hai imparato l'istruzione `Wait()`.

II. Un programma più interessante

Il nostro primo programma non è stato molto spettacolare. Vediamo quindi di renderlo un poco più interessante. Lavoreremo passo per passo, introducendo alcune importanti caratteristiche del nostro linguaggio NQC.

Sterzare

Puoi far svoltare il tuo robot arrestando o cambiando la direzione di uno dei due motori. Eccoti un esempio. Copialo, scaricalo sul robot e fallo partire. Dovrebbe avanzare per un po' e quindi eseguire una curva di 90 gradi a destra.

```
task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(100);
  OnRev(OUT_C);
  Wait(85);
  Off(OUT_A+OUT_C);
}
```

Puoi anche provare ad utilizzare un numero diverso da 85 nel secondo `Wait()` per eseguire una precisa curva ad angolo retto. Questo infatti dipende dal tipo di superficie su cui il robot lavora. Piuttosto che cambiare continuamente un numero però, sarebbe più facile adoperare un nome per questo. In NQC puoi definire valori costanti (fissi), come mostrato dal seguente programma.

```
#define MOVE_TIME   100
#define TURN_TIME    85

task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(MOVE_TIME);
  OnRev(OUT_C);
  Wait(TURN_TIME);
  Off(OUT_A+OUT_C);
}
```

Le prime due linee definiscono due costanti. Queste possono essere poi usate all'interno del programma. È bene definire costanti per due ragioni: rendono il programma più leggibile e facilitano il cambio dei valori. Nota che RCX Command Center assegna al comando define un colore proprio. Come vedremo nel capitolo VI, puoi anche definire elementi che non siano costanti.

Ripetizione di comandi

Proviamo ora a scrivere un programma che faccia fare al robot dei quadrati. Fare un quadrato vuol dire: avanzare, svoltare di 90 gradi, avanzare ancora, svoltare di 90 gradi, ecc. Potremmo ripetere questo pezzo di codice quattro volte, ma l'intera operazione può essere definita molto più facilmente con il comando **`repeat`**.

```
#define MOVE_TIME   100
#define TURN_TIME    85

task main()
{
  repeat(4)
  {
    OnFwd(OUT_A+OUT_C);
    Wait(MOVE_TIME);
    OnRev(OUT_C);
    Wait(TURN_TIME);
  }
  Off(OUT_A+OUT_C);
}
```

Il numero che segue a **repeat**, scritto tra parentesi, indica quante volte qualcosa deve essere ripetuto. Le istruzioni da ripetersi sono poste tra parentesi graffe, come le istruzioni di un task. Nota che, nel programma sopra, abbiamo indentato le istruzioni. Ciò non è necessario, ma rende il codice più leggibile.

Come esempio finale, facciamo fare al robot dieci quadrati. Ecco il programma:

```
#define MOVE_TIME   100
#define TURN_TIME    85

task main()
{
  repeat(10)
  {
    repeat(4)
    {
      OnFwd(OUT_A+OUT_C);
      Wait(MOVE_TIME);
      OnRev(OUT_C);
      Wait(TURN_TIME);
    }
  }
  Off(OUT_A+OUT_C);
}
```

Abbiamo così un comando **repeat** dentro l'altro. Ciò prende il nome di comando repeat nidificato. Puoi nidificare quanti comandi repeat vuoi. Osserva attentamente le parentesi graffe e l'indentazione usate nel programma. Il task inizia alla prima parentesi e finisce all'ultima. Il primo comando repeat inizia alla seconda e finisce alla quinta. Il secondo comando repeat, nidificato, inizia alla terza e finisce alla quarta. Come puoi vedere le parentesi graffe sono sempre presenti in coppia, e indentiamo sempre il pezzo di codice compreso tra ognuna di queste coppie.

Aggiungere commenti

Per rendere il tuo programma ancora più leggibile, è bene aggiungerci qualche commento. Ogni volta che poni `//` su una riga, il resto di questa sarà ignorata e considerata commento. Un commento più lungo può adoperare `/*` e `*/` per poter stare su più righe. I commenti in RCX Command Center sono colorati in verde. Il programma può quindi diventare così:

```
/*  10 SQUARES

    di Mark Overmars

Questo programma fa fare al robot 10 quadrati
*/

#define MOVE_TIME   100     // Tempo per avanzare
#define TURN_TIME    85     // Tempo per svoltare a 90 gradi

task main()
{
  repeat(10)                // Esegui 10 quadrati
  {
    repeat(4)
    {
      OnFwd(OUT_A+OUT_C);
      Wait(MOVE_TIME);
      OnRev(OUT_C);
      Wait(TURN_TIME);
    }
  }
  Off(OUT_A+OUT_C);         // Ora spegni i motori
}
```

Ricapitolazione

In questo capitolo hai imparato l'uso del comando **`repeat`** e come porre commenti nel codice. Hai anche imparato come utilizzare istruzioni nidificate. Con quello che ormai conosci puoi costruire robot che si muovano su ogni tipo di percorso. Può rivelarsi un buon esercizio provare a scrivere alcune variazioni dei programmi di questo capitolo prima di continuare con i prossimi.

III. Uso delle variabili

Le variabili sono tra gli aspetti più importanti di ogni linguaggio di programmazione. Esse rappresentano locazioni di memoria nelle quali possiamo memorizzare un valore. Possiamo usare questo valore in diverse occasioni e, soprattutto, possiamo cambiarlo. Permettetemi di mostrarvi l'uso delle variabili con un esempio.

Movimento a spirale

Assumiamo di voler adattare il programma presentato in precedenza per far eseguire al robot delle spirali. Ciò può essere realizzato aumentando gradualmente il tempo di attesa. Ovvero, vogliamo aumentare il valore di `MOVE_TIME` ogni volta. Ma come possiamo fare ciò? `MOVE_TIME` è una costante, e le costanti non possono essere modificate. Abbiamo piuttosto bisogno di una variabile. Le variabili in NQC vengono definite molto facilmente. Puoi averne al massimo 32, e puoi dare ad ognuna di esse un nome diverso. Ecco il programma per la spirale.

```
#define TURN_TIME   85

int move_time;              // definisce una variabile

task main()
{
  move_time = 20;           // imposta il valore iniziale
  repeat(50)
  {
    OnFwd(OUT_A+OUT_C);
    Wait(move_time);        // usa la variabile per l'attesa
    OnRev(OUT_C);
    Wait(TURN_TIME);
    move_time += 5;         // incrementa la variabile
  }
  Off(OUT_A+OUT_C);
}
```

Le righe degne di nota sono indicate dai commenti. Prima di tutto definiamo una variabile con la parola chiave **`int`** seguita dal nome che scegliamo per essa. (In genere si usano lettere minuscole per le variabili e maiuscole per le costanti, ma ciò non è necessario.) Il nome deve iniziare con una lettera ma può contenere numeri e caratteri di separazione inferiore. Non sono ammessi altri simboli. (Lo stesso vale per costanti, task, ecc.) La parola **`int`** sta per integer. Può infatti contenere solo numeri interi. Nella seconda riga contrassegnata gli assegniamo un valore di 20. Da questo momento, ogni volta che usi la variabile, essa sta per 20. Ora segui il comando repeat nel quale usiamo la variabili per indicare il tempo di attesa e, alla fine dell'iterazione nota come aumentiamo il valore della variabile di 5. Quindi la prima volta il robot attende per 20 battiti, la seconda per 25, la terza per 30, ecc.

Oltre ad aumentare il valore di una variabile possiamo anche moltiplicarlo usando `*=`, diminuirlo con `-=` e dividerlo con `/=`. (Nota che per la divisione il risultato viene arrotondato.) Puoi anche usare più variabili insieme, e scrivere operazioni più complicate. Ecco alcuni esempi:

```
int aaa;
int bbb, ccc;

task main()
{
  aaa = 10;
  bbb = 20 * 5;
  ccc = bbb;
  ccc /= aaa;
  ccc -= 5;
  aaa = 10 * (ccc + 3);  // aaa e' ora uguale a 80
}
```

Nota che possiamo anche definire più variabili sulla stessa riga. Volendo, nella prima riga potevamo combinarle tutte e tre.

Numeri casuali

In ogni programma abbiamo finora definito esattamente ciò che il robot doveva fare. Ma le cose si fanno più interessanti quando non siamo a conoscenza di ciò che il robot stia per fare. Vogliamo un po' di casualità nel movimento. In NQC puoi creare numeri casuali. Il seguente programma usa questi per far avanzare il robot in maniera casuale. Avanza per un numero casuale di secondi e quindi compie una curva casuale.

```
int move_time, turn_time;

task main()
{
  while(true)
  {
    move_time = Random(60);
    turn_time = Random(40);
    OnFwd(OUT_A+OUT_C);
    Wait(move_time);
    OnRev(OUT_A);
    Wait(turn_time);
  }
}
```

Il programma definisce due variabili, e quindi assegna loro valori casuali. `Random(60)` significa un numero casuale tra 0 e 60 (0 e 60 inclusi). Ogni volta i numeri saranno così differenti. (Nota che in questo caso potremmo anche evitare l'uso delle variabili scrivendo ad esempio `Wait(Random(60))`.)

Hai potuto anche osservare in questo esempio un nuovo tipo di iterazione. Piuttosto che usare il comando repeat abbiamo scritto **`while(true)`**. Il comando **`while`** ripete le proprie istruzioni fintanto che la condizione tra parentesi risulta vera. La speciale parola **`true`** è sempre vera, quindi le istruzioni tra le parentesi graffe saranno eseguite *in aeternum*. Approfondiremo il comando while nel capitolo IV.

Ricapitolazione

In questo capitolo hai imparato l'uso delle variabili. Le variabili sono molto utili, ma a causa delle restrizioni imposte dai robot sono un po' limitate. Puoi definirne solo 32 e possono immagazzinare solo valori interi. Ma per molti compiti ciò è più che sufficiente.

Hai anche scoperto come creare numeri casuali, cosicché puoi dare al tuo robot un comportamento imprevedibile. Infine abbiamo visto l'uso del comando while per creare iterazioni infinite.

IV. Strutture di controllo

Nei capitoli precedenti abbiamo visto i comandi repeat e while. Questi comandi regolano l'ordine con cui le altre istruzioni del programma vengono eseguite. Sono perciò chiamate "strutture di controllo". In questo capitolo vedremo qualche altro comando di questo tipo.

Il comando if

A volte vuoi che una parte del programma venga eseguita solo in certe situazioni. In questi casi viene usato il comando if. Eccoti un esempio. Modifichiamo ancora il nostro programma, ma con una variazione. Vogliamo che il robot proceda diritto e quindi giri a sinistra o a destra. Per far ciò abbiamo nuovamente bisogno di numeri casuali. Prendiamo un numero casuale tra 0 e 1, cioè, o 0 o 1. Se viene 0 giriamo a destra; altrimenti a sinistra. Ecco il programma:

```
#define MOVE_TIME   100
#define TURN_TIME    85

task main()
{
  while(true)
  {
    OnFwd(OUT_A+OUT_C);
    Wait(MOVE_TIME);
    if (Random(1) == 0)
    {
      OnRev(OUT_C);
    }
    else
    {
      OnRev(OUT_A);
    }
    Wait(TURN_TIME);
  }
}
```

Il comando if potrebbe assomigliare al comando while. Se la condizione tra parentesi è vera viene eseguita la sezione di codice tra le parentesi graffe. Altrimenti, viene eseguita l'altra parte tra parentesi graffe, che segue la parola chiave **`else`**. Osserviamo un po' meglio la condizione qui usata. Essa è `Random(1) == 0`. Vuol dire che `Random(1)` deve essere uguale a 0 per rendere la condizione vera. Potresti chiederti perché usiamo == piuttosto che =. Il motivo è: per distinguerlo dall'operatore che assegna un valore ad una variabile. Puoi confrontare i valori in diversi modi. Ecco alcuni tra i più comuni:

==	uguale a
<	minore di
<=	minore o uguale a
>	maggiore di
>=	maggiore o uguale a
!=	diverso da

Puoi combinare le condizioni usando `&&`, che significa "e", o `||`, che significa "o". Ecco alcuni esempi di condizioni:

`true`	sempre vero
`false`	sempre falso
`ttt != 3`	vero se ttt è diverso da 3
`(ttt >= 5) && (ttt <= 10)`	vero se ttt è compreso tra 5 e 10
`(aaa == 10) || (bbb == 10)`	vero se aaa, o bbb, o entrambe sono uguali a 10

Nota che il comando if è formato da due parti. La parte subito dopo la condizione, che viene eseguita quando la condizione è vera, e la parte dopo else, che viene eseguita quando la condizione è falsa. La parola chiave else e le istruzioni che ad essa seguono sono opzionali. Quindi puoi anche tralasciarle se non devi fare niente quando la condizione sia falsa.

Il comando do

Esiste un'altra struttura di controllo, il comando do. Esso ha la seguente forma generale:

```
do
{
  istruzioni;
}
while (condizione);
```

Le istruzioni tra le parentesi graffe dopo il do sono eseguite fintanto che la condizione è vera. La condizione ha poi la stessa forma che si applica al comando if. Eccoti un esempio. Il robot va a passeggio per 20 secondi e quindi si ferma.

```
int move_time, turn_time, total_time;

task main()
{
  total_time = 0;
  do
  {
    move_time = Random(100);
    turn_time = Random(100);
    OnFwd(OUT_A+OUT_C);
    Wait(move_time);
    OnRev(OUT_C);
    Wait(turn_time);
    total_time += move_time; total_time += turn_time;
  }
  while (total_time < 2000);
  Off(OUT_A+OUT_C);
}
```

Nota che in questo esempio abbiamo piazzato due istruzioni su una sola riga. Ciò è infatti permesso. Puoi porre quante istruzioni vuoi su una sola riga (sempre che tra esse siano presenti i punto e virgola). Per la leggibilità del programma però, ciò è spesso una cattiva idea.

Nota inoltre che il comando do si comporta pressoché allo stesso modo del comando while. Nel comando while però la condizione viene controllata prima di eseguire le istruzioni, mentre nel comando do la condizione è controllata alla fine. Con il comando while, le istruzioni possono non venire mai eseguite, mentre con il comando do esse vengono eseguite almeno una volta.

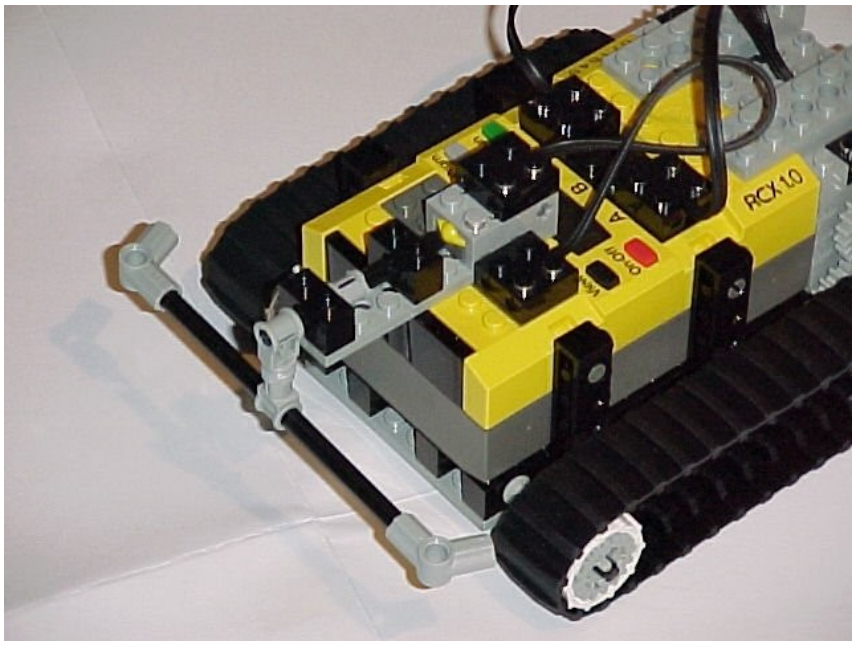
Ricapitolazione

In questo capitolo abbiamo visto due nuove strutture di controllo: il comando if ed il comando do. Insieme con repeat e while essi rappresentano i comandi che controllano il flusso del programma. È molto importante che tu comprenda il loro utilizzo. Quindi sarebbe consigliabile provare a scrivere qualche breve programma prima di continuare.

Inoltre, abbiamo visto che è possibile porre più istruzioni su di una sola riga.

V. I sensori

Uno degli aspetti più interessanti dei robot Lego è che puoi usare dei sensori per reagire a determinati stimoli. Prima che ti possa insegnare come fare ciò, è necessario modificare il nostro robot aggiungendo un sensore. Per far ciò, applichiamo la costruzione rappresentata nella figura 4 a pagina 28 della constructopedia. Puoi realizzarla un poco più larga, facendo diventare il tuo robot più o meno così:



Connetti il sensore alla porta di input 1 sull'RCX.

Aspettare un sensore

Partiamo con un semplice programma con il quale il robot avanzi finché non urta qualcosa. Ecco:

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  OnFwd(OUT_A+OUT_C);
  until (SENSOR_1 == 1);
  Off(OUT_A+OUT_C);
}
```

Qui ci sono due righe degne di nota. La prima riga dice al robot che tipo di sensore stiamo usando. `SENSOR_1` è il numero della porta alla quale è connesso il sensore. Le altre due porte di input sono chiamate `SENSOR_2` e `SENSOR_3`. `SENSOR_TOUCH` indica che questo è un sensore al tatto. Per il sensore alla luce avremmo usato `SENSOR_LIGHT`. Dopo aver specificato il tipo di sensore, il programma accende entrambi i motori e il robot comincia a procedere. La terza riga rappresenta una costruzione molto utile. Essa attende finché la condizione tra le parentesi non diventi vera. Questa condizione dice che il valore di `SENSOR_1` deve essere 1, ovvero che il sensore deve essere stato premuto. Se il sensore non viene premuto, il suo valore è 0. Quindi questa istruzione attende che il sensore venga premuto. Infine, spegniamo i motori, e il robot ha finito i suoi compiti.

Reagire al tatto

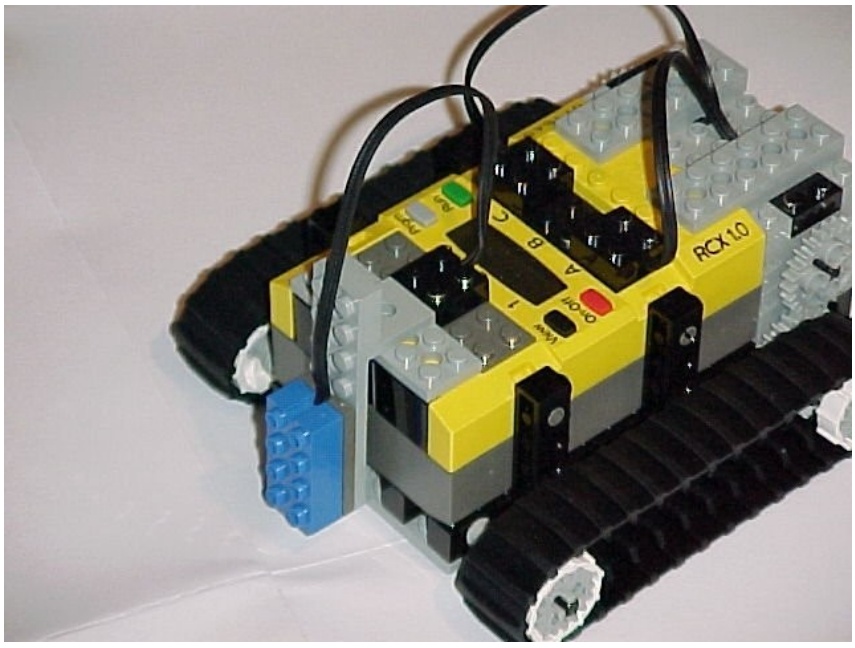
Cerchiamo ora di realizzare un robot che eviti gli ostacoli. Quando il robot urta un oggetto, lo facciamo indietreggiare, voltare, e quindi nuovamente avanzare. Ecco il programma:

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  OnFwd(OUT_A+OUT_C);
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      OnRev(OUT_A+OUT_C); Wait(30);
      OnFwd(OUT_A); Wait(30);
      OnFwd(OUT_A+OUT_C);
    }
  }
}
```

Come nell'esempio precedente, iniziamo definendo il tipo di sensore. Quindi il robot comincia ad avanzare. Nell'infinito ciclo while, controlliamo costantemente lo stato del sensore e, se è stato premuto, indietreggiamo per 1/3 di secondo, giriamo a destra per 1/3 di secondo, e quindi riprendiamo ad avanzare.

Sensori alla luce

Oltre ai sensori al tatto, nel set MindStorms è compreso un sensore alla luce. Esso misura la quantità di luce in una particolare direzione. Inoltre, emette luce propria. In questo modo è possibile puntare il sensore in una direzione e distinguere i diversi gradi di intensità dell'oggetto a cui stiamo puntando. Questo particolare è utile, ad esempio, per creare un robot che segua una linea sul pavimento. Ciò è quello che stiamo per fare nel prossimo esempio. Prima però, abbiamo bisogno di attaccare il sensore alla luce al robot in maniera che stia nel mezzo della parte anteriore, e punti in basso. Connettilo alla porta di input 2. Dovresti ottenere un veicolo del genere:



Abbiamo inoltre bisogno del circuito da gara fornito con il RIS (Il grande foglio di carta con un tracciato segnato sopra). L'idea è ora che il robot controlli che il sensore alla luce stia posizionato sul tracciato. Se l'intensità della luce aumenta, il sensore è fuori tracciato e dobbiamo cambiare direzione. Ecco un semplice programma che fa procedere il robot sul tracciato in direzione oraria.

```
#define THRESHOLD 40

task main()
{
  SetSensor(SENSOR_2,SENSOR_LIGHT);
  OnFwd(OUT_A+OUT_C);
  while (true)
  {
    if (SENSOR_2 > THRESHOLD)
    {
      OnRev(OUT_C);
      until (SENSOR_2 <= THRESHOLD);
      OnFwd(OUT_A+OUT_C);
    }
  }
}
```

Il programma inizia col definire il sensore 2 come sensore alla luce. Quindi fa avanzare il robot ed entra in un ciclo infinito. Se il valore della luce sale oltre 40 (usiamo una costante in maniera che possa venire facilmente adattato, poiché questo valore dipende in gran parte dal livello di luce della stanza) invertiamo la direzione di un motore ed aspettiamo che il sensore vada di nuovo sul tracciato.

Come potrai osservare eseguendo il programma, il movimento non è molto elegante. Prova ad aggiungere un `Wait(10)` prima di **until** per far muovere meglio il robot. Nota che questo programma non è valido per muoversi in senso antiorario. Per poter far muovere il robot su qualsiasi tracciato sarebbe infatti necessario un programma molto più complesso.

Ricapitolazione

In questo capitolo abbiamo visto come usare i sensori al tatto e alla luce. Abbiamo anche scoperto il comando **until**, molto utile quando si lavora con i sensori.

Ti raccomando di scrivere un bel po' di programmi a questo punto. Ormai hai a disposizione tutti gli ingredienti per dare ai tuoi robot comportamenti abbastanza complicati. Per esempio, prova a mettere due sensori al tatto sul tuo robot, uno davanti sulla sinistra ed uno davanti sulla destra, e fagli evitare gli ostacoli. Inoltre, prova a costruire un robot che possa seguire una linea tracciata sul pavimento.

VI. Task e subroutine

Fino ad ora tutti i nostri programmi hanno posseduto un solo task. I programmi NQC però possono avere diversi task. È inoltre possibile piazzare frammenti di codice nelle cosiddette subroutine, per poterli poi riutilizzare in diversi punti del tuo programma. Usando task e subroutine i tuoi programmi diventeranno più facili da comprendere e più compatti. In questo capitolo tratteremo delle varie possibilità d'uso di questi importanti elementi della programmazione.

I task

Un programma NQC può avere al massimo 10 task. Ogni task ha un nome. Un solo task deve essere chiamato `main`, e viene eseguito automaticamente. Gli altri task vengono invece eseguiti solamente se sono chiamati con il comando start. Dopo ciò, entrambi i task saranno eseguiti contemporaneamente (quindi il task principale continuerà il suo corso). Un task in esecuzione può anche arrestare altri task usando il comando stop. Più tardi questo task può essere fatto ripartire, ma dall'inizio, e non dal punto in cui era stato fermato.

Lasciatemi mostrare l'uso dei task. Attacca il sensore al tatto sul tuo robot. Vogliamo scrivere un programma che guidi il robot disegnando quadrati, come in precedenza avevamo fatto. Ma quando urta un ostacolo, deve reagire. È difficile fare tutto ciò in un solo task, poiché il robot deve eseguire due compiti nello stesso momento: guidare (ovvero, accendere e spegnere i motori al momento giusto) e controllare i sensori. Quindi sarebbe meglio usare due task, uno che guidi, e l'altro che reagisca ai sensori. Ecco il programma.

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  start check_sensors;
  start move_square;
}

task move_square()
{
  while (true)
  {
    OnFwd(OUT_A+OUT_C); Wait(100);
    OnRev(OUT_C); Wait(85);
  }
}

task check_sensors()
{
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      stop move_square;
      OnRev(OUT_A+OUT_C); Wait(50);
      OnFwd(OUT_A); Wait(85);
      start move_square;
    }
  }
}
```

Il task principale imposta il tipo di sensore e quindi fa partire gli altri due task. Dopo ciò, esso ha portato a termine i suoi compiti. Il task `move_square` muove il robot a disegnare quadrati *in aeternum*. Il task `check_sensors` controlla se il sensore sia stato premuto. In tal caso, compie le seguenti operazioni: in primo luogo arresta il task `move_square`; questo è di grande importanza. `check_sensors` prende quindi il controllo del movimento del robot: lo fa indietreggiare e voltare. Quindi fa nuovamente partire `move_square` per continuare a disegnare quadrati.

È importante ricordare che i task che fai partire vengono eseguiti in parallelo. Ciò può portare a risultati imprevisti. Il capitolo Errore: sorgente del riferimento non trovata spiega questi problemi dettagliatamente e fornisce delle soluzioni.

Le subroutine

A volte hai bisogno dello stesso frammento di codice in diversi punti del tuo programma. In questi casi, puoi porre le istruzioni in una subroutine e dar loro un nome. Potrai quindi eseguire il codice semplicemente chiamandolo per nome da un task. NQC (per restrizioni dell'RCX) ti consente di usare fino ad 8 subroutine. Ecco un esempio.

```
sub turn_around()
{
  OnRev(OUT_C); Wait(340);
  OnFwd(OUT_A+OUT_C);
}

task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(100);
  turn_around();
  Wait(200);
  turn_around();
  Wait(100);
  turn_around();
  Off(OUT_A+OUT_C);
}
```

In questo programma abbiamo definito una subroutine che fa girare il robot sul suo centro. Il task principale chiama la subroutine tre volte. Nota che chiamiamo la subroutine scrivendo il suo nome seguito da parentesi. In questa maniera, assomiglia a molte delle istruzioni che abbiamo visto finora. Non ci sono parametri, quindi non dobbiamo inserire nulla tra le parentesi.

Sono ora d'uopo alcuni avvertimenti. Le subroutine sono un po' strane. Per esempio, una subroutine non può chiamare un'altra subroutine. Esse possono essere chiamate da task diversi ma ciò non è consigliabile. Ciò può infatti facilmente portare ad errori poiché la stessa subroutine potrebbe essere in esecuzione due volte nello stesso momento, chiamata da due task diversi. Ciò può provocare comportamenti non voluti. Inoltre, chiamando una subroutine da task diversi, a causa di una limitazione del firmware dell'RCX, non si possono usare espressioni complicate. Quindi, a meno che tu non conosca precisamente ciò che stai facendo, *non chiamare una subroutine da task diversi!*

Le inline function

Come detto sopra, le subroutine causano alcuni problemi. Il bello è che devono essere memorizzate una volta sola nell'RCX. Ciò salva molta memoria e, poiché l'RCX non dispone di molto spazio libero, sono di alta utilità. Quando però le subroutine sono brevi, è meglio usare delle inline function. Queste non sono memorizzate separatamente, ma copiate in ogni punto in cui vengono usate. Ciò porta ad occupare più memoria, ma problemi come quello di non poter usare espressioni complicate non si presentano. Inoltre, non c'è limite al numero di inline function che possono essere utilizzate.

La definizione e la chiamata di inline function avviene praticamente come per le subroutine. Basta usare la parola chiave **void** invece di **sub** (la parola **void** deriva dal linguaggio C). Quindi l'esempio sopra esposto, usando le inline function, appare così:

```
void turn_around()
{
  OnRev(OUT_C); Wait(340);
  OnFwd(OUT_A+OUT_C);
}

task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(100);
  turn_around();
  Wait(200);
  turn_around();
  Wait(100);
  turn_around();
  Off(OUT_A+OUT_C);
}
```

Le inline function hanno un altro vantaggio sulle subroutine. Possono avere argomenti. Questi possono essere usati per passare dei valori di certe variabili. Per esempio, nel programma qui sopra possiamo far sì che alla function venga passato il valore della durata, come mostrato di seguito:

```
void turn_around(int turntime)
{
  OnRev(OUT_C); Wait(turntime);
  OnFwd(OUT_A+OUT_C);
}

task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(100);
  turn_around(200);
  Wait(200);
  turn_around(50);
  Wait(100);
  turn_around(300);
  Off(OUT_A+OUT_C);
}
```

Gli argomenti vengono specificati tra le parentesi che seguono il nome della funzione. In questo caso specifichiamo che il nostro argomento è di tipo int (ci sono anche altre possibilità) e che il suo nome è `turntime`. Quando ci sono più argomenti, vanno separati con delle virgole.

Definizione di macro

C'è ancora un altro modo per dare ad un pezzo di codice un nome. In NQC si possono infatti definire delle macro (da non confondersi con le macro di RCX Command Center). Abbiamo visto in precedenza che possiamo definire costanti, usando #define, e dare loro un nome. Ma in realtà possiamo definire qualsiasi frammento di codice. Ecco ancora lo stesso programma, con l'uso però delle macro.

```
#define turn_around OnRev(OUT_C);Wait(340);OnFwd(OUT_A+OUT_C);

task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(100);
  turn_around;
  Wait(200);
  turn_around;
  Wait(100);
  turn_around;
  Off(OUT_A+OUT_C);
}
```

dopo il comando #define, la parola turn_around sta per il nome della macro. Ora, ovunque scriverai turn_around, sarà rimpiazzato con questo testo. Nota che esso deve essere tutto su una riga. (Ci sono modi per scrivere un comando #define su più righe, ma non è consigliabile.)

Il comando define può inoltre avere argomenti. Per esempio, possiamo porre il tempo della svolta come argomento alla macro. Ecco un esempio in cui definiamo quattro macro; una per avanzare, un a per indietreggiare, una per svoltare a sinistra e una per svoltare a destra. Ognuna ha due argomenti: la velocità ed il tempo.

```
#define turn_right(s,t)   SetPower(OUT_A+OUT_C,s);OnFwd(OUT_A);OnRev(OUT_C);Wait(t);
#define turn_left(s,t)    SetPower(OUT_A+OUT_C,s);OnRev(OUT_A);OnFwd(OUT_C);Wait(t);
#define forwards(s,t)     SetPower(OUT_A+OUT_C,s);OnFwd(OUT_A+OUT_C);Wait(t);
#define backwards(s,t)    SetPower(OUT_A+OUT_C,s);OnRev(OUT_A+OUT_C);Wait(t);

task main()
{
  forwards(3,200);
  turn_left(7,85);
  forwards(7,100);
  backwards(7,200);
  forwards(7,100);
  turn_right(7,85);
  forwards(3,200);
  Off(OUT_A+OUT_C);
}
```

È molto utile definire queste macro. Rendono il tuo programma più compatto e leggibile. Inoltre, puoi cambiare il codice più facilmente quando, ad esempio, cambi le connessioni dei motori.

Ricapitolazione

In questo capitolo abbiamo affrontato l'uso di task, subroutine, inline function, e macro. Questi hanno usi differenti. I task vengono normalmente eseguiti allo stesso momento e si prendono cura di diverse operazioni da essere eseguite contemporaneamente. Le subroutine sono utili quando diverse istruzioni devono essere utilizzate in diversi punti nello stesso task. Le inline function sono utili quando i frammenti di codice devono essere usati in diversi punti ed in diversi task, ma occupano più memoria. Infine, le macro sono utili per piccoli pezzi di codice che devono essere usati in più punti. Possono anche avere parametri.

Ora che sei giunto fin qui, dovresti avere tutte le conoscenze necessarie per far fare al tuo robot qualsiasi sorta di operazione. I prossimi capitoli di questo tutorial ti daranno nozioni concernenti applicazioni specifiche.

VII. Fare musica

L'RCX è dotato di uno speaker interno che può emettere suoni e addirittura suonare semplici motivi musicali. Ciò è particolarmente utile quando vuoi che l'RCX ti comunichi ciò che sta succedendo. Ma può essere anche divertente avere un robot che suona mentre svolge i suoi compiti.

Suoni predefiniti

Esistono sei suoni predefiniti nell'RCX, numerati da 0 a 5. Eccoli elencati qui di seguito:

0 Pressione di tasto
1 Beep beep
2 Frequenza decrescente
3 Frequenza in crescendo
4 Suono d'errore 'Buhhh'
5 Frequenza in veloce crescendo

Puoi eseguirli con l'istruzione `PlaySound()`. Ecco un breve programma che li esegue tutti.

```
task main()
{
  PlaySound(0); Wait(100);
  PlaySound(1); Wait(100);
  PlaySound(2); Wait(100);
  PlaySound(3); Wait(100);
  PlaySound(4); Wait(100);
  PlaySound(5); Wait(100);
}
```

Potresti chiederti il motivo dei comandi wait. La ragione è che l'istruzione che esegue il suono non aspetta che questo finisca. Passa subito all'istruzione seguente. L'RCX possiede un piccolo buffer nel quale può immagazzinare alcuni suoni, ma dopo un poco questo si riempie e avviene una perdita di informazioni. Ciò non è così importante per i suoni semplici, ma molto di più per le musiche, come vedremo in seguito.

Nota che l'argomento di `PlaySound()` deve essere un valore costante. Non puoi usare una variabile qui!

Musica vera

Per musiche più interessanti, NQC utilizza l'istruzione `PlayTone()`. Questo richiede due argomenti. Il primo è la frequenza, e il secondo la durata del suono (in battiti di 1/100 di secondo, come per il comando wait). Ecco un'utile tabella delle frequenze:

Nota	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
G#	52	104	208	415	831	1661	3322	
G	49	98	196	392	784	1568	3136	
F#	46	92	185	370	740	1480	2960	
F	44	87	175	349	698	1397	2794	
E	41	82	165	330	659	1319	2637	
D#	39	78	156	311	622	1245	2489	
D	37	73	147	294	587	1175	2349	
C#	35	69	139	277	554	1109	2217	
C	33	65	131	262	523	1047	2093	4186
B	31	62	123	247	494	988	1976	3951
A#	29	58	117	233	466	932	1865	3729
A	28	55	110	220	440	880	1760	3520

Come abbiamo precedentemente visto per i suoni, anche per la musica l'RCX non aspetta la fine della nota per continuare. Quindi, è meglio aggiungere delle istruzioni wait (un poco più lunghe) tra esse. Ecco un esempio:

```
task main()
{
  PlayTone(262,40);  Wait(50);
  PlayTone(294,40);  Wait(50);
  PlayTone(330,40);  Wait(50);
  PlayTone(294,40);  Wait(50);
  PlayTone(262,160); Wait(200);
}
```

Puoi facilmente creare delle melodie utilizzando l'RCX Piano, che è parte di RCX Command Center.

Se vuoi che l'RCX suoni mentre svolge altre operazioni, è meglio usare un task separato. Eccoti come esempio un programma piuttosto stupido con il quale l'RCX gira avanti e indietro, suonando in continuazione.

```
task music()
{
  while (true)
  {
    PlayTone(262,40);  Wait(50);
    PlayTone(294,40);  Wait(50);
    PlayTone(330,40);  Wait(50);
    PlayTone(294,40);  Wait(50);
  }
}

task main()
{
  start music;
  while(true)
  {
    OnFwd(OUT_A+OUT_C); Wait(300);
    OnRev(OUT_A+OUT_C); Wait(300);
  }
}
```

Ricapitolazione

In questo capitolo hai imparato a far suonare l'RCX. Hai anche visto come usare task separati per la musica.

VIII. Ancora sui motori

Esistono diversi comandi addizionali che puoi usare per controllare i motori in maniera più precisa. Li approfondiremo in questo capitolo.

Arresto gentile

Quando usi l'istruzione `Off()`, il motore si arresta immediatamente, usando il freno. In NQC è però anche possibile fermare i motori non usandolo. Per fare ciò, si usa l'istruzione `Float()`. A volte, questo metodo può risultare più adatto ai compiti del tuo robot. Eccoti un esempio. Dapprima il robot si ferma usando i freni; quindi, non usandoli. Prova a notare la differenza. (Con questo particolare robot, la differenza non sarà poi così netta, ma con robot più grandi, usare o meno i freni può essere di fondamentale importanza.)

```
task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(200);
  Off(OUT_A+OUT_C);
  Wait(100);
  OnFwd(OUT_A+OUT_C);
  Wait(200);
  Float(OUT_A+OUT_C);
}
```

Comandi avanzati

L'istruzione `OnFwd()` esegue effettivamente a sua volta due operazioni: accende il motore ed imposta la direzione in avanti. Lo steso vale per `OnRev()`: accende il motore ed imposta la direzione indietro. NQC possiede anche istruzioni per eseguire queste operazioni separatamente. Se vuoi cambiare una sola di queste proprietà del motore, è più efficiente usare comandi separati; ciò richiede infatti meno memoria all'RCX, è più veloce e può anche risultare più elegante. Questi due comandi sono `SetDirection()`, che imposta la direzione (`OUT_FWD`, `OUT_REV` o `OUT_TOGGLE` che inverte la direzione corrente) e `SetOutput()`, che imposta la modalità (`OUT_ON`, `OUT_OFF` or `OUT_FLOAT`). Ecco un semplice programma che fa avanzare, indietreggiare ed avanzare ancora il robot.

```
task main()
{
  SetPower(OUT_A+OUT_C,7);
  SetDirection(OUT_A+OUT_C,OUT_FWD);
  SetOutput(OUT_A+OUT_C,OUT_ON);
  Wait(200);
  SetDirection(OUT_A+OUT_C,OUT_REV);
  Wait(200);
  SetDirection(OUT_A+OUT_C,OUT_TOGGLE);
  Wait(200);
  SetOutput(OUT_A+OUT_C,OUT_FLOAT);
}
```

Nota che, all'inizio di ogni programma, tutti i motori sono impostati in avanti e con una potenza di 7. Quindi, nell'esempio sopra, le prime due istruzioni non sono necessarie.

Ci sono ancora degli altri comandi, che rappresentano scorciatoie per i comandi presentati qui sopra. Eccone una lista completa:

`On('motori')`	Accende i motori
`Off('motori')`	Spegne i motori
`Float('motori')`	Spegne i motori più elegantemente
`Fwd('motori')`	Imposta la direzione dei motori in avanti
`Rev('motori')`	Imposta la direzione dei motori indietro
`Toggle('motori')`	Cambia la direzione dei motori
`OnFwd('motori')`	Imposta la direzione in avanti e accende i motori
`OnRev('motori')`	Imposta la direzione indietro e accende i motori

`OnFor('motori','battiti')`	Accende i motori per una durata in battiti
`SetOutput('motori','modo')`	Imposta la modalità (`OUT_ON`, `OUT_OFF` or `OUT_FLOAT`)
`SetDirection('motori','dir')`	Imposta la direzione (`OUT_FWD`, `OUT_REV` or `OUT_TOGGLE`)
`SetPower('motori','potenza')`	Imposta la potenza (0-9)

Cambiare la velocità dei motori

Come avrai probabilmente notato, cambiare la velocità dei motori non ha grandi effetti. La ragione è che stai modificando la potenza, non la velocità. Potrai notare gli effetti solo con pesanti carichi. E anche allora, la differenza tra 2 e 7 sarà molto piccola. Se vuoi ottenere migliori risultati, il trucco è accendere e spegnere i motori in rapida successione. Ecco un semplice programma che lo dimostra. Ha un task, chiamato run_motor, che guida i motori. Controlla costantemente la variabile speed per conoscere la velocità corrente. Un valore positivo significa avanti, negativo indietro. Imposta i motori nella giusta direzione e quindi attende per un po', a seconda della velocità, prima di spegnere nuovamente i motori. Il task principale decide semplicemente la velocità e le attese.

```
int speed, __speed;

task run_motor()
{
  while (true)
  {
    __speed = speed;
    if (__speed > 0) {OnFwd(OUT_A+OUT_C);}
    if (__speed < 0) {OnRev(OUT_A+OUT_C); __speed =
-__speed;}
    Wait(__speed);
    Off(OUT_A+OUT_C);
  }
}

task main()
{
  speed = 0;
  start run_motor;
  speed = 1;   Wait(200);
  speed = -10; Wait(200);
  speed = 5;   Wait(200);
  speed = -2;  Wait(200);
  stop run_motor;
  Off(OUT_A+OUT_C);
}
```

Questo programma può essere notevolmente migliorato, includendo rotazioni e possibilmente un'attesa prima dell'istruzione `Off()`. Provare per credere.

Ricapitolazione

In questo capitolo hai imparato ad usare le istruzioni supplementari per il controllo dei motori: `Float()`, che spegne i motori elegantemente, `SetDirection()`, che imposta la direzione (`OUT_FWD`, `OUT_REV` o `OUT_TOGGLE` che inverte la direzione corrente) , e `SetOutput()`, che imposta la modalità (`OUT_ON`, `OUT_OFF` or `OUT_FLOAT`). Hai visto l'intera lista dei comandi disponibili. Hai anche imparato un trucco per meglio controllare la velocità dei motori.

IX. Ancora sui sensori

Nel capitolo V ti ho mostrato le basi dell'utilizzo dei sensori. Ma puoi fare molte altre cose con essi. In questo capitolo vedremo la differenza tra modalità e tipo del sensore, vedremo come usare il sensore di rotazione (un tipo di sensore non compreso nel RIS, ma che può essere acquistato separatamente e si rivela molto utile), ed infine vedremo come usare più di tre sensori e costruire un sensore di prossimità.

Modalità e tipo del sensore

L'istruzione `SetSensor()` che abbiamo precedentemente visto esegue due operazioni: imposta il tipo del sensore, e la modalità in cui opera. Impostando queste proprietà separatamente, potrai controllare il comportamento del sensore più precisamente.

Il tipo del sensore viene impostato con l'istruzione `SetSensorType()`. Esistono quattro tipi differenti: `SENSOR_TYPE_TOUCH`, per il sensore al tatto, `SENSOR_TYPE_LIGHT`, per il sensore alla luce, `SENSOR_TYPE_TEMPERATURE`, per il sensore termico (non fa parte del RIS ma si può comprare separatamente), e `SENSOR_TYPE_ROTATION`, per il sensore di rotazione (anche questo non parte del RIS ma disponibile separatamente). Impostare il tipo di sensore è particolarmente importante per indicare se questo abbisogni di energia (come per esempio il sensore alla luce).

La modalità del sensore viene impostata con l'istruzione `SetSensorMode()`. Esistono 8 modalità diverse. La più importante è sicuramente `SENSOR_MODE_RAW`. In questa modalità, il valore che ottieni controllando il sensore è un numero compreso tra 0 e 1023. Esso è il valore grezzo prodotto dal sensore. Il significato di questo valore dipende dal tipo di sensore utilizzato. Per esempio, un sensore al tatto, quando è rilasciato restituisce un valore che si avvicina a 1023. Quando è completamente premuto, è vicino a 50. Premendolo parzialmente, il valore oscillerà tra 50 e 1000. Quindi, se imposti un sensore al tatto nella modalità grezza, puoi anche sapere se è stato premuto solo in parte. Quando il sensore è alla luce, il valore si trova generalmente tra 300 (molto chiaro) e 800 (molto scuro). Ciò fornisce un'indicazione molto più precisa rispetto a quella che si ottiene usando l'istruzione `SetSensor()`.

La seconda modalità è `SENSOR_MODE_BOOL`. In questa modalità, i valori che ottieni sono sempre o 0 o 1. Quando il valore grezzo supera 550 il sensore restituisce 0, altrimenti 1. `SENSOR_MODE_BOOL` è la modalità predefinita per un sensore al tatto. Le modalità `SENSOR_MODE_CELSIUS` e `SENSOR_MODE_FAHRENHEIT` sono usati solo con i sensori termici e forniscono una temperatura usando la gradazione indicata del nome. `SENSOR_MODE_PERCENT` converte il valore grezzo in un valore tra 0 e 100. Una valore grezzo di 400 o inferiore restituisce 100 percento. Man mano che il valore sale, la percentuale si avvicina a 0. `SENSOR_MODE_PERCENT` è la modalità predefinita per i sensori alla luce. `SENSOR_MODE_ROTATION` sembra essere utilizzabile solo con il sensore di rotazione (vedi sotto).

Esistono inoltre altre due interessanti modalità: `SENSOR_MODE_EDGE` e `SENSOR_MODE_PULSE`. Essi contano le transizioni, ovvero i passaggi da un valore alto ad uno basso e viceversa. Per esempio, la pressione di un sensore al tatto provoca un passaggio da un valore alto ad uno basso. Quando lo rilasci, ottieni poi un'altra transizione. Quando imposti la modalità a `SENSOR_MODE_PULSE`, vengono contati solo i cambiamenti da un valore basso ad uno alto. Quindi ogni pressione e rilascio valgono per uno. Quando invece la imposti a `SENSOR_MODE_EDGE`, vengono contate entrambe le transizioni. Quindi pressione e rilascio valgono per due. Detto ciò, puoi usarlo per vedere quante volte un sensore viene premuto. Oppure, con un sensore alla luce, puoi contare quante volte una (forte) lampadina viene accesa e spenta. Ovviamente, quando si conta, bisognerebbe anche essere in grado di riazzerare il conteggio. Per fare ciò si usa l'istruzione `ClearSensor()`.

Osserviamo ora un esempio. Il seguente programma usa un sensore al tatto per far girare il robot. Connetti il sensore con un lungo cavo alla porta di input numero uno. Premendo velocemente due volte il sensore il robot avanzerà. Premendolo una sola volta invece, si arresterà.

```
task main()
{
  SetSensorType(SENSOR_1,SENSOR_TYPE_TOUCH);
  SetSensorMode(SENSOR_1,SENSOR_MODE_PULSE);
  while(true)
  {
    ClearSensor(SENSOR_1);
    until (SENSOR_1 >0);
    Wait(100);
    if (SENSOR_1 == 1) {Off(OUT_A+OUT_C);}
    if (SENSOR_1 == 2) {OnFwd(OUT_A+OUT_C);}
  }
}
```

Nota che abbiamo prima impostato il tipo del sensore e solo dopo la modalità. Sembra che ciò sia essenziale, poiché il tipo del sensore influenza anche la modalità.

Il sensore di rotazione

Il sensore di rotazione è un tipo molto utile di sensore che non viene sfortunatamente fornito con il RIS. Può però essere acquistato separatamene. Esso possiede una rientranza nella quale si può inserire un'asse rotante. Il sensore misura quanto suddetta asse viene ruotata. Una rotazione completa corrisponde ad un valore di 16 (o di -16 ruotandola nell'altra direzione). I sensori di rotazione risultano utili per controllare i movimenti del robot. Puoi controllare i movimenti con una precisione fenomenale. Se vuoi un controllo più preciso che non usando 16 gradi d'inclinazione, puoi sempre usare delle ruote per creare un sistema di riduzione, e quindi usarlo per le misurazioni.

Un'applicazione diffusa è connettere due sensori alle ruote del robot, da controllare poi con i motori. Per procedere in linea retta, vorrai che i motori girino allo stesso ritmo. Sfortunatamente, i motori non girano solitamente alla stessa velocità. Usando i sensori di rotazione puoi notare se una ruota sia più veloce dell'altra. Puoi fermare momentaneamente il motore in questione (meglio usando `Float()`) finché entrambi i sensori non diano gli stessi valori. Il programma che segue mostra come fare ciò. Fa semplicemente procedere il robot in linea retta. Per poterlo usare, attacca i sensori sopra i motori, e connettili alle porte di input 1 e 3.

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_ROTATION); ClearSensor(SENSOR_1);
  SetSensor(SENSOR_3,SENSOR_ROTATION); ClearSensor(SENSOR_3);
  while (true)
  {
    if (SENSOR_1 < SENSOR_3)
      {OnFwd(OUT_A); Float(OUT_C);}
    else if (SENSOR_1 > SENSOR_3)
      {OnFwd(OUT_C); Float(OUT_A);}
    else
      {OnFwd(OUT_A+OUT_C);}
  }
}
```

Il programma inizia con l'indicare che entrambi i sensori sono di rotazione, quindi azzera i rispettivi valori. Fatto ciò, fa partire un ciclo infinito. Ogni volta, controlliamo i sensori per vedere se indicano valori uguali. In tal caso, il robot avanza semplicemente. Se invece uno è maggiore, un motore viene fermato finché non si ripristinano le condizioni ideali.

Naturalmente, questo è un programma di semplicità elementare. Puoi ampliarlo per far percorrere al tuo robot distanze esatte, o curve molto precise.

Più sensori con una sola porta

L'RCX ha solo tre porte di input, quindi puoi connettervi solo tre sensori. Quando vorrai realizzare robot più complicati (e sarai magari in possesso di qualche sensore extra) quindi, sarai notevolmente limitato. Fortunatamente però, con qualche trucco, puoi connettere più sensori ad una sola porta.

La soluzione più comune è connettere assieme due sensori. Se uno di essi (o entrambi) viene premuto, il valore restituito è 1, altrimenti è 0. In questa maniera non potrai distinguerli, ma a volte ciò non è importante. Per esempio, quando installi un sensore nella parte anteriore ed uno in quella posteriore, puoi saper quale è stato premuto a seconda della direzione nella quale il robot sta procedendo. Puoi anche impostare la modalità del sensore a raw, ed ottenere molte più informazioni. Se sei fortunato, il valore della pressione non diviene lo stesso per entrambi i sensori. In tal caso puoi anche distinguere direttamente tra i due sensori. E quando entrambi sono premuti, ottieni un valore più basso del normale, attorno a 30.

Puoi anche connettere insieme un sensore al tatto ed uno alla luce. Imposta il tipo a sensore alla luce (altrimenti questo non può funzionare), e la modalità a raw. Ora, quando il sensore al tatto sarà premuto, otterrai un valore inferiore a 100. Altrimenti il valore non scenderà mai sotto questa soglia. Il seguente programma mette in pratica questa idea. Il robot deve essere equipaggiato con un sensore alla luce che punti in basso, ed un sensore al tatto nella parte anteriore. Connettili entrambi alla porta 1. Il robot andrà a spasso in maniera casuale cercando un'aerea luminosa. Quando il sensore vede una linea nera (valore grezzo > 750), esso indietreggia un poco. Lo stresso avviene se il sensore al tatto viene premuto (valore grezzo < 100). Ecco il codice:

```
int ttt,tt2;

task moverandom()
{
  while (true)
  {
    ttt = Random(50) + 40;
    tt2 = Random(1);
    if (tt2 > 0)
      { OnRev(OUT_A); OnFwd(OUT_C); Wait(ttt); }
    else
      { OnRev(OUT_C); OnFwd(OUT_A);Wait(ttt); }
    ttt = Random(150) + 50;
    OnFwd(OUT_A+OUT_C);Wait(ttt);
  }
}

task main()
{
  start moverandom;
  SetSensorType(SENSOR_1,SENSOR_TYPE_LIGHT);
  SetSensorMode(SENSOR_1,SENSOR_MODE_RAW);
  while (true)
  {
    if ((SENSOR_1 < 100) || (SENSOR_1 > 750))
    {
      stop moverandom;
      OnRev(OUT_A+OUT_C);Wait(30);
      start moverandom;
    }
  }
}
```

Spero che il programma ti sia chiaro. Esso è composto da due task. Il compito di `moverandom` è far muovere il robot in maniera casuale. Il task principale esegue `moverandom`, imposta il sensore e aspetta che qualcosa accada. Se il valore diventa troppo basso (pressione) o troppo alto (area oscura) termina i movimenti casuali, indietreggia, e quindi riprende a muoversi.

È anche possibile connettere assieme due sensori alla luce. Il valore grezzo sarà in qualche modo collegato con l'ammontare di luce totale ricevuta dai sensori. Ciò diviene però poco chiaro e difficile da usare. Altri abbinamenti dei sensori poi, non sembrano di particolare utilità.

Costruire un sensore di prossimità

Usando un sensore al tatto, il tuo robot può reagire quando urta qualcosa. Effettivamente però, sarebbe molto più elegante farlo reagire prima dell'impatto. Dovrebbe sapere di essere vicino all'ostacolo. Sfortunatamente non esistono sensori per svolgere questa mansione. Dovremo quindi usare un piccolo trucco. L'RCX è dotato di una porta ad infrarossi per poter comunicare con il computer o con altri robot. (Approfondiremo questo argomento

nel capitolo Errore: sorgente del riferimento non trovata.) Ora, guarda caso, salta fuori che il sensore alla luce è molto sensibile alla luce infrarossa. Possiamo costruire un sensore di prossimità che si basa su ciò. L'idea è che un task continua ad inviare messaggi con la porta ad infrarossi; Un altro task intanto misura i cambiamenti dell'intensità della luce riflessa. Più alta l'intensità, più vicini saremo all'oggetto.

Per mettere in pratica quest'idea, attacca il sensore alla luce sopra la porta ad infrarossi del robot, puntandolo in avanti. In questa maniera, può misurare solo la luce riflessa. Connettilo alla porta numero 2. Per il sensore alla luce adoperiamo la modalità grezza, per poter usufruire della massima precisione. Ecco un semplice programma che fa avanzare il robot finché questo non sia vicino ad un oggetto, per farlo quindi svoltare di 90 gradi.

```
int lastlevel;            // Memorizza il livello precedente

task send_signal()
{
  while(true)
    {SendMessage(0); Wait(10);}
}

task check_signal()
{
  while(true)
  {
    lastlevel = SENSOR_2;
    if(SENSOR_2 > lastlevel + 200)
      {OnRev(OUT_C); Wait(85); OnFwd(OUT_A+OUT_C);}
  }
}

task main()
{
  SetSensorType(SENSOR_2, SENSOR_TYPE_LIGHT);
  SetSensorMode(SENSOR_2, SENSOR_MODE_RAW);
  OnFwd(OUT_A+OUT_C);
  start send_signal;
  start check_signal;
}
```

Il task `send_signal` spedisce 10 segnali con gli infrarossi ogni secondo, usando l'istruzione `SendMessage(0)`. `check_signal` salva il valore del sensore alla luce in continuazione. Quindi controlla se sia aumentato almeno di 200, indicando un ampio cambiamento. Se le cose stanno così, fa svoltare il robot a destra. Il valore 200 è piuttosto arbitrario. Se lo diminuisci, il robot gira ben lontano dagli ostacoli. Se lo aumenti si avvicina maggiormente ad essi. Ma dipende anche dal tipo di materiale e dal grado di luminosità della stanza. Potresti escogitare qualche sistema per calcolare automaticamente il valore più consono ad ogni situazione.

Uno svantaggio di questa tecnica è che può funzionare in una sola direzione. Avrai comunque bisogno di sensori al tatto ai lati del robot per evitare collisioni. È comunque un'ottima tecnica per i robot che devono orientarsi all'interno di labirinti. Un altro svantaggio è che non puoi comunicare al robot col computer, poiché interferirebbe con i messaggi inviati dal robot. (Anche il telecomando del tuo televisore potrebbe non funzionare.)

Ricapitolazione

In questo capitolo abbiamo visto diverse questioni riguardanti i sensori. Abbiamo visto come impostare il tipo e la modalità dei sensori e come poter ottenere misurazioni più precise. Abbiamo imparato ad usare il sensore di rotazione, e abbiamo anche visto come sia possibile connettere più sensori ad una sola porta di input. Infine, abbiamo imparato un trucco per poter usufruire di un comodo sensore di prossimità, usando la porta ad infrarossi dell'RCX ed un sensore alla luce. Tutte queste tecniche si rivelano utili durante la realizzazione dei robot più complessi. Infatti in questi casi i sensori rappresentano sempre punti cruciali.

X. Task paralleli

Come detto in precedenza, i task in NQC sono eseguiti simultaneamente, o parallelamente come in genere si dice. Ciò è estremamente utile; ti permette infatti di controllare i sensori mentre muovi il robot e suoni della buona musica. Ma task paralleli possono anche causare problemi. Perché un task può anche interferire con gli altri.

Un programma sbagliato

Considera il seguente programma. Un task fa disegnare al robot quadrati (come abbiamo spesso fatto in precedenza) mentre il secondo controlla i sensori. Quando uno di questi viene premuto, il robot indietreggia un poco, ed esegue una curva ad angolo retto.

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  start check_sensors;
  while (true)
  {
    OnFwd(OUT_A+OUT_C); Wait(100);
    OnRev(OUT_C); Wait(85);
  }
}

task check_sensors()
{
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      OnRev(OUT_A+OUT_C);
      Wait(50);
      OnFwd(OUT_A);
      Wait(85);
      OnFwd(OUT_C);
    }
  }
}
```

Questo codice potrebbe sembrare valido. Se però provi ad eseguirlo, noterai dei comportamenti inaspettati. Prova ad esempio a far toccare al robot qualcosa mentre sta girando. Inizierà ad indietreggiare, ma dopo poco avanzerà nuovamente, andando ad urtare l'ostacolo. La ragione è che il task potrebbe interferire. Accade quanto segue. Il robot sta svoltando a destra, ovvero, il primo task si trova nel secondo comando d'attesa. Ora il sensore viene premuto. Inizia ad indietreggiare, ma in quel preciso istante, il task principale riprende a far avanzare il robot; così, va a colpire l'ostacolo. Il secondo task in questo momento sta attendendo, quindi non può rendersi conto della pressione del sensore. Questo comportamento non è certo quello che volevamo. Il problema è che, mentre il secondo task attende, il primo è ancora in esecuzione, ed interferisce quindi con le operazioni del secondo.

Fermare e riavviare i task

Una maniera per risolvere questo problema è accertarsi che, in ogni momento, un solo task stia guidando il robot. Questo è l'approccio che abbiamo adottato nel capitolo VI. Lasciatemi qui riproporre il programma.

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  start check_sensors;
  start move_square;
}

task move_square()
{
  while (true)
  {
    OnFwd(OUT_A+OUT_C); Wait(100);
    OnRev(OUT_C); Wait(85);
  }
}

task check_sensors()
{
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      stop move_square;
      OnRev(OUT_A+OUT_C); Wait(50);
      OnFwd(OUT_A); Wait(85);
      start move_square;
    }
  }
}
```

il punto cruciale sta nel fatto che `check_sensors` muove il robot solo dopo aver fermato `move_square`. Questo task non può quindi più interferire durante l'allontanamento dall'ostacolo. Dopo aver indietreggiato, fa ripartire nuovamente `move_square`.

Nonostante questa rappresenti una buona soluzione al problema, si presenta un'altra questione. Quando riavviamo `move_square`, questa riparte dall'inizio. Ciò è adatto a questo piccolo task, ma spesso non rappresenta il comportamento più adeguato. Potremmo preferire fermare un task e quindi riprendere da dove avevamo smesso. Sfortunatamente, ciò non è molto facile da mettere in pratica.

Usare i semafori

Una tecnica standard per risolvere il problema è l'uso di una variabile che indichi quale task abbia il controllo dei motori. Agli altri task non è quindi permesso accedere ai motori finché il primo, usando la variabile, non dica di essere pronto. Una variabile di questo tipo viene spesso chiamata semaforo. Mettiamo che `sem` sia il nostro semaforo. Poniamo che il valore 0 indichi che nessun task sta correntemente facendo girare i motori. Ora, quando un task vuole usare i motori, utilizzerà questo forma:

```
until (sem == 0);
sem = 1;
// Fai qualcosa con i motori
sem = 0;
```

Aspettiamo quindi che nessuno stia usando i motori. Dichiariamo quindi l'utilizzo impostando `sem` a 1. Ora possiamo controllare i motori. Quando abbiamo finito, reimpostiamo `sem` a 0. Di seguito è riportata una versione del programma precedente, con l'utilizzo di un semaforo. Quando il sensore urta qualcosa, il semaforo viene attivato e parte la procedura di allontanamento. Durante questa procedura, il task `move_square` è obbligato ad aspettare. Quando infine abbiamo terminato, il semaforo viene nuovamente posto a 0 cosicché `move_square` possa continuare.

```
int sem;

task main()
{
  sem = 0;
  start move_square;
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      until (sem == 0); sem = 1;
      OnRev(OUT_A+OUT_C); Wait(50);
      OnFwd(OUT_A); Wait(85);
      sem = 0;
    }
  }
}

task move_square()
{
  while (true)
  {
    until (sem == 0); sem = 1;
    OnFwd(OUT_A+OUT_C);
    sem = 0;
    Wait(100);
    until (sem == 0); sem = 1;
    OnRev(OUT_C);
    sem = 0;
    Wait(85);
  }
}
```

Potresti obiettare che in `move_square` non sia necessario impostare il semaforo a 1 e di nuovo a 0. Invece, ciò si rivela necessario. La ragione è che l'istruzione `OnFwd()` è praticamente costituita da due operazioni (vedi capitolo VIII). Ovviamente, non vogliamo che questa sequenza venga interrotta da altri task.

I semafori sono spesso utili, soprattutto per scrivere programmi complicati con task che operano in parallelo. (C'è comunque sempre una possibilità che essi possano fallire. Prova ad immaginare perché.)

Ricapitolazione

In questo capitolo abbiamo studiato alcuni problemi che possono capitare quando si usano diversi task. Devi sempre stare attento ai possibili effetti collaterali. Molti comportamenti inaspettati sono infatti spesso dovuti a questo. Abbiamo visto due modi per risolvere il problema. La prima soluzione ferma e riavvia i task per esser sicuri che uno solo stia correntemente avendo il controllo. Il secondo approccio fa invece uso di semafori per controllare i task. Questo garantisce che in ogni momento sia eseguita solo la parte critica di ogni task.

XI. Comunicazione tra i robot

Se possiede più di un RCX, questo capitolo fa per te. I robot possono comunicare tra di loro usando la porta ad infrarossi. Con questo sistema puoi far collaborare diversi robot (o farli combattere insieme). Puoi anche costruire un robot gigante usando due RCX, per poter così usufruire di sei motori ed altrettanti sensori (o magari anche di più usando il trucco descritto nel capitolo IX).

La comunicazione tra i robot avviene, in generale, come segue. Un robot può usare l'istruzione `SendMessage()` per inviare un valore (0-255) con gli infrarossi. Tutti gli altri robot ricevono il messaggio e lo memorizzano. Un programma di un robot può ottenere il valore dell'ultimo messaggio ricevuto usando `Message()`. Usando questo valore il programma può reagire di conseguenza.

Dare ordini

Spesso, quando usi due o più robot, uno è il leader. Questo prende il nome di *master*. Gli altri robot sono invece *slave*. Il robot master invia comandi agli slave e questi li eseguono. A volte gli slave possono rispondere al master, per indicare ad esempio il valore di un sensore. Dovrai quindi scrivere due programmi, uno per il master e uno per gli slave. Da ora in poi assumo che tu utilizzi un solo slave. Partiamo con un esempio semplicissimo. Qui lo slave può obbedire a tre diversi ordini: avanzare, indietreggiare e fermarsi. Il suo programma consiste di un semplice ciclo iterativo. In questo ciclo, imposta il valore del messaggio corrente a 0 usando l'istruzione `ClearMessage()`. Quindi aspetta finché questo non sia diverso da 0. A seconda del valore del messaggio, esegue uno dei tre comandi. Ecco il programma.

```
task main()           // SLAVE
{
  while (true)
  {
    ClearMessage();
    until (Message() != 0);
    if (Message() == 1) {OnFwd(OUT_A+OUT_C);}
    if (Message() == 2) {OnRev(OUT_A+OUT_C);}
    if (Message() == 3) {Off(OUT_A+OUT_C);}
  }
}
```

Il master è dotato di un programma ancora più semplice. Si limita infatti a spedire dei messaggi per poi attendere un poco. Nel programma qui sotto, ordina agli slave di avanzare, e poi, dopo due secondi, di tornare indietro, e poi, dopo altri due secondi, di fermarsi.

```
task main()           // MASTER
{
  SendMessage(1); Wait(200);
  SendMessage(2); Wait(200);
  SendMessage(3);
}
```

dopo aver scritto questi due programmi, devi scaricarli sui robot. Ogni programma deve andare ad un robot. Assicurati di tenerne acceso uno soltanto durante l'operazione (vedi gli avvertimenti riportati sotto). Accendi quindi entrambi i robot ed esegui i programmi: prima quello dello slave e poi quello del master.

Se stai usando più slave, dovrai scaricare il programma ad uno per uno (e non insieme, vedi sotto). Ora tutti gli slave eseguiranno perfettamente le stesse azioni.

Affinché i robot potessero comunicare tra loro abbiamo definito quel che viene chiamato un protocollo: abbiamo deciso che 1 significa muoversi avanti, 2 indietro, e 3 fermarsi. È molto importante definire il protocollo di comunicazione, in particolare quando si ha a che fare con un gran numero di messaggi. Per esempio, quando ci sono più slave, puoi definire un protocollo in cui vengono inviati due numeri (con una breve attesa frapposta): il primo indica il numero di uno slave, ed il secondo il comando vero e proprio. Lo slave quindi eseguirà l'operazione solo se questa è per lui. (ciò richiede che ad ogni slave corrisponda un numero, operazione resa possibile dall'uso, ad esempio, di una costante da modificare per ogni robot.)

Eleggiamo un leader

Come abbiamo visto in precedenza, avendo a che fare con più robot, ognuno deve avere il proprio programma. Sarebbe molto più facile avere un solo programma da scaricare su tutti i robot. Si presenterebbe però un problema: chi sarà il master? La risposta è semplice: fallo decidere ai robot. Fai eleggere loro un leader a cui gli altri obbediranno. Come possiamo fare ciò? L'idea è piuttosto semplice. Facciamo sì che i robot attendano per un po' di tempo e quindi inviino un messaggio. Quello che invierà il messaggio per primo sarà il leader. Questo ragionamento potrebbe però fallire nel caso in cui due robot attendano lo stesso numero di battiti (puoi progettare sistemi più complicati che rilevino ciò ed in tal caso ripetano le elezioni). Ecco il programma:

```
task main()
{
  ClearMessage();
  Wait(200);               // Aspetta che tutti siano accesi
  Wait(Random(400));       // Attendi da 0 a 4 secondi
  if (Message() > 0)       // E' arrivato prima qualcun altro
  {
    start slave;
  }
  else
  {
    SendMessage(1);        // Sono io il master ora
    Wait(400);             // Aspettiamo che tutti lo sappiano
    start master;
  }
}

task master()
{
  SendMessage(1); Wait(200);
  SendMessage(2); Wait(200);
  SendMessage(3);
}

task slave()
{
  while (true)
  {
    ClearMessage();
    until (Message() != 0);
    if (Message() == 1) {OnFwd(OUT_A+OUT_C);}
    if (Message() == 2) {OnRev(OUT_A+OUT_C);}
    if (Message() == 3) {Off(OUT_A+OUT_C);}
  }
}
```

scarica questo programma su tutti i robot (uno per uno, non assieme; vedi sotto). Accendi i robot nello stesso momento e guarda che cosa succede. Uno di questi dovrebbe comandare e gli altri obbedire. In alcune occasioni potrebbe accadere che nessuno di essi diventi il master. Come indicato sopra, per risolvere il problema vi è bisogno di protocolli più complicati.

Avvertimenti

Devi stare un po' attento quando gestisci diversi robot. Possono infatti presentarsi due problemi: se due robot (o un robot ed il computer) inviano delle informazioni contemporaneamente, queste potrebbero andare perse. Il secondo problema è che inviare lo stesso programma dal computer a più robot causa problemi.

Iniziamo col discutere il secondo problema. Quando scarichi un programma su di un robot, questo risponde se ha o meno ricevuto correttamente le diverse parti del programma. Il computer reagisce quindi continuando oppure ripetendo ciò che non era stato compreso. Se due robot sono accesi, entrambi risponderanno al computer. Questo non può comprendere la risposta (non sa che ci sono due robot!). in pratica, il programma viene comunicato in maniera errata. I robot non faranno ciò che è giusto. *Accertati sempre che, durante lo scaricamento di un programma, sia acceso un solo robot!*

L'altro problema è che si può inviare un solo messaggio nello stesso istante. Se due messaggi vengono inviati nello stesso momento, possono andare persi. Inoltre, un robot non può inviare e ricevere messaggi allo stesso tempo. Questo non rappresenta un problema quando è uno solo il robot che da ordini (il master), ma in altri casi la situazione potrebbe diventare molto complessa. Per esempio, puoi immaginare un programma in cui uno slave invia un messaggio quando urta qualcosa, cosicché il master possa prendere provvedimenti. Ma se il master deve inviare un messaggio nello stesso momento, questo sarà perso. Per porre un rimedio, è importante definire un protocollo di comunicazione, in maniera che, se una comunicazione fallisce, viene corretta. Per esempio, quando il master invia un ordine, dovrebbe ricevere una risposta dallo slave. Se non riceve una risposta entro un tempo ragionevole, ripete l'invio. Il risultato sarebbe un frammento di codice simile a questo:

```
do
{
  SendMessage(1);
  ClearMessage();
  Wait(10);
}
while (Message() != 255);
```

Qui 255 viene usato come segnale di comprensione.

A volte, utilizzando più robot, potresti volere che uno solo di questi, il più vicino, riceva il segnale. Ciò può essere ottenuto tramite l'istruzione `SetTxPower(TX_POWER_LO)` nel programma del master. Così facendo, il segnale IR viene inviato molto debolmente e solo un robot vicino al master può sentirlo. Ciò è particolarmente utile per costruire un robot più grande con due RCX. Usa invece `SetTxPower(TX_POWER_HI)` per impostare nuovamente un lungo raggio di trasmissione.

Ricapitolazione

In questo capitolo abbiamo studiato alcuni aspetti fondamentali della comunicazione tra i robot. Esistono comandi per inviare, controllare ed azzerare i messaggi. Abbiamo visto l'importanza di definire un protocollo per la comunicazione. Questi giocano un ruolo cruciale in ogni forma di comunicazione tra computer. Abbiamo anche visto alcune restrizioni che potrebbero rendere difficile la definizione di un buon protocollo.

XII. Altri comandi

In NQC esistono diverse altre istruzioni. In questo capitolo vedremo tre categorie: l'uso dei timer, il controllo del display, e l'uso del datalog dell'RCX.

I timer

L'RCX possiede quattro timer. Questi timer producono battiti da 1/10 di secondo, e sono numerati da 0 a 3. Puoi azzerare un timer con l'istruzione `ClearTimer()` ed ottenere il valore corrente con `Timer()`. Ecco un esempio di uso dei timer. Il seguente programma fa procedere il robot casualmente per 20 secondi.

```
task main()
{
  ClearTimer(0);
  do
  {
    OnFwd(OUT_A+OUT_C);
    Wait(Random(100));
    OnRev(OUT_C);
    Wait(Random(100));
  }
  while (Timer(0)<200);
  Off(OUT_A+OUT_C);
}
```

potresti voler confrontare questo programma con quello riportato nel capitolo IV, che esegue esattamente le stesse operazioni. Quello con i timer è però nettamente più semplice.

I timer risultano molto utili al posto delle istruzioni `Wait()`. Puoi attendere per un tot di tempo azzerando un timer e quindi aspettando che raggiunga un determinato valore. Ma nel frattempo puoi anche reagire ad altri eventi (provocati ad esempio dai sensori). Il seguente programma dimostra come fare ciò. Fa guidare il robot finché non siano passati 10 secondi, oppure il sensore non tocchi qualcosa.

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  ClearTimer(3);
  OnFwd(OUT_A+OUT_C);
  until ((SENSOR_1 == 1) || (Timer(3) >100));
  Off(OUT_A+OUT_C);
}
```

non dimenticare che i timer lavorano con battiti da 1/10 di secondo, mentre per esempio il comando wait usa battiti da 1/100.

Il display

È possibile controllare il display dell'RCX in due maniere diverse. Prima di tutto, puoi indicare cosa vuoi che sia visualizzato: l'orologio di sistema, uno dei sensori o uno dei motori. Ciò equivale ad usare il bottone nero view sull'RCX. Per impostare il tipo del display, usa l'istruzione `SelectDisplay()`. Il seguente programma mostra tutte e sette le possibilità, una dopo l'altra.

```
task main()
{
  SelectDisplay(DISPLAY_SENSOR_1); Wait(100);  // Input 1
  SelectDisplay(DISPLAY_SENSOR_2); Wait(100);  // Input 2
  SelectDisplay(DISPLAY_SENSOR_3); Wait(100);  // Input 3
  SelectDisplay(DISPLAY_OUT_A);    Wait(100);  // Output A
  SelectDisplay(DISPLAY_OUT_B);    Wait(100);  // Output B
  SelectDisplay(DISPLAY_OUT_C);    Wait(100);  // Output C
  SelectDisplay(DISPLAY_WATCH);    Wait(100);  // Orologio
}
```

Nota che non puoi, ad esempio, utilizzare `SelectDisplay(SENSOR_1)`.

Il secondo modo per controllare il display è tramite l'orologio di sistema. Puoi usarlo per mostrare, ad esempio, informazioni per la diagnostica. Per fare ciò, usa l'istruzione `SetWatch()`. Ecco un breve programma che fa ciò:

```
task main()
{
  SetWatch(1,1); Wait(100);
  SetWatch(2,4); Wait(100);
  SetWatch(3,9); Wait(100);
  SetWatch(4,16); Wait(100);
  SetWatch(5,25); Wait(100);
}
```

Gli argomenti a `SetWatch()` devono essere costanti.

Datalogging

L'RCX può immagazzinare il valore di variabili, sensori e timer in uno spazio di memoria chiamato datalog. I valori nel datalog non possono essere usati dall'RCX, ma possono essere letti dal computer. Ciò si rivela utile per controllare, ad esempio, quello che è successo al tuo robot. RCX Command Center ha una speciale finestra nella quale puoi vedere il contenuto corrente del datalog.

Usare il datalog consiste in tre operazioni: prima di tutto, il programma in NQC deve definire le dimensioni del datalog, usando l'istruzione `CreateDatalog()`. Ciò inoltre elimina i contenuti correnti del datalog. Dopo ciò, i valori possono essere scritti con l'istruzione `AddToDatalog()`. I valori saranno scritti uno dopo l'altro. (se osservi il display del tuo RCX, vedrai che una dopo l'altra appariranno quattro parti di un disco. Quando il disco è completo, il datalog è pieno) raggiunta la fine del datalog, non succede più niente. I valori non vengono più memorizzati. Il terzo passaggio consiste infine nel caricare i dati sul PC. Per fare ciò, seleziona in RCX Command Center la voce **Datalog** dal menu **Tools**. Premi quindi il pulsante **Upload Datalog**, ed appariranno tutti i valori. Puoi semplicemente osservarli oppure salvarli in un file per un qualche scopo. C'è ad esempio chi ha usato questa caratteristica per realizzare uno scanner con l'RCX.

Ecco un semplice programma per un robot con il sensore alla luce. Il robot procede per 10 secondi, e cinque volte al secondo il valore del sensore viene immagazzinato nel datalog.

```
task main()
{
  SetSensor(SENSOR_2,SENSOR_LIGHT);
  OnFwd(OUT_A+OUT_C);
  CreateDatalog(50);
  repeat (50)
  {
    AddToDatalog(SENSOR_2);
    Wait(20);
  }
  Off(OUT_A+OUT_C);
}
```

XIII. Riferimenti al linguaggio NQC

Qui sotto puoi trovare una lista di istruzioni, comandi, costanti, ecc. in NQC. Molti di questi sono già stati trattati nei capitoli precedenti, e sono quindi corredati solo di una breve descrizione.

Comandi

Comando	Descrizione
while (*cond*) *corpo*	esegui il corpo zero o più volte fintanto che la condizione è vera
do *corpo* while (*cond*)	esegui il corpo una o più volte fintanto che la condizione è vera
until (*cond*) corpo	esegui il corpo zero o più volte fintanto che la condizione è falsa
break	esci dal corpo di un while/do/until
continue	salta alla prossima iterazione di un while/do/until
repeat (*espressione*) corpo	ripeti il corpo un determinato numero di volte
if (*cond*) *stmt1* **if** (*cond*) *stmt1* **else** *stmt2*	esegui stmt1 se la condizione è vera. Esegui stmt2 (se presente) se la condizione è falsa.
start nome_*task*	esegui il task specificato
stop nome_*task*	arresta il task specificato
function(*argomenti*)	chiama una funzione con gli argomenti specificati
var = espressione	valuta un'espressione e la memorizza in una variabile
var += espressione	valuta un'espressione e la aggiunge ad una variabile
var -= espressione	valuta un'espressione e la sottrae da una variabile
*var *= espressione*	valuta un'espressione e la moltiplica con una variabile
var /= espressione	valuta un'espressione e ci divide una variabile
var |= espressione	valuta un'espressione ed esegue un OR bit a bit con una variabile
var &= espressione	valuta un'espressione ed esegue un AND bit a bit con una variabile
return	ritorna da una funzione a dove è stata chiamata
espressione	valuta un'espressione

Condizioni

Le condizioni sono utilizzate dalle strutture di controllo per prendere decisioni. Nella maggior parte dei casi la condizione implica un confronto tra espressioni.

Condizione	Significato
true	sempre vero
false	sempre falso
espr1 == espr2	vero se le espressioni sono uguali
espr1 != espr2	vero se le espressioni sono diverse
espr1 < espr2	vero se la prima espressione è minore dell'altra
espr1 <= espr2	vero se la prima espressione è minore o uguale all'altra
espr1 > espr2	vero se la prima espressione è maggiore dell'altra
espr1 >= espr2	vero se la prima espressione è maggiore o uguale all'altra
! condizione	negazione logica di una condizione
cond1 && cond2	AND logico tra due condizioni (vero se e soltanto se entrambe sono vere)
cond1 || cond2	OR logico tra due condizioni (vero se e soltanto se almeno una delle due è vera)

Espressioni

Ci sono diversi valori che possono essere usati tra le espressioni, tra cui costanti, variabili e valori dei sensori. Nota che SENSOR_1, SENSOR_2 e SENSOR_3 sono macro che si espandono rispettivamente a SensorValue(0), SensorValue(1) e SensorValue(2).

Valore	Descrizione
numero	un valore costante (ad esempio "123")
variabile	una variabile con un nome (ad esempio "x")
Timer(*n*)	valore del timer n, dove n è compreso tra 0 e 3
Random(*n*)	un numero casuale compreso tra 0 e n

`SensorValue(n)`	valore corrente dl sensore n, dove n è compreso tra 0 e 2
`Watch()`	valore dell'orologio di sistema
`Message()`	valore dell'ultimo messaggio IR ricevuto

I valori possono essere combinati usando degli operatori. Alcuni operatori possono essere usati solo per valutare espressioni costanti, il che significa che gli operandi devono essere valori costanti. Gli operatori sono qui elencati in ordine di precedenza (dalla più alta alla più bassa).

Operatore	Descrizione	Associazione	Restrizioni	Esempio
`abs()` **`sign()`**	valore assoluto segno	n/a n/a		`abs(x)` `sign(x)`
`++` `--`	incremento decremento	sinistra sinistra	solo variabili solo variabili	`x++ o ++x` `x-- o --x`
`-` `~`	meno unario negazione bit a bit (unario)	destra destra	solo costanti	`-x` `~123`

`*`
`/`
`%`
	moltiplicazione
	divisione
	modulo
		sinistra
		sinistra
		sinistra
			solo costanti	`x * y`
				`x / y`
				`123 % 4`

`+` `-`	addizione sottrazione	sinistra sinistra		`x + y` `x - y`
`<<` `>>`	spostamento a sinistra spostamento a destra	sinistra sinistra	solo costanti solo costanti	`123 << 4` `123 >> 4`
`&`	AND bit a bit	sinistra		`x & y`
`^`	XOR bit a bit	sinistra	solo costanti	`123 ^ 4`
`|`	OR bit a bit	sinistra		`x | y`
`&&`	AND logico	sinistra	solo costanti	`123 && 4`
`||`	OR logico	sinistra	solo costanti	`123 || 4`

Funzioni dell'RCX

La maggior parte delle funzioni richiede come argomenti dei valori costanti. Le eccezioni sono quelle funzioni che richiedono sensori come argomento, e quelle che possono usare qualsiasi espressione. Nel caso dei sensori, l'argomento deve essere un nome di sensore: SENSOR_1, SENSOR_2 o SENSOR_3. In alcuni casi esistono nomi predefiniti (ad esempio SENSOR_TOUCH) per le costanti appropriate.

Funzione	Descrizione	Esempio
`SetSensor(sensore, config)`	configura un sensore	`SetSensor(SENSOR_1, SENSOR_TOUCH)`
`SetSensorMode(sensore, modo)`	imposta la modalità del sensore	`SetSensor(SENSOR_2, SENSOR_MODE_PERCENT)`
`SetSensorType(sensore, tipo)`	imposta il tipo del sensore	`SetSensor(SENSOR_2, SENSOR_TYPE_LIGHT)`
`ClearSensor(sensore)`	azzera un sensore	`ClearSensor(SENSOR_3)`
`On(output)`	accende uno o più output	`On(OUT_A + OUT_B)`
`Off(output)`	spegne uno o più output	`Off(OUT_C)`
`Float(output)`	spegne uno o più output senza usare il freno	`Float(OUT_B)`
`Fwd(output)`	imposta la direzione dell'output in avanti	`Fwd(OUT_A)`
`Rev(output)`	imposta la direzione dell'output indietro	`Rev(OUT_B)`
`Toggle(output)`	inverte la direzione dell'output	`Toggle(OUT_C)`
`OnFwd(output)`	accende ed imposta la direzione in avanti	`OnFwd(OUT_A)`
`OnRev(output)`	accende ed imposta la direzione indietro	`OnRev(OUT_B)`
`OnFor(output, tempo)`	accende per uno specificato numero di	`OnFor(OUT_A, 200)`

	centesimi di secondo. Il tempo può essere un'espressione	
`SetOutput(output, modo)`	imposta la modalità dell'output	`SetOutput(OUT_A, OUT_ON)`
`SetDirection(output, dir)`	imposta la direzione dell'output	`SetDirection(OUT_A, OUT_FWD)`
`SetPower(output, potenza)`	imposta la potenza dell'output (0-7). La potenza può essere un'espressione	`SetPower(OUT_A, 6)`
`Wait(tempo)`	Attende per uno specificato numero di centesimi di secondo. Il tempo può essere un'espressione	`Wait(x)`
`PlaySound(suono)`	esegue il suono specificato (0-5)	`PlaySound(SOUND_CLICK)`
`PlayTone(freq, durata)`	esegue un tono della frequenza indicata per la durata specificata (in centesimi di secondo)	`PlayTone(440, 5)`
`ClearTimer(timer)`	azzera un timer (0-3)	`ClearTimer(0)`
`StopAllTasks()`	arresta tutti i task in esecuzione	`StopAllTasks()`
`SelectDisplay(modo)`	imposta una delle 7 modalità del display: 0: orologio, 1-3: valore dei sensori, 4-6: impostazione degli output. La modalità può essere un'espressione	`SelectDisplay(1)`
`SendMessage(messaggio)`	invia un messaggio IR (1-255). Il messaggio può essere un'espressione	`SendMessage(x)`
`ClearMessage()`	azzera il buffer dei messaggi	`ClearMessage()`
`CreateDatalog(dimensione)`	crea un nuovo datalog della dimensione specificata	`CreateDatalog(100)`
`AddToDatalog(valore)`	aggiunge un valore al datalog. Il valore può essere un'espressione	`AddToDatalog(Timer(0))`
`SetWatch(ore, minuti)`	imposta l'orologio di sistema	`SetWatch(1,30)`
`SetTxPower(hi_lo)`	imposta la potenza della porta ad infrarossi	`SetTxPower(TX_POWER_LO)`

Costanti dell'RCX

Molti valori per le funzioni dell'RCX hanno un nome sotto forma di costante che può rendere il codice più leggibile. Dove possibile, è preferibile usare il nome della costante piuttosto che il valore direttamente.

Configurazione dei sensori per SetSensor()	`SENSOR_TOUCH`, `SENSOR_LIGHT`, `SENSOR_ROTATION`, `SENSOR_CELSIUS`, `SENSOR_FAHRENHEIT`, `SENSOR_PULSE`, `SENSOR_EDGE`
Modalità per SetSensorMode()	`SENSOR_MODE_RAW`, `SENSOR_MODE_BOOL`, `SENSOR_MODE_EDGE`, `SENSOR_MODE_PULSE`, `SENSOR_MODE_PERCENT`, `SENSOR_MODE_CELSIUS`, `SENSOR_MODE_FAHRENHEIT`, `SENSOR_MODE_ROTATION`
Tipi per SetSensorType()	`SENSOR_TYPE_TOUCH`, `SENSOR_TYPE_TEMPERATURE`, `SENSOR_TYPE_LIGHT`, `SENSOR_TYPE_ROTATION`

Output per On(), Off(), ecc.	OUT_A, OUT_B, OUT_C
Modalità per SetOutput()	OUT_ON, OUT_OFF, OUT_FLOAT
Direzioni per SetDirection()	OUT_FWD, OUT_REV, OUT_TOGGLE
Potenza di output per SetPower()	OUT_LOW, OUT_HALF, OUT_FULL
Suoni per PlaySound()	SOUND_CLICK, SOUND_DOUBLE_BEEP, SOUND_DOWN, SOUND_UP, SOUND_LOW_BEEP, SOUND_FAST_UP
Modalità per SelectDisplay()	DISPLAY_WATCH, DISPLAY_SENSOR_1, DISPLAY_SENSOR_2, DISPLAY_SENSOR_3, DISPLAY_OUT_A, DISPLAY_OUT_B, DISPLAY_OUT_C
Potenza per SetTxPower()	TX_POWER_LO, TX_POWER_HI

Parole chiave

Le parole chiave sono quelle parole riservate dal compilatore NQC per il suo linguaggio. È un errore usare una di queste come nome di funzioni, task o variabili. Esistono le seguenti parole chiave: **__sensor, abs, asm, break, const, continue, do, else, false, if, inline, int, repeat, return, sign, start, stop, sub, task, true, void, while.**

XIV. Considerazioni finali

Se sei giunto fin qui attraverso tutto il tutorial puoi ora considerarti un esperto di NQC. Se non l'hai ancora fatto, è ora che tu inizi a fare degli esperimenti. Con la creatività nel design e la programmazione puoi creare robot Lego che facciano cose impensabili.

Questo tutorial non ha coperto tutti gli aspetti di RCX Command Center. Ti raccomando di leggerne la documentazione. Inoltre, NQC è ancora in fase di sviluppo. Le versioni future potrebbero includere funzionalità aggiuntive. Diversi concetti di programmazione non sono stati trattati in questo tutorial. In particolare, non abbiamo considerato i comportamenti dei robot o gli aspetti dell'intelligenza artificiale.

È anche possibile controllare un robot Lego direttamente dal PC. Questo richiede che tu scriva un programma in un linguaggio come Visual Basic, Java o Delphi. È anche possibile far lavorare questi programmi insieme con un programma NQC che viene eseguito dall'RCX. Una combinazione del genere risulta molto efficiente. Se sei interessato a questa modalità di programmazione, è meglio partire scaricandosi la documentazione spirit dal sito web MindStorms.

http://www.legomindstorms.com/

Il web è una fonte perfetta per ottenere informazioni addizionali. Alcuni altri importanti punti di partenza si trovano nella pagina dei collegamenti del mio sito:

http://www.cs.uu.nl/people/markov/lego/

e su LUGNET, il LEGO® Users Group Network (non ufficiale):

http://www.lugnet.com/

Si possono reperire molte informazioni soprattutto nelle newsgroup lugnet.robotics e lugnet.robotics.rcx.nqc su lugnet.com.