Il Linguaggio di Programmazione NXC

Da GGtnWiki.
Vai a: navigazione, ricerca

Indice

[nascondi]

- 1 Autori e licenze di utilizzo
- 2 Il programma Brixcc (ide)
  - 2.1 Il Menu “Tools”
  - **2.2 IL Menu “Compile”**
  - **2.3 La Finestra Principale**
  - **2.4 Sommario**
- **3 Scrittura di un primo programma**
  - **3.1 E se c'è un errore?**
  - **3.2 Sommario**
- **4 Un programma più interessante**
  - **4.1 Fare le curve**
  - **4.2 Ripetizione di comandi**
  - **4.3 Commenti**
  - **4.4 Sommario**
- **5 Uso delle Variabili**
  - **5.1 Movimento a spirale**
  - **5.2 Array di variabili**
  - **5.3 Numeri casuali**
  - **5.4 Sommario**
- **6 Strutture di controllo**
  - **6.1 L'istruzione IF**
  - **6.2 L'istruzione do**
  - **6.3 Sommario**
- **7 Sensori**
  - **7.1 In attesa di un valore dal sensore**
  - **7.2 Utilizzare un sensore di contatto**
  - **7.3 Sensore di luce**
  - **7.4 Sensore di suono**
  - **7.5 Sensore di Ultrasuoni**
  - **7.6 Sommario**
- **8 I task e le subroutine**
  - **8.1 Task**
  - **8.2 Subroutines**
  - **8.3 Definizione di macro.**
  - **8.4 Sommario**
- **9 Fare musica:**
  - **9.1 Riproduzione di file sonori**
  - **9.2 Suonare musica**
  - **9.3 Sommario**
- **10 Gestione avanzata dei motori**
  - **10.1 Arresto non brusco**
  - **10.2 Comandi avanzati**
  - **10.3 PID control**
  - **10.4 Sommario**
- **11 Gestione avanzata dei sensori**
  - **11.1 Sensor Type**
  - **11.2 Sensor Mode**
  - **11.3 Il sensore di rotazione**
  - **11.4 Connettere più sensori ad una porta di input**
  - **11.5 Sommario**
- **12 Task paralleli**
  - **12.1 Un programma sbagliato**
  - **12.2 Regione critica, e variabili mutex**
  - **12.3 Uso dei semafori**
  - **12.4 Sommario**
- **13 Comunicazione tra i robot**
  - **13.1 Messaggi Master-Slave**
  - **13.2 Spedizione con ricevuta di ricezione.**
  - **13.3 Controllo diretto**
  - **13.4 Sommario**
- **14 Ulteriori comandi**
  - **14.1 Timer**
  - **14.2 Display a matrice di pixel**
  - **14.3 File system**
  - **14.4 Sommario**
- **15 Considerazioni finali**

Autori e licenze di utilizzo

Il materiale raccolto in questa sezione è creato da **Cristofori Andrea** presso il **Liceo Galilei di Trento**, e liberamente tratto da **NXC tutorial** di **Daniele Benedettelli** [1] che possiede anche un bel sito dedicato all'NXT: [[2]] L'autore, in accordo con l'estensore originale, sceglie di pubblicare il loro lavoro con licenza Creative Commons "Attribuzione - Non commerciale - Condividi allo stesso modo 2.5.". Ciò significa che ognuno può utilizzare questo materiale, o modificarlo ed

integrarlo a proprio piacimento, purché, a sua volta, consenta il libero utilizzo dell'opera derivata, e che nessuno ne faccia uso commerciale. L'intento è quello di fornire una documentazione in italiano per la programmazione dei robot **Lego NXT**, liberamente fruibile, scritta in maniera collaborativa, in continua crescita ed evoluzione. Scopo di quest'opera, oltre a costituire una documentazione agli studenti che frequentano il corso di robotica presso il Liceo Galilei, è quello di fornire materiale educativo ai docenti che lo vogliano utilizzare per insegnare la programmazione dei robot **Lego NXT.** Chissà che anche lo sviluppo della documentazione da parte dei docenti mediante un approccio collaborativo che stimoli i rapporti interpersonali possa essere espressione dello stesso spirito che anima la **Robocup junior**.

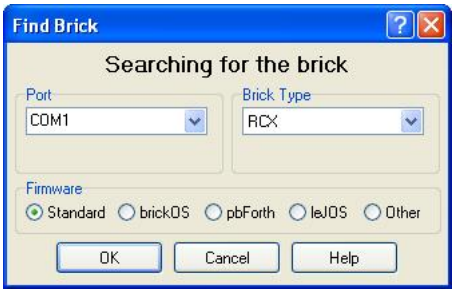
Non solo nell'apprendere la programmazione, ma anche nell'insegnarla, è bello, formativo ed utile condividere la conoscenza.

Dal 2011 collabora attivamente all'aggiornamento del Wiki **Matteo Poletti** che è stato allievo del corso 2010 ed ora tiene il corso di robotica al **Liceo Da Vinci di Trento**.

Il programma Brixcc (ide)

Il software che utilizziamo per la programmazione è **BrixCC** [3]. **BrixCC** permette la scrittura dei programmi, la loro compilazione, la ricerca degli errori, ed il trasferimento del codice al robot. Può gestire programmi scritti in vari linguaggi, ma quello che utilizzeremo in questo tutorial è **NXC**, molto simile al **C** ,quindi un linguaggio di alto livello, molto diffuso ed utilizzato. **BrixCC** è un programma open source, liberamente scaricabile ed utilizzabile, e gira sotto i sistemi operativi proprietari di Microsoft. Benchè la sua utilizzazione sia semplice, una piccola guida può essere utile:

Al lancio del Brixcc il programma verifica se è attivo il collegamento al robot. Si apre una finestrella che chiede quale tipo di interfaccia usare per parlare col robot (**Port**), che tipo di robot si intende programmare (**Brick Type**), ed il tipo di firmware caricato sul robot (**Firmware**). Si scelagno i valori appropriati e si clicchi **OK**



Scelti i valori appropriati nella finestra introduttiva, si apre la finestra principale del programma. Le parti che ci interessano sono:un ampio spazio in cui scrivere il programma, ed i menu a tendina nella parte superiore della finestra.

Se osserviamo i menu a tendina, le funzionalità offerte dal programma sono simili a molti altri già conosciuti. Ad esempio, il menu **File** ci consente di salvare i file prodotti e di ricaricarli da disco al momento opportuno. Niente di nuovo rispetto a tutti gli altri programmi che conosciamo,quindi è inutile soffermarci su queste funzionalità. Le finestre peculiari del programma, cui rivolgere la nostra attenzione, sono essenzialmente due: la finestra **Compile** e la finestra **Tools**

Il Menu “Tools”

Tools-Diagnostics. Vi dice se il robottino è connesso, il voltaggio della batteria ecc. In pratica è comodo per vedere se il computer “vede” l' NXT e cioè se la connessione USB è attiva.

Tools-Direct control. Permette di impostare i parametri costruttivi del robot. Si possono indicare i sensori (fino a 4) collegati, specificare a quale porta sono connessi ed il metodo di funzionamento che preferiamo abbiano, si possono specificare gli attuatori (fino a 3) e le porte cui sono connessi.

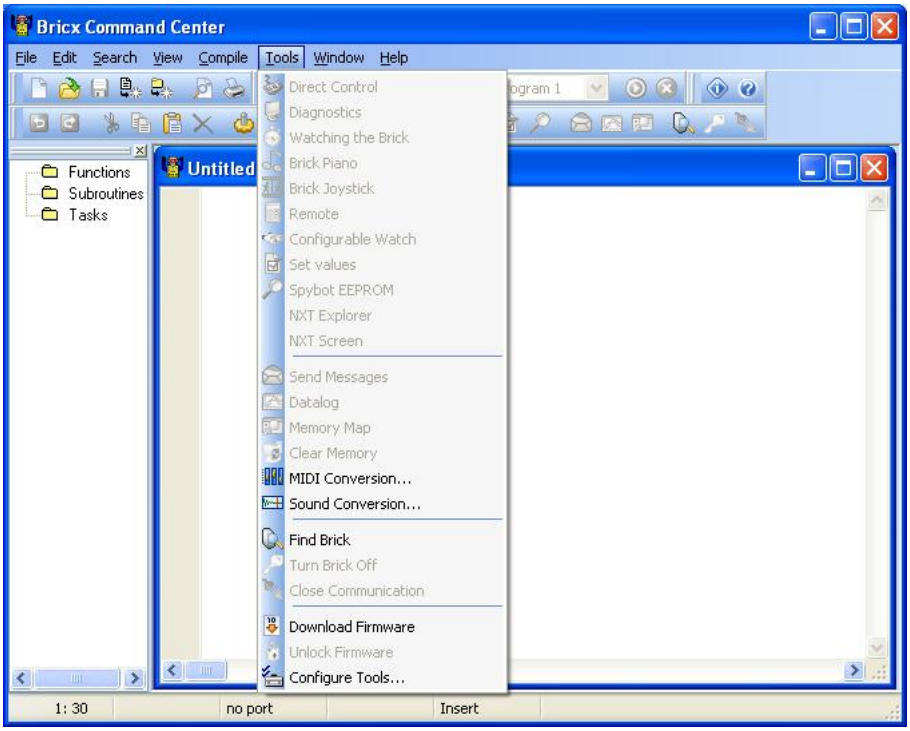
Tools- Watching the Brick. Bello in ogni fase. Tiene sotto controllo i valori dell' NXT e riproduce addirittura un grafico della loro variazione nel tempo. Molto comodo per testare il funzionamento dei programmi una volta realizzati.

La coppia di utilità Direct-Control + Watching the Brick è molto utile. Si specifica come è fatto il robot tramite la prima, e tramite la seconda si può andare a vedere esattamente come si comporta il robot: cosa leggono i suoi sensori, e come si mouovono i suoi motori

Tools-Brick Joistick Muove il robot quando si clicca col mouse su delle icone a forma di freccia: Comodo, dopo aver costruito un robot, per verificare se è montato giusto. Ad esempio se clicchiamo la freccia che va a destra anche il robot deve andare a destra, altrimenti potrebbero essere invertiti i fili che vanno ai motori.

Tools – Brick piano Serve a comporre musichette da trasferire e fare eseguire all' NXT Ci sono dei programmi di conversione che trasformano i comuni file .waw in file comprensibili dall' NXT senza prendersi la briga di comporre nulla, ma questa voce di menu ci permette di comporre dei motivi senza installare altro software.

Tools – Download Firmware Ci aiuta a scaricare sull' NXT il firmware più aggiornato.



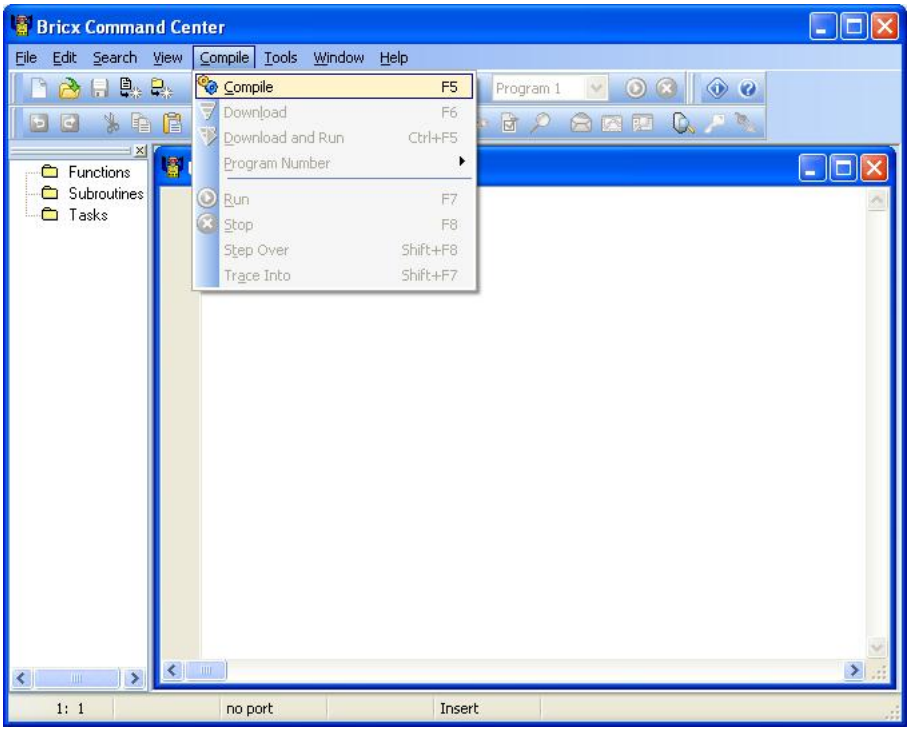
IL Menu “Compile”

E' il cuore del programma. (Ovviamente, il compilatore è fatto principalmente per compilare)

Compile-Compile. Dal nome non lo sospettereste mai:Compila il programma che avete appena scritto.:) Non occorre più che si specifichi cosa significa compilare, vero? Quello che è davvero degno di nota è che, se trova degli errori, il compilatore li segnala con dei messaggi espliciti (che compaiono in basso, sotto la finestra col programma), che già vi aiutano a capire in che punto del programma c'è l'errore e la natura dell' errore stesso, in modo che sia davvero facile andare a correggerlo.

Compile-Download Compila il programma e lo trasferisce contemporaneamente all' NXT. A questo punto basta premere sull' NXT il tasto di avvio del programma ed il robot eseguirà le istruzioni.

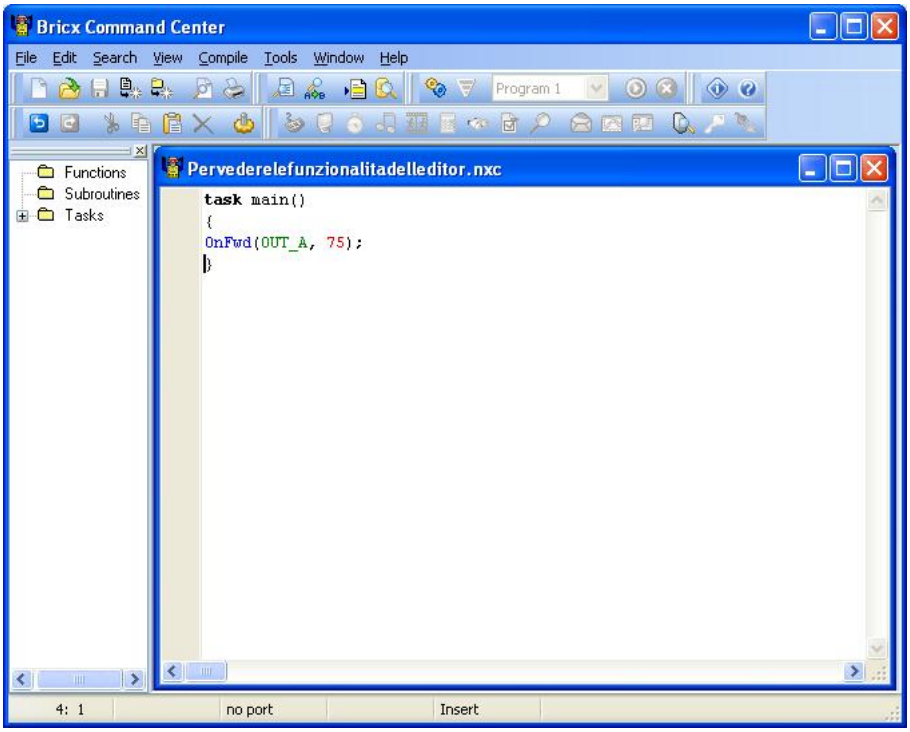
Compile-Download and Run Compila il programma, lo trasferisce all' NXT, e lo esegue immediatamente.



La Finestra Principale

La finestra principale del BRIXCC è foglio bianco, nel quale potete scrivere il programma. È sostanzialmente un editor di testo. Da notare però alcune cosette interessanti ed utili. Se scrivete delle parole “normali”, lui le scrive in testo normale. Se scrivete dei comandi in linguaggio di programmazione (per esempio **task**) lui li riconosce e li evidenzia in grassetto. Se scrivete dei comandi speciali li evidenzia automaticamente in **Blu**. I parametri forniti sono evidenziati invece in **Rosso** .

Questa funzionalità è molto utile per capire, già mentre si sta scrivendo un programma, se sono stati commessi degli errori. Infatti, se stiamo scrivendo un comando, e ci accorgiamo che il testo non diventa **grassetto**, oppure blu, è probabile che la sintassi del comando sia sbagliata, per esempio perché banalmente abbiamo scritto male da tastiera. E' quindi un invito immediato a verificare l'esattezza della nostra digitazione.



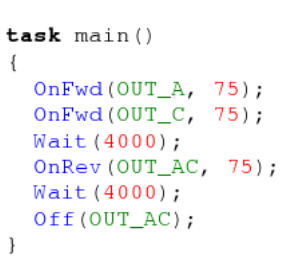
Sommario

In questo primo capitolo abbiamo imparato a conoscere
- Il programma Brixcc.

Scrittura di un primo programma

Un programma è un semplice file di testo, che, al limite, possiamo scrivere con un qualsiasi editor, e che contiene una serie di istruzioni. Grazie al compilatore queste istruzioni vengono tradotte in linguaggio comprensibile al microprocessore del robot.
Per nostra fortuna, come spiegato nel capitolo precedente, le funzioni di scrittura del testo, di compilazione del programma,di correzione degli errori, e di trasferimento al robot, sono svolte dal software **Brixcc**,che ci facilita la vita.
Al momento non sappiamo programmare in NXC, quindi ci limiteremo a scrivere meccanicamente nella finestra di scrittura del programma del brixcc delle cose incomprensibili, e cercheremo poi di analizzarle e comprenderne il significato.

Testo copiabile	Screenshot del BricxCC

`task main()`
`{`
`OnFwd(OUT_A, 75);`
`OnFwd(OUT_C, 75);`
`Wait(4000);`
`OnRev(OUT_AC, 75);`
`Wait(4000);`
`Off(OUT_AC);`
`}`
	

Ogni programma scritto in NXC deve contenere un **"task"** ed in particolare deve contenere il **task** (main) che potremmo definire **"compito principale"**.
I task possono essere anche più di uno, ma, qualora altri task esistano, vengono "chiamati" dal task principale. Ogni task inizia con una parentesi graffa aperta e termina con una graffa chiusa, per inticare rispettivamente l'inizio e la fine delle istruzioni che ne fanno parte. Contenute tra le parentesi, ci sono le istruzioni semplici, dette **statements**.

Analizziamo il codice dall' inizio:
Nella prima riga il programma indica che è'è un task, quello principale: **task** (main)()
C'è la parentesi graffa aperta, ad indicare che quello che segue fa parte del task ({)
C'è una serie di istruzioni semplici contenute tra le parentesi del task (**statements**).
Ogni istruzione termina con un punto e virgola. Un esempio:**OnFwd**(OUT_A, 75);
Per chiarezza di lettura del codice ogni istruzione occupa una riga
C'è la graffa di chiusura del task (})

Vediamo ora, una per una, il significato delle singole istruzioni:

- **OnFwd**(OUT_A, 75);

Questa prima istruzione dice al programma di muovere un motore in avanti **OnFwd** sta per (Accendi=On e muovi in avanti = Forward = Fwd)
Quale motore? (Un modulo NXT può gestirne fino a 4) Quello collegato all' uscita **A** (Out_A)
Con che velocità? (Al 75% della velocità massima) ,75
L'istruzione poi termina con il classico punto e virgola.

- **OnFwd** (OUT_C, 75);

Come l'istruzione precedente, ma attiva il motore collegato all' uscita **C** (Capiamo adesso che le 4 porte di uscita tramite cui il modulo NXT gestisce i motori sono identificate da una lettera maiuscola: **A, B, C, D**)

- **Wait** (4000);

Wait in inglese significa attendi. Abbiamo dato al robot l'istruzione di muovere i motori, ma quanto a lungo deve muoversi il robot?
L'istruzione **wait(4000);** indica che deve muoversi in avanti per 4 secondi esatti.
Infatti l'argomento (4000) dice al processore che deve mantenere attive le uscite per 4000 millesimi di secondo.

- **OnRev** (OUT_AC, 75);

Questa istruzione dice al programma di muovere due motori indietro **OnRev** sta per (Accendi=On e muovi indietro = Reverse = Rev)
Perchè due motori, stavolta? **(OUT_AC)** Perchè questo argomento specifica di attivare contemporaneamente le uscite **A e C**

- **Wait** (4000);

Vedi sopra: esegue il comando impartito per 4 secondi.

- **Off** (OUT_AC);

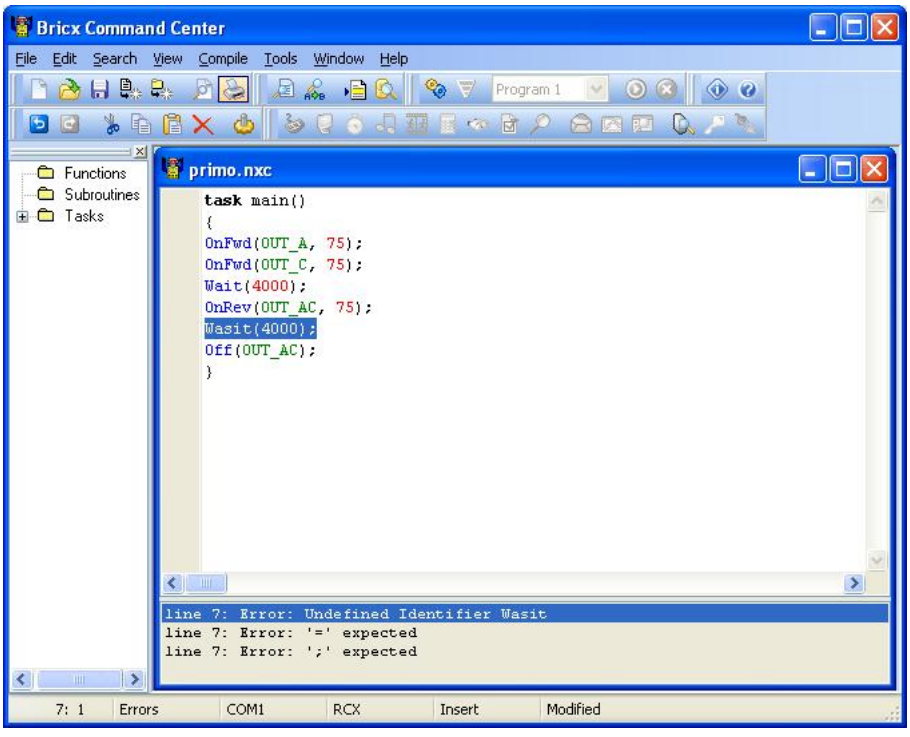
Spegne (Off) i motori collegati alle uscite **A e C**

Perfetto: il nostro primo programma farà muovere il robot in avanti per 4 secondi, poi lo farà tornare indietro per altri 4, ed infine spegnerà i motori. Non è un compito difficilissimo, ma è pur sempre un inizio. Abbiamo capito qualcosa sulle uscite che comandano i motori, abbiamo capito come si possono muovere avanti ed indietro, abbiamo capito che si possono fare andare alla velocità che volgilamo e per il tempo che vogliamo. Non è poi così poco. Abbiamo anche capito che è buona norma spegnere i motori dopo che sono stati usati.

Ora non resta che tradurre le istruzioni a noi familiari in codice intellegibile dal microprocessore, e quindi compilare il nostro programma.
Prima di tutto salviamolo. Tramite i comandi **"File" - "Save as"** del **Brixcc** salviamolo su disco con un nome a piacere e con l'estensione **"nxc"**.
Ora possiamo chiedere al Brixcc di compilare il nostro programma e, se tutto va bene e non ci sono errori, di trasferirlo al robot.
La compilazione si ottiene coi comandi **"Compile" - "Compile"**.
La compilazione seguita dal trasferimento del file al robot, coi comandi **"Compile" - "Download"**
Oppure,al limite, la compilazione con immediato trasferimento ed esecuzione del file compilato da parte del robot tramite **"Compile" - "Download and run"**
E' ovvio specificare che prima di trasferire i programmi al robot è necessario connetterlo al computer sul quale si sta scrivendo il programma tramite un cavo usb.

E se c'è un errore?

Succede spesso: **Brixcc** accetta solo comandi senza errori di sintassi e con parametri compresi in un range ben definito. Un solo errore di battitura , come evidenziato nella finestra sotto, provoca un errore di compilazione.
Brixcc però ci aiuta a trovare e risolvere gli errori in molti modi.
Qui sotto vediamo che, compilando il codice con l'errore di battitura nell' istruzione **"Wait"**, il programma evidenzia la parola errata in blu, e scrive sotto , nella finestra in basso, il numero di linea di codice in cui si è verificato l'errore ed il tipo. Notate che un errore all' interno del codice di un programma ne può generare altri.
Quando ci sono errori multipli si comincia sempre a risolvere il primo in alto nella finesta, e si continua poi verso il basso.
E' possibile che, risolvendo un singolo problema, vengano annullati una serie di successivi errori.
Si noti , per ultimo, che **Brixcc** evidenzia le istruzioni ed i parametri forniti con colorazioni differenti.
Questo è un aiuto non indifferente nella scrittura del codice.
Un errore come quello che si verifica in questo caso può essere individuato all' istante in fase di battitura: il testo dell' istruzione non si colora di blu e quindi ci deve essere qualcosa che non va.



Sommario

In questo primo capitolo abbiamo conosciuto l'ambiente di sviluppo **Brixcc**
- Abbiamo scritto un programma,
- Abbiamo analizzato le sue componenti principali e le singole istruzioni che lo compongono,
- Lo abbiamo salvato e compilato
- Lo abbiamo trasferito al robot,
- Abbiamo parlato del sistema di evidenziazione tramite colori degli statement e dei parametri utilizzato dal **Brixcc**
- Abbiamo conosciuto il debugger, cioè il modo di correggere gli errori.
Abbiamo capito che
- Il modulo NXT possiede 3 uscite, comandate dal microprocessore, ed identificate con le lettere **A,B,C**

Inoltre abbiamo iniziato a conoscere alcune
- Istruzioni del linguaggio NXC :

- **OnFwd**
- **OnRev**
- **Wait**

Un programma più interessante

Il primo programma che abbiamo scritto ci ha aiutati a capire molte cose, ma non è particolarmente entusiasmante: si limita a fare andare il nostro robot avanti per 4 secondi e indietro per altrettanti 4.Vediamo adesso di iniziare a scrivere un codice che ci consenta di far fare al robot quacosa di più divertente

Fare le curve

Per far girare il robot possiamo fermare un motore mentre l'altro continua a funzionare.Il codice seguente dovrebbe far fare al robot una curva approssimativamente di 90 gradi

Testo copiabile	Screenshot del BricxCC

`task main()`
`{`
`OnFwd(OUT_AC, 75);`
`Wait(800);`
`OnRev(OUT_C, 75);`
`Wait(360);`
`Off(OUT_AC);`
`}`
	`task main()`
	`{`
	`  OnFwd(OUT_AC, 75);`
	`  Wait(800);`
	`  OnRev(OUT_C, 75);`
	`  Wait(360);`
	`  Off(OUT_AC);`
	`}`

Analizziamone il codice:

- Si apre il **task main** (dopodichè si apre una parentesi graffa)
- Si fanno girare in avanti i motori alle uscite **A e C** con "velocità" 75 (il 75 % del massimo)
- Si aspettano 800 millesimi di secondo (0,8 secondi)
- Si inverte il moto del motore all' uscita **C** lasciando inalterata la velocità di rotazione
- Si aspettano 360 millesimi di secondo (0,36 secondi)
- Si spengono i motori
- Si chiude il **task** (chiusa parentesi graffa)

Scriviamo il codice,compiliamolo,trasferiamolo al robot e facciamolo eseguire. L'angolo di rotazione sarà circa di 90 gradi, ma probabilmente non esattamente 90, dipende dal tipo di supeficie su cui si muove il robot. Proviamo a cambiare il valore del secondo **Wait**, (attualmente 360), fino a quando l'angolo non è esattamente quello che desideriamo. In questo modo possimo adattare il programma al tipo di superficie.

Dovendo intervenire più volte sullo stesso programma ci risulta più comodo un altro sistema: non usiamo direttamente valori numerici come argomento dell' istruzione **Wait** ma sostituiamoli con qualcosa di diverso.

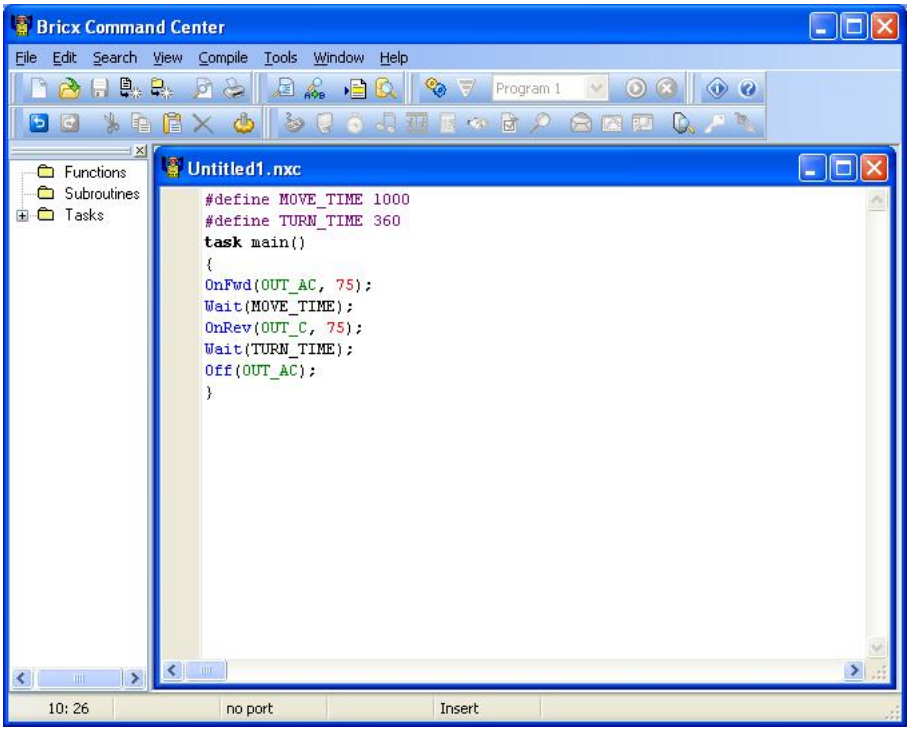
Ecco qui sotto un codice di esempio:

Testo copiabile	Screenshot del BricxCC

`#define MOVE_TIME =1000`
`#define TURN_TIME =360`
 	
`task main()`
`{`
`OnFwd(OUT_AC, 75);`
`Wait(MOVE_TIME);`
`OnRev(OUT_C, 75);`
`Wait(TURN_TIME);`
`Off(OUT_AC);`
`}`
	`#define MOVE_TIME  1000`
	`#define TURN_TIME   360`
	
	`task main()`
	`{`
	`  OnFwd(OUT_AC, 75);`
	`  Wait(MOVE_TIME);`
	`  OnRev(OUT_C, 75);`
	`  Wait(TURN_TIME);`
	`  Off(OUT_AC);`
	`}`

In questo codice le prime due righe definiscono due **COSTANTI**. Abbiamo ciè associato,prima che il programma cominci,ai termini **MOVE_TIME** e **TURN_TIME**, dei valori numerici.
Durante l'esecuzione del programma possiamo fare riferimento ai termini così definiti, ed il programma li considererà come un valore numerico.
L'uso delle costanti rende il programma più facile da interpretare, ed al contempo da modificare.

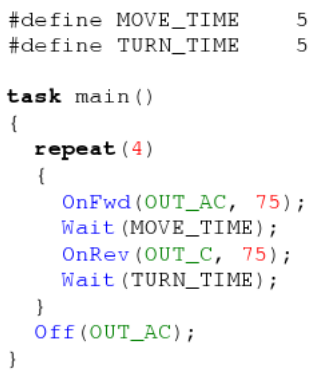
Come si nota nell' immagine sottostante, il **Brixcc** ha un colore particolare (magenta) con cui evidenzia il codice di dichiarazione delle costanti.



Ripetizione di comandi

Ora, ipotizziamo di voler far percorrere un quadrato al nostro robot. Sappiamo come procedere in linea retta, sappiamo fargli fare le curve di 90 gradi, quindi basterebbe scrivere un codice in cui sono contenute alternativamente le istruzioni per andare dritto e per curvare ad angolo retto. Ciò è certamente possibile, ma è noioso e porta il codice ad essere inutilmente lungo. Possiamo, invece, utlilizzare l'istruzione **Repeat ()**. Questa istruzione ripete, per un numero di volte uguale al numero tra parentesi, una istruzione od una serie di istruzioni. Ecco qui sotto un programma che fa muovere il robot su un percorso quadrato e che utilizza l'istruzione **Repeat ()**.

Testo copiabile	Screenshot del BricxCC

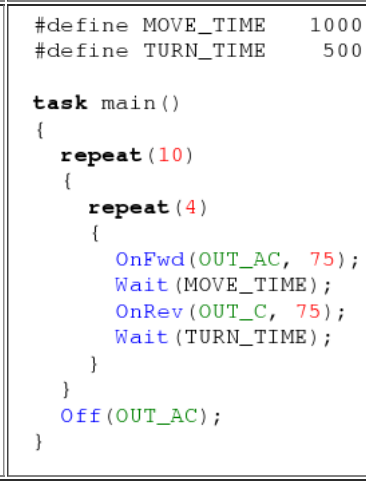
`#define MOVE_TIME 500`
`#define TURN_TIME 500`
 	
`task main()`
`{`
`repeat(4)`
`{`
`OnFwd(OUT_AC, 75);`
`Wait(MOVE_TIME);`
`OnRev(OUT_C, 75);`
`Wait(TURN_TIME);`
`}`
`Off(OUT_AC);`
`}`
	

Subito dopo l'istruzione **Repeat (x)** si apre una parentesi graffa , segue una serie di istruzioni, e poi una parentesi graffa chiusa. Le istruzioni comprese tra le parentesi graffe saranno ripetute un x numero di volte.

Notate che il codice qui sopra ha una particolarità: le righe comprese tra le parentesi graffe relative al **task main()** sono rientrate leggermente verso destra , quelle comprese tra le parentesi graffe relative a **repeat (4)** sono ancor più rientrate. Questo tipo di scrittura del codice è definito **indentazione**: non ha nessun effetto pratico sulla funzionalità del programma, un codice con indentazione si comporterà una volta compilato ed eseguito allo stesso modo di un codice non indentato, ma l'indentazione aumenta grandemente la leggibilità del software da parte dei programmatori. Infatti, usando correttamente l'indentazione,è molto semplice capire, già a colpo d'occhio, dove iniziano e dove finiscono i cicli od i task.

Ecco qui sotto un altro codice , un po' più complesso, contenente cicli:

Testo copiabile	Screenshot del BricxCC

`#define MOVE_TIME 500`
`#define TURN_TIME 500`
 	
`task main()`
`{`
`repeat(10)`
`{`
`repeat(4)`
`{`
`OnFwd(OUT_AC, 75);`
`Wait(MOVE_TIME);`
`OnRev(OUT_C, 75);`
`Wait(TURN_TIME);`
`}`
`}`
`Off(OUT_AC);`
`}`
	

Questo programma contiene due cicli **"nidificati"**: il ciclo più esterno contiene il ciclo più interno: il ciclo esterno viene eseguito 10 volte, mentre quello interno viene eseguito 40 volte.
Notate che l'indentazione aiuta notevolmente a capire dove iniziano e dove finiscono i cicli.
Quello sotto è lo stesso identico programma ma senza indentazione. Non è molto più difficile comprendere esattamente cosa fa?

Testo copiabile	Screenshot del BricxCC

`#define MOVE_TIME 500`
`#define TURN_TIME 500`
`task main()`
`{`
`repeat(10)`
`{`
`repeat(4)`
`{`
`OnFwd(OUT_AC, 75);`
`Wait(MOVE_TIME);`
`OnRev(OUT_C, 75);`	

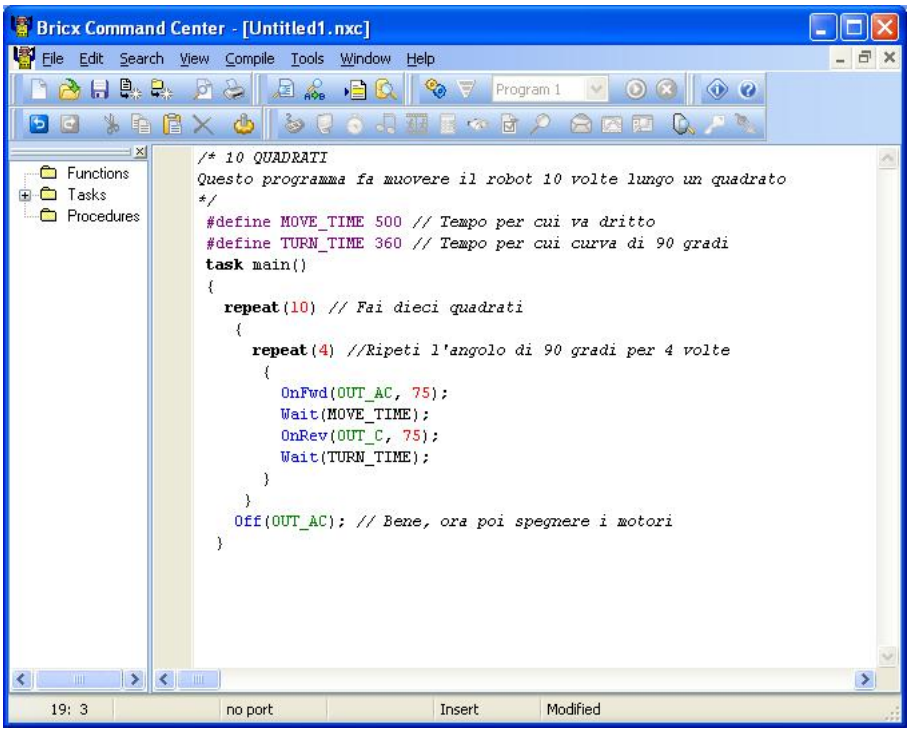
```
  Wait(TURN_TIME);
        }
        }
    Off(OUT_AC);
        }
```

```
#define MOVE_TIME 500
#define TURN_TIME 500
task main()
{
repeat(10)
{
repeat(4)
{
OnFwd(OUT_AC, 75);
Wait(MOVE_TIME);
OnRev(OUT_C, 75);
Wait(TURN_TIME);
}
}
Off(OUT_AC);
}
```

Commenti

Per rendere il nostro codice ancora più leggibile è possibile aggiungere dei **commenti**. I commenti sono delle parti di testo aggiunte al programma, in cui il programmatore spiega, come promemoria per sè medesimo, o come spiegazione per altri che dovranno interpretare o completare il suo lavoro. Per aggiungere un commento bisogna far capire al compilatore che non deve "tradurre" quella parte di codice, ma che deve bellamente ignorarlo. Perchè il compilatore capisca che il testo seguente deve essere ignorato, basta farlo precedere da un simbolo speciale, in modo che il compilatore sappia che "è roba destinata agli umani" e non deve essere presa in considerazione.
Sono stati individuati due speciali simboli: se una riga di testo comincia con il simbolo // tutto il restante della riga viene ignorato dal compilatore.
Se il commento è lungo, e si compone di più righe di testo, si fa cominciare dal simbolo /* e terminare con */ .

Ecco qui sotto un esempio di utilizzo dei commenti per rendere più comprensibile un programma, in cui sono utilizzati entrambi i sistemi su esposti. Fate caso a come il **Brixcc** mostra il testo del commento.



Sommario

In questo capitolo abbiamo imparato ad usare le
- **Costanti**

Abbiamo inoltre preso familiarità con l'istruzione
- **Repeat** e coi cicli. Abbiamo utilizzato cicli semplici e nidificati. Abbiamo conosciuto l'utilità della
- **Indentazione**, che rende più leggibili i programmi. Sempre riguardo a questo argomento abbiamo imparato ad usare i
- **Commenti**, per arricchire il codice con le nostre annotazioni e renderlo comprensibile a noi stessi ed agli altri.

Uso delle Variabili

L'uso delle variabili è un aspetto molto importante della programmazione dei robot e non solo. Usare una variabile è un modo per immagazzinare nella memoria del robot un dato. Si utilizza dunque una parte della memoria per immagazzinarvi qualcosa, che verrà poi utile durante l' esecuzione del programma: un numero intero, un numero con la virgola,un carattere, una serie di caratteri. Possiamo immaginare la memoria del robot come una serie di cassetti vuoti: usare una variabile significa assegnare un nome ad un cassetto,ed inserire nel cassetto quello che si vuole:in un secondo momento, quando avremo bisogno di usare quanto immagazzinato, sarà molto facile ritrovarlo.

Movimento a spirale

Mettiamo il caso che vogliamo modificare il programma scritto sopra,quello che fa percorrere un quadrato al robot, facendo in modo, questa volta, che percorra una spirale:

Testo copiabile	Screenshot del BricxCC

```
#define TURN_TIME 360
int move_time; // la variabile viene dichiarata
task main()
{
move_time = 200; // definisce il valore iniziale
repeat(50)
{
OnFwd(OUT_AC, 75);
Wait(move_time); //usa la variabile per definire il tempo di attesa
OnRev(OUT_C, 75);
Wait(TURN_TIME);
move_time += 200; // incrementa la variabile
}
Off(OUT_AC);
}
```

```
#define TURN_TIME   360

int move_time;               // define a variable

task main()
{
  move_time = 200;            // set the initial val
  repeat(50)
  {
    OnFwd(OUT_AC, 75);
    Wait(move_time);         // use the variable for
    OnRev(OUT_C, 75);
    Wait(TURN_TIME);
    move_time += 200;          // increase the varia
  }
  Off(OUT_AC);
}
```

Le linee di codice interessanti sono indicate da commenti:

- **int move_time;** all' inizio del codice dichiariamo una variabile. Poniamo attenzione ad ogni dettagio della riga di dichiarazione:
- **int** significa che la variabile che verrà dichiarata è di tipo intero, questo significa che potrà contenere solo numeri interi (es. 10 oppure 1037) ma non numeri con virgola (es.14,4)
- **move_time** è il nome della variabile. Un nome di vadiabile può essere lungo a piacere, può contenere lettere, numeri, e un carattere speciale: underscore, e deve obbligatoriamente iniziare con una lettera.Le stesse regole valgono anche per i nomi delle costant , dei task, eccetera.
- **move_time = 200;** in questa riga poniamo il numero intero 200 all' interno della variabile. Il simbolo = significa quindi "prendi il valore e mettilo nella variabile"

A questo punto il programma ricorderà sempre, fino a quando gli assegneremo un diverso valore, che dentro la variabile **move_time** è memorizzato il numero 200. Il programma poi prosegue, e viene ereguito un ciclo. All' interno del ciclo l' istruzione **wait (move_time)** sarà equivalente a **wait (200)** proprio in quanto dentro ***move_time** c'è adesso il numero 200. Il robot quindi eseguirà il primo ciclo,si muoverà dritto e poi farà un angolo di 90 gradi ma alla fine del ciclo troverà l'istruzione:

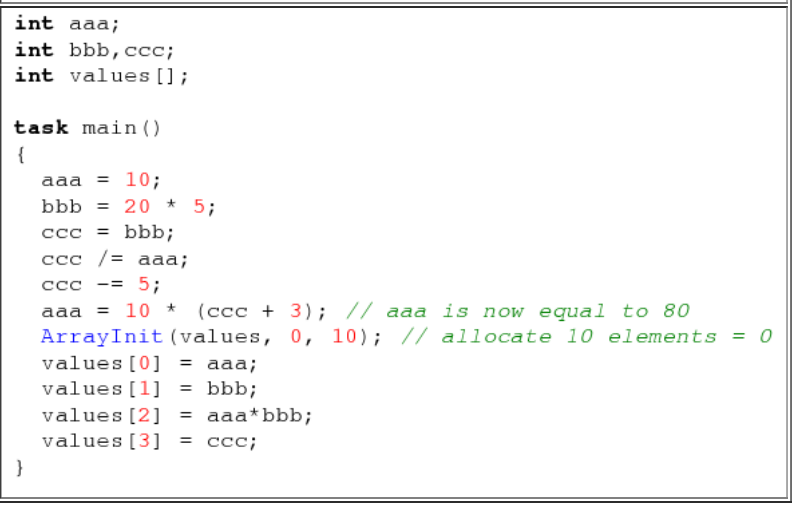
- **move_time** += 200; Questa istruzione significa: somma alla variabile **move_time** il numero 200. Ovviamente, dopo aver eseguito l'istruzione, in **move_time** ci sarà 400.

Ma l'istruzione **move_time** += 200; fa parte del ciclo che viene eseguito 50 volte, per cui ad ogni esecuzione del ciclo la variabile verrà incermentata del valore 200. Questo fa sì che le linee rette percorse dal robot tra una curva e l'altra diventino sempre più lunghe, e cioè, all' atto pratico, che il robot percorra una spirale.
Per scrivere questo codice abbiamo sommato ad una variabile il valore 200 per parecchie volte. Ma possiamo anche moltiplicare il valore di una variabile, oppure sottrargli una cifra , oppure dividerlo, oppure effettuare operazioni matematiche che coinvolgono più variabili. Abbiamo la massima libertà, purchè rispettiamo i tipi delle variabili.

Array di variabili

Il programma che segue non esegue un compito preciso: è un programma che useremo per vedere in pratica alcune operazioni sulle variabili e soprattutto per comprendere che cos'è un **Array** e come lo si crea ed utilizza nel linguaggio **NXC**.
Un **Array** è una serie di variabili indicizzata, cioè una serie contigua di "cassetti" di memoria, contrassenati da un indice numerico. In essi possiamo memorizzare con grande facilità una serie di valori, che possiamo poi andare a recuperare quando ci saranno utili. Un **Array** è pur sempre una serie di variabili, e quindi bisogna dichiarare, all' atto della sua creazione, il tipo di dati che vi sarà contenuto.

Testo copiabile	Screenshot del BricxCC

`int aaa;`
`int bbb,ccc;  //due variabili dichiarate assieme: si può.`
`int values[];  //la dichiarazione di un array`
`task main()`
`{`
`aaa = 10;`
`bbb = 20 * 5;`
`ccc = bbb;`
`ccc /= aaa;`
`ccc -= 5;`
`aaa = 10 * (ccc + 3); // aaa ora vale 80`
`ArrayInit(values, 0, 10); // alloca 10 elementi e ponili = 0`
`values[0] = aaa;`
`values[1] = bbb;`
`values[2] = aaa*bbb;`
`values[3] = ccc;`
`}`
	

Analizziamo il codice: ci sono importanti novità di cui tenere conto.
Nella seconda riga notiamo che si possono dichiarare più variabili sulla stessa linea di codice. Nella terza riga compare un costrutto nuovo: **values []**: non una semplice variabile ma un **array**. Come vedete è dichiarato con un nome, come se fosse una variabile semplice, ma è seguito da due parentesi quadre []. Dentro al programma, prima di scrivere dei valori nelle variabili costituenti l'array, esse vengono allocate (create) ed inizializzate, cioè poste uguali a zero col seguente comando:
ArrayInit(values, 0, 10); dopo di che possono essere utilizzate per immagazzinare dei valori.

Numeri casuali

Fin qui il nostro robot ha eseguito alla lettera i compiti che gli abbiamo assegnato. Le cose cambiano, e diventano più unteressanti, se il robot è dotato, per cosiddire, di una sua autonomia, se si comporta in maniera autonoma ed imprevista. Per far ciò la cosa più semplice è ricorrere alla casualalità.Per produrre un comportamento casuale, niente è più semplice che generare un numero a caso e legare il comportamento del robot a questo numero. Il programma seguente va andare a spasso il robot secondo un itinerario casuale: esso si muoverà in avanti per un tratto di lunghezza imprecisata , e farà poi una curva, anch'essa a caso.

Testo copiabile	Screenshot del BricxCC

`int move_time, turn_time;`
`task main()`
`{`
`while(true)`
`{`
`move_time = Random(600);`
`turn_time = Random(400);`
`OnFwd(OUT_AC, 75);`
`Wait(move_time);`
`OnRev(OUT_A, 75);`
`Wait(turn_time);`
`}`
`}`
	

Il programma definisce due variabili: move_time (per l'avanzamento rettilineo) e turn_time (per la curva), ed assegna ad esse dei valori casuali di volta in volta.
Random(600) genera un numero casuale compreso tra zero e 600. Ogni volta che la funzione viene chiamata genera un numero differente.

Notate che potreste tranquillamente ottenere lo stesso risultato senza ricorrere all' uso delle variabili, utilizzando:
Wait (Random(600));
in luogo di

move_time = Random(600)
Wait (move_time);

Approfondimento: Un metodo per calcolo di pi greco che fa uso dei numeri casuali
http://www2.polito.it/didattica/polymath/htmlS/argoment/APPUNTI/TESTI/Apr_03/Cap17.html

Nell' esempio che abbiamo fatto sopra è stato utilizzato un ciclo nuovo e differente.
Non si fa più uso dell' istruzione **repeat** , ma si utlizza l'istruzione **while (true).**

L'istruzione **while (true)** esegue la serie di istruzioni successive se è vera la condizione posta tra parentesi.

L' espressione **true** è vera per definizione, per cui il ciclo continua all' infinito.

Maggiori approfondimenti sull' istruzione **while** li troveremo nel capitolo **6.**

Sommario

In questo capitolo avete imparato l'uso delle
- Variabili e degli
- Array.
Potete dichiararne anche di tipo differente da **int**, ad esempio **short**, **long**, **byte**, **bool**, e **string**.
Avete familiarizzato con l'uso dei numeri casuali, per dare al robot un comportamento imprevedibile.
In ultimo, avete imparato ad usare una seconda istruzione per eseguire un ciclo: l' istruzione
- while, per far eseguire un ciclo un numero infinito di volte.

Strutture di controllo

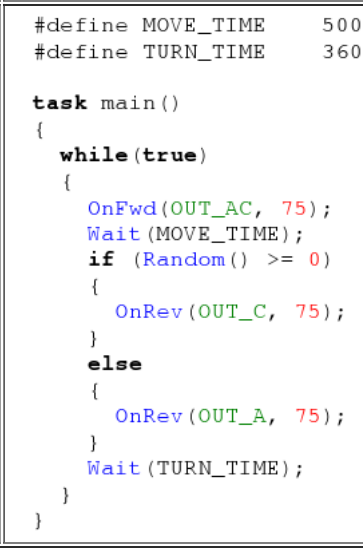
Nel paragrafo precedente abbiamo familiarizzato con le istruzioni **repeat** e **while**. Queste istruzioni definiscono in che maniera eseguire altre istruzioni, e si definiscono dunque "istruzioni di controllo". In questo capitolo analizzeremo qualche altra struttura di questo tipo.

L'istruzione IF

A volte desideriamo che alcune istruzioni del nostro programma siano eseguite solo in certe particolari situazioni. In questo caso utilizziamo l'istruzione **if**. Facciamo un esempio:modifichiamo ulteriormente il programma che abbiamo utilizzato finora. Vogliamo far percorrere al robot un tratto rettilineo, e poi gli vogliamo far fare una curva, a destra oppure a sinistra. Per far questo utilizziamo ancora l'istruzione **random**: facciamo generare al programma un numero casuale che può essere maggiore o minore di zero. Se il numero casuale è maggiore di zero, il robot gira a sinistra, se minore di zero il robot gira a destra.

Ecco il codice del programma.

Testo copiabile	Screenshot del BricxCC

`#define MOVE_TIME 500`
`#define TURN_TIME 360`
 	
`task main()`
`{`
`while(true)`
`{`
`OnFwd(OUT_AC, 75);`
`Wait(MOVE_TIME);`
`if (Random() >= 0)`
`{`
`OnRev(OUT_C, 75);`
`}`
`else`
`{`
`OnRev(OUT_A, 75);`
`}`
`Wait(TURN_TIME);`
`}`
`}`
	

L'istruzione **if** sembra simile all' istruzione **while**. Se la condizione è vera la parte di programma racchiusa tra le parentesi graffe viene esegita, mentre, in caso contrario, veine eseguita la serie di istruzioni preceduta dall' istruzione **else**
Analizziamo un po' meglio l' istruzione condizionale che abbiamo usato:
Random( ) >= 0. Significa che **Random( )** deve essere maggiore o uguale a zero perchè la condizione sia vera. Ci sono un sacco di modi, nel nostro linguaggio, per comparare dei valori, eccone un elenco:

```
==      Uguale a

<       minore

>       maggiore

<=      minore o ugule

>=      maggiore o uguale

!=      diverso da
```

Nota: ora conoscete l'istruzione "=" e l'istruzione "==". Confrontatene il significato

possiamo pure combinare queste comparazioni utilizzando && (che significa "e") o || (che significa "o").
Ecco alcuni esempi di combinazioni di condizioni:

```
true  sempre vera

false  sempre falsa

ttt!= 3 vera se ttt è diverso da 3

(ttt>= 5) && (ttt<=10) vera se ttt è maggiore o uguale a 5 e se ttt è minore o uguale a 10 (cioè se compreso tra 5 e 10)

(aaa==10) || (bbb ==) vera se aaa è uguale a 10 oppure se bbb è uguale a 10
```

Fate caso che l'istruzione if ha due componenti: una componente immediatamente successiva all' istruzione, che viene eseguita nel caso in cui la condizione dopo **if** sia vera, e la parte seguente all' istruzione **else**, che viene eseguita nel caso la condizione successiva ad if sia falsa.

L'istruzione do

Un' altra istruzione di controllo è l'istruzione **do**, che ha la forma seguente

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

Le istruzioni tra le parentesi graffe successive all' istruzione **do** vengono eseguite fin quando è vera la condizione (successiva all' istruzione **while**).

Qui sotto un esempio di programma che usa l'istruzione **do**. Il robot si muove a casaccio per un tempo di 20 secondi, poi si ferma.

Testo copiabile	Screenshot del BricxCC

`int move_time, turn_time, total_time;`
`task main()`
`{`
`total_time = 0;`
`do`
`{`
`move_time = Random(1000);`
`turn_time = Random(1000);`
`OnFwd(OUT_AC, 75);`
`Wait(move_time);`
`OnRev(OUT_C, 75);`
`Wait(turn_time);`
`total_time += move_time;`
`total_time += turn_time;`
`}`
`while (total_time < 20000);`	

Off(OUT_AC);
}
 	
	int move_time, turn_time, total_time;
	
	task main()
	{
	  total_time = 0;
	  do
	  {
	  move_time = Random(1000);
	  turn_time = Random(1000);
	  OnFwd(OUT_AC, 75);
	  Wait(move_time);
	  OnRev(OUT_C, 75);
	  Wait(turn_time);
	  total_time += move_time;
	  total_time += turn_time;
	  }
	  while (total_time < 20000);
	  Off(OUT_AC);
	}

Notate che l'istruzione si comporta quasi come l'istruzione **while**, tranne che per una particolarità: nell' istruzione **if** si controlla immediatamete se la condizione è vera, mentre quando si utilzza l'istruzione **do** si controlla alla fine dell' esecuzione delle istruzioni. Quindi, nel caso dell' istruzione **do**, le istruzioni tra parentesi graffe verranno eseguite con certezza almeno una volta.

Sommario

In questo capitolo abbiamo familiarizzato con due nouve strutture di controllo: l'istruzione
- if
e l'istruzione
- do.
Assieme alle istruzioni **repeat** e **while** queste istruzioni ci permettono di controllare la maniera secondo cui il programma viene eseguito.
E' molto importante che capiate cosa fanno, e sappiate utilizzare bene queste istruzioni, perciò provate a fare alcune esercitazioni da soli prima di continuare il corso.

Sensori

Possiamo connettere dei sensori all' NXT per consentirgli di agire in base ad informazioni provenienti dall' esterno prima che io vi spieghi come, dovete modificare il robot, aggiungendo un sensore di tocco.

Come prima, fare riferimento al manuale di istruzione per modificare il vostro tribot aggiungendo un sensore frontale.

In attesa di un valore dal sensore

Partiamo con un esempio elementare, nel quale il robot si muove diritto in avanti, finche non si imbatte in qualcosa.

Testo copiabile	Screenshot del BricxCC

task main()
{
SetSensor(IN_1,SENSOR_TOUCH);
OnFwd(OUT_AC, 75);
until (SENSOR_1 == 1);
Off(OUT_AC);
}
	task main()
	{
	  SetSensor(IN_1,SENSOR_TOUCH);
	  OnFwd(OUT_AC, 75);
	  until (SENSOR_1 == 1);
	  Off(OUT_AC);
	}

Ci sono due righe importanti in questo codice: La prima riga del programma dice al robot che tipo di sensore è connesso.
IN_1 è il numero della porta cui è connesso il sensore. Le altre porte di ingresso sono ovviamente chiamate IN_2, IN_3 e IN_4. SENSOR_TOUCH indica che alla

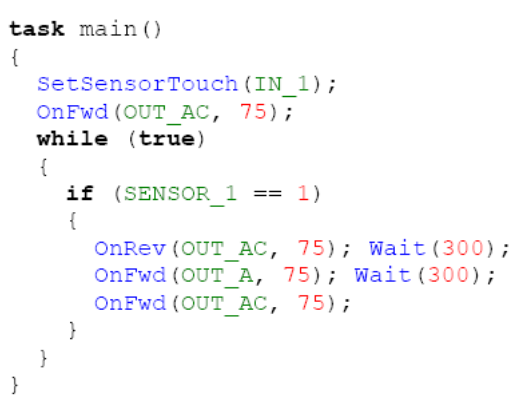
porta IN_1 è connesso un sensore di tocco.Se avessimo connesso un sensore di luce avremmo dovuto indicare SENSOR_LIGHT.Dopo aver specificato il tipo di sensore, il programma avvia entrambi i motori ed il robot inizia a mouversi in avanti. La prossima istruzione è davvero utlile: attende finchè la condizione specificata tra parentesi si avvera.

La condizione specificata in questo caso è ce il valore letto dal sensore SENSOR_1 deve valere 1, e cioè che il sensore di tocco sia stato premuto. Se il sensore non è premuto, il valore letto sarà 0. Dunque, westa istruzione aspetta fino a quando il sensore di tocco viene premuto;se il sensore è premuto i motori sono spenti ed il task termina.

Utilizzare un sensore di contatto

Vediamo adesso come si può fare per fare evitare al robot un ostacolo. Quando il robot urta un ostacolo, lo faremo indietreggiare un po' poi girare, e infine continuare il suo percorso. Ecco il programma:

Testo copiabile	Screenshot del BricxCC

`task main()`
`{`
`SetSensorTouch(IN_1);`
`OnFwd(OUT_AC, 75);`
`while (true)`
`{`
`if (SENSOR_1 == 1)`
`{`
`OnRev(OUT_AC, 75); Wait(300);`
`OnFwd(OUT_A, 75); Wait(300);`
`OnFwd(OUT_AC, 75);`
`}`
`}`
`}`
	

Come nell' esempio precedente, per prima cosa indichiamo il tipo di sensore connesso. Quindi, il robot parte diritto in avanti. Durante il ciclo infinito, noi controllando di continuo se il sensore è stato premuto. Se ciò accade, il robot si muove all' indietro per 300 millisecondi,gira a destra per 300 millisecondi, e ricomincia ad avanzare diritto.

Sensore di luce

Oltre al sensore di tocco, l'NXT può essere collegato anche a dei sensori di luce, di suono, oppure a dei sensori di ultrasuoni. Il sensore di luce può essere regolato in modo da emettere egli stesso luce, oppure no, in modo da misurare la luce riflessa da una superficie oppure la luce ambientale.
Utilizzare il sensore per misurare la luce riflessa è molto utile quando vogliamo fargli seguire una linea tracciata sul pavimento.
E' quello che faremo nell'esempio seguente. (Per poter continuare a sperimentare, finiamo la costruzione del Tribot.)
Connettiamo il sensore di luce alla porta 3, il sensore sonoro alla 2, ed il sensore di ultrasuoni alla 4.
Abbiamo anche bisogno di una superficie chiara, su cui è riportata una traccia scura, che il robot possa riconoscere e seguire.Possiamo utilizzare quella in dotazione all' NXT oppure autocostruirla. Il principio di funzionamento della routine utilizzata per far seguire la linea scura al robot è il seguente: il robot cerca di rimanere col sensore sul bordo della linea scura. In questa zona la luce riflessa ha un valore medio: metà della luce prodotta dal sensore si riflette sulla superficie chiara esterna alla traccia, l'altra metà sulla superficie scura della traccia stessa.
Ogni variazione di traiettoria rispetto alla direzione corretta da seguire comporterà un aumento di luminosità (se il sensore va a finire completamente sulla parte chiara esterna alla linea), oppure una diminuzione di luminosità (se il sensore si trova completamente sopra al nero).

Qui sotto c'è un esempio di un semplice programma in grado di seguire una linea tramite un valore di soglia di luminosità.

Testo copiabile	Screenshot del BricxCC

`#define THRESHOLD 40`
`task main()`
`{`
`SetSensorLight(IN_3);`
`OnFwd(OUT_AC, 75);`
`while (true)`
`{`
`if (Sensor(IN_3) > THRESHOLD)`
`{`
`OnRev(OUT_C, 75);`
`Wait(100);`
`until(Sensor(IN_3) <= THRESHOLD);`
`OnFwd(OUT_AC, 75);`	

```
    }
  }
}
```

```
#define THRESHOLD 40

task main()
{
  SetSensorLight(IN_3);
  OnFwd(OUT_AC, 75);
  while (true)
  {
    if (Sensor(IN_3) > THRESHOLD)
    {
      OnRev(OUT_C, 75);
      Wait(100);
      until(Sensor(IN_3) <= THRESHOLD);
      OnFwd(OUT_AC, 75);
    }
  }
}
```

La prima cosa che fa il programma è dire all' NXT che sulla porta 3 c'è un sensore di luce.Poi, ordina al robot di avanzare ed entra in un ciclo infinito.
Quando il valore di luce letta dal sensore è maggiore di 40 (usiamo una costante, in questo caso, perché il parametro è adattabile: dipende dalle condizioni di luce ambientale che possono cambiare) uno dei motori viene mosso al contrario, fino a quando il sensore sarà riposizionato sulla traccia ed il valore della luce letta sarà corretto.
Vi accorgerete che l'andatura del robot non sarà molto scorrevole, quindi aggiungete l'istruzione Wait (100) prima del comando UNTIL , per rendere l'avanzamento del robot un po' migliore.
Il programma funziona, ma non per movimenti in senso antiorario. Per renderlo universale abbiamo bisogno di complicarlo un pochino.
Per leggere la luce ambientale, con il led spento, configuriamo il sensore nel modo seguente:

Testo copiabile	Screenshot del BricxCC

`SetSensorType(IN_3,IN_TYPE_LIGHT_INACTIVE);`
`SetSensorMode(IN_3,IN_MODE_PCTFULLSCALE);`
`ResetSensor(IN_3);`
	`SetSensorType(IN_3,IN_TYPE_LIGHT_INACTIVE);`
	`SetSensorMode(IN_3,IN_MODE_PCTFULLSCALE);`
	`ResetSensor(IN_3);`

Sensore di suono

Usando il sensore di suono, potrete trasformare il vostro costoso robot NXT in un interruttore sonoro :)
Scriveremo un programma che aspetta un forte suono, e fa avanzare il robot fino a quando esso udrà un altro suono. Collegate il sensore di suono alla porta di ingresso N° 2, come specificato nelle istruzioni di montaggio del Tribot

Testo copiabile	Screenshot del BricxCC

`#define THRESHOLD 40`
`#define MIC SENSOR_2`
`task main()`
`{`
`SetSensorSound(IN_2);`
`while(true)`
`{`
`until(MIC > THRESHOLD);`
`OnFwd(OUT_AC, 75);`
`Wait(300);`
`until(MIC > THRESHOLD);`
`Off(OUT_AC);`
`Wait(300);`
`}`
`}`
	`#define THRESHOLD 40`
	`#define MIC SENSOR_2`
	`task main()`
	`{`
	`SetSensorSound(IN_2);`
	`while(true){`
	`until(MIC > THRESHOLD);`
	`OnFwd(OUT_AC, 75);`
	`Wait(300);`
	`until(MIC > THRESHOLD);`
	`Off(OUT_AC);`
	`Wait(300);`
	`}`
	`}`

Per prima cosa definiremo una costante di soglia (Treshold) ed un alias per SENSOR_2
Nel Main Task configuriamo la porta 2 per leggere il sensore di suono, e facciamo poi partire un ciclo infinito. Usando l'istruzione **until**, il programma aspetta che il livello sonoro diventi maggiore della soglia che abbiamo scelto. Fate caso che SENSOR_2 non è semplicemente un nome, ma è una macro, che quando è chiamata restituisce il valore sonoro letto dal sensore. Se viene captato un forte suono il robot parte diritto, fin quando un altro suonoforte lo fermerà.
E' stata inserita un' istruzione **wait** , affinchè il robot non parta e si fermi istantaneamente. In effetti,in asenza di questa istruzione l' NXT è così veloce nel leggere il

sensore, che un singolo suono anche di durata brevissima viene letto più volte dal sensore, col risultato di attivare e subito disattivare il movimento del robot. Provate a commentare e decommentare la prima e la seconda istruzione **wait**, e potrete capire meglio il concetto.
Una alternativa all' uso dell' istruzione **until** per bloccare l'esecuzione del codice fino a quando una certa condizione è verificata, è l'uso dell' istruzione **while**.
Usando **while**, è necessario porre tra parentesi la condizione che si deve verificare per uscire dal ciclo. ES: **while** (MIC <= THRESHOLD).
Non c'è molto altro da sapere sui sensori analogici dell' NXT; ricordate solamente che sia il sensore di luce che di suono restituiscono valori che sono compresi tra lo 0(zero) ed il 100(cento).

Sensore di Ultrasuoni

Il sensore di ultrasuoni lavora come un radar. Per capirci, esso invia dei segnali ultrasonici ed attende che tornino riflessi dall'oggetto di cui vuole misurare la distanza. Il tempo intercorso tra la partenza ed il ritorno dei segnali,visto che la velocità del suono in aria è nota , darà una misura abbastanza corretta della distanza dell'oggetto.

Questo tipo di sensore è un sensore digitale, intendendo con ciò che contiene in sé un circuito autonomo in grado di calcolare le distanze in base al tempo impiegato dal segnale sonoro ad andare e tornare.(cioè ha un "cervello" proprio, in grado di preelaborare i dati raccolti prima di trasferirli all' NXT).
Con questo sensore possiamo realizzare un robot che riveli ed eviti ostacoli senza toccarli, come accade invece col sensore di tocco.

Testo copiabile	Screenshot del BricxCC

`#define NEAR 15 //cm`
`task main()`
`{`
`SetSensorLowspeed(IN_4);`
`while(true)`
`{`
`OnFwd(OUT_AC,50);`
`while(SensorUS(IN_4)>NEAR);`
`Off(OUT_AC);`
`OnRev(OUT_C,100);`
`Wait(800);`
`}`
`}`
	`#define NEAR 15 //cm`
	
	`task main(){`
	`SetSensorLowspeed(IN_4);`
	`while(true){`
	`OnFwd(OUT_AC,50);`
	`while(SensorUS(IN_4)>NEAR);`
	`Off(OUT_AC);`
	`OnRev(OUT_C,100);`
	`Wait(800);`
	`}`
	`}`

Sommario

In questo capitolo abbiamo capito come
- Lavorare con i sensori
di cui l'NXT è dotato. Oltre a ciò abbiamo imparato a conoscere i comandi
- until e
- while.
Vi consiglio di scrivere ora alcuni programmi con le vostre mani. Adesso avete sufficiente capacità per far svolgere al vostro robot anche compiti complessi.

I task e le subroutine

Finora abbiamo considerato dei programmi costituiti ciascuno da un **task**. Ma i programmi in NXC possono essere costituiti da molti **task**. E' anche possibile inserire dei pezzi di codice nelle cosiddette subroutines che possiamo poi chiamare da più parti, all' interno del programma principale. L'uso di molteplici **task** e di subroutines consente di rendere i programmi più leggibili e più compatti.
In questo capitolo comprenderemo come sfruttare le varie potenzialità potenzialità che ci offrono.

Task

Un programma NXC consiste di un numero di task a piacere, tra 1 e 255. Il **task** main deve sempre esistere,dato che è il principale **task** del programma, ed il primo ad essere eseguito. Gli altri **task** vengono eseguiti qualora vengano "chiamati" da un altro **task** in esecuzione, oppure viene programmata la loro esecuzione all' interno del **task** main.
Nell' esempio seguente il **task** main richiama altri due **task** secondari. Il compito del **task** main è solo quello di lanciare i **task** secondari, e quindi terminare.
Da quel momento, entrambi i **task** vengono eseguiti simultaneamente.
Col programma seguante passiamo a mostrare praticamente l'utilità dei **task**: vogliamo far muovere il robot seguendo un quadrato, come già abbiamo fatto in precedenza, ma contemporaneamente vogliamo che sia eseguito un suono (una specie di sirena d'ambulanza) in modo che il robot avverta del suo arrivo. Usiamo allora due **task**: uno gestisce i motori, e il secondo produce un suono multitonale per simulare una sirena.

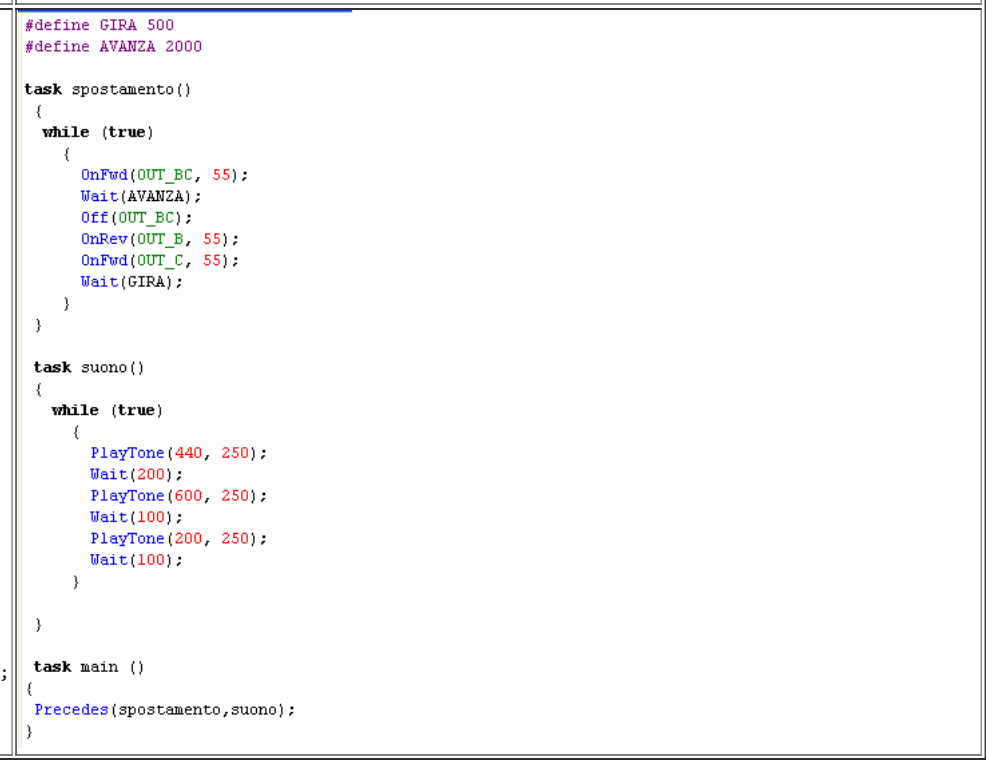
Ecco il programma:

Testo copiabile	Screenshot del BricxCC

```
#define GIRA 500
#define AVANZA 2000
task spostamento()
      {
    while (true)
          {
   OnFwd(OUT_BC, 55);
     Wait(AVANZA);
      Off(OUT_BC);
   OnRev(OUT_B, 55);
   OnFwd(OUT_C, 55);
      Wait(GIRA);
          }
        }

    task suono()
          {
     while (true)
          {
   PlayTone(440, 250);
        Wait(200);
   PlayTone(600, 250);
        Wait(100);
   PlayTone(200, 250);
        Wait(100);
          }
        }

    task main ()
          {
Precedes(spostamento,suono);
          }
```



Il **task** main fa partire entrambi gli altri task, dopodichè termina.
Il **task** spostamento fa muovere il robot lungo un quadrato.
Il **task** suouno produce una tonalità sonora multipla.

E' importante comprendere che i **task** avviati vengono eseguiti simultaneamente,cosa che in questo esempio non ha controindicazioni. Ma cosa accade se entrembi i **task** cercano di accedere ad un' unica risorsa?
Se non sappiamo gestire le richieste effettuate da parte di ogni **task**, la cosa può portare a dei risultati inattesi, ad esempio: se entrambi i **task** cercano di far muovere i motori simultaneamente, chi dei due o più **task** attivi ha la precedenza?. (ricordatevi che i **task** sono eseguiti simultaneamente)
Per ovviare a questo tipo di problema, è stato inventato un tipo particolare di variabili, le variabili **mutex** (mutual exclusion).
Possiamo assegnare queste variabili alle funzioni **Acquire** e **Release**, e racchiudere le parti critiche di codice tra di esse. In questo modo saremo certi che il controllo (ad esempio dei motori) verrà acquisito da un solo **task** per volta. Questo tipo di variabili (mutex) viene chiamato variabili-semaforo e la tecnica di programmazione che le utilizza viene detta programmazione concorrente.
Descriveremo più approfonditamente quest'argomento al capitolo relativo ai **Task paralleli**. Intanto ecco un esempio del loro utilizzo: un programma che usa due **task**: il primo task fa percorrere al robot un quadrato in un senso, l'altro gli fa percorrere il quadrato in snso opposto. Come nell' erampio precedente c'è bisogno di un terzo **task**, il **task main**, che solamente lancia gli altri due **task** e muore.

Ecco l'esempio:

Testo copiabile	Screenshot del BricxCC

```
#define MOVE 2000
 #define NOV 850
task quadr_orario()
      {
    while (true)
          {
       repeat(4)
            {
   OnFwd(OUT_BC, 75);
       Wait(MOVE);
    OnRev(OUT_B, 75);
       Wait(NOV);
      Off(OUT_BC);
          }
        }
      }
task quadr_antiorario()
      {
    while (true)
          {
       repeat(4)
            {
   OnFwd(OUT_BC, 75);
       Wait(MOVE);
    OnRev(OUT_C, 75);
       Wait(NOV);
      Off(OUT_BC);
          }
```

```
                    }
                  }
              task main()
                    {
Precedes(quadr_orario, quadr_antiorario);
                    }
```

```
#define MOVE 2000
#define NOV 850

task quadr_orario()
{
  while (true)
  {
    repeat(4)
    {
      OnFwd(OUT_BC, 75);
      Wait(MOVE);
      OnRev(OUT_B, 75);
      Wait(NOV);
      Off(OUT_BC);
    }
  }
}

task quadr_antiorario()
{
  while (true)
  {
    repeat(4)
    {
      OnFwd(OUT_BC, 75);
      Wait(MOVE);
      OnRev(OUT_C, 75);
      Wait(NOV);
      Off(OUT_BC);
    }
  }
}

task main()
  {
    Precedes(quadr_orario, quadr_antiorario);
  }
```

Bene, se provate ad eseguire l'esempio precedente, vi accorgerete che non funziona affatto:entrambi i **task** tentano di accedere alla stessa risorsa: i motori. E' palese che, essendo i **task** eseguiti in contemporanea, il motore non può dar retta ad entrambi. In assenza di un modo di gestirli , vige il caos, ed i comportamenti del nostro robot possono davvero essere imprevedibili.
Ecco qui sotto lo stesso programma scritto correttamente, utilizzando le variabili mutex:

Testo copiabile	Screenshot del BricxCC

```
          #define MOVE 2000
           #define NOV 850
             mutex motor;
         task quadr_orario()
                {
              while (true)
                    {
              Acquire(motor);
                 repeat(4)
                      {
              OnFwd(OUT_BC, 75);
                  Wait(MOVE);
               OnRev(OUT_B, 75);
                   Wait(NOV);
                  Off(OUT_BC);
                      }
               Release(motor);
                    }
                  }
      task quadr_antiorario()
                {
              while (true)
                    {
              Acquire(motor);
                 repeat(4)
                      {
              OnFwd(OUT_BC, 75);
                  Wait(MOVE);
               OnRev(OUT_C, 75);
                   Wait(NOV);
                  Off(OUT_BC);
                      }
              Release(motor);
                    }
                  }
             task main()
                    {
Precedes(quadr_orario, quadr_antiorario);
                    }
```

```
#define MOVE 2000
#define NOV 850
mutex motor;
task quadr_orario()
{
  while (true)
  {
    Acquire(motor);
    repeat(4)
    {
      OnFwd(OUT_BC, 75);
      Wait(MOVE);
      OnRev(OUT_B, 75);
      Wait(NOV);
      Off(OUT_BC);
    }
    Release(motor);
  }
}
task quadr_antiorario()
{
  while (true)
  {
    Acquire(motor);
    repeat(4)
    {
      OnFwd(OUT_BC, 75);
      Wait(MOVE);
      OnRev(OUT_C, 75);
      Wait(NOV);
      Off(OUT_BC);
    }
    Release(motor);
  }
}
task main()
  {
    Precedes(quadr_orario, quadr_antiorario);
  }
```

Ora sì, ci siamo: all' inizio dei **task** le variabili **mutex** acquisiscono **(acquire)** il controllo, e quando il **task** termina lo rilasciano **(release)**.
Adesso ogni **task** gestisce indipendentemente i motori fin quando gli serve,per poi restituirne il controllo al **task** concorrente.
Ora il conflitto è superato e la risorsa motori è gestita con corretezza. Infatti il programma funziona a meraviglia.

Subroutines

A volte capita di voler lanciare un pezzo di codice da svariati punti del programma. In questo caso possiamo racchiudere questo codice in una subroutine e dare ad essa un nome. A questo punto possiamo eseguire la subroutine semplicemente richiamandola con il suo nome all' interno di un task.

Ecco un esempio:

Testo copiabile	Screenshot del BricxCC

sub turn_around(int pwr)
{
OnRev(OUT_C, pwr); Wait(900);
OnFwd(OUT_AC, pwr);
}
task main()
{
OnFwd(OUT_AC, 75);
Wait(1000);
turn_around(75);
Wait(2000);
turn_around(75);
Wait(1000);
turn_around(75);
Off(OUT_AC);
}
	sub turn_around(int pwr)
	{
	OnRev(OUT_C, pwr); Wait(900);
	OnFwd(OUT_AC, pwr);
	}
	
	task main()
	{
	OnFwd(OUT_AC, 75);
	Wait(1000);
	turn_around(75);
	Wait(2000);
	turn_around(75);
	Wait(1000);
	turn_around(75);
	Off(OUT_AC);
	}

In questo programma abbiamo definito una subroutine che fa girare il robot attorno al proprio centro. Il **task** main chiama la subroutine tre volte. Notate che chiamiamo la subroutine col suo nome e le passiamo un parametro numerico inserendolo tra parentesi. Se chiamiamo una subroutine che non è stata creata per accettare parametri, basta far seguire il suo nome da due parentesi con niente tra esse. Vista così, la chiamata di una subroutine sembra proprio uno dei vari comandi che siamo soliti adoperare. Il vantaggio principale delle subroutine è che sono scritte una volta sola nella memoria dell' NXT e quindi non sprecano spazio.
Nel caso in cui, le subroutine siano corte, è conveniente utilizzare al loro posto le funzioni **inline**. Le funzioni inline non sono immagazzinate una sola volta nella memoria, ma cono copiate ogni volta nel posto in cui devono essere utilizzate. Questo ci fa sprecare un po' di memoria, ma non c'è limite al numero di funzioni **inline** che possiamo utilizzare.
Definire ed usare le funzioni **inline** è la stessa cosa che definire ed usare le subroutine, ecco qui un esempio di come vengono dichiarate:

```
inline int Name( Args ) {
   //body;
   return x*y;
}
```

Usare le funzioni **inline** è la stessa cosa che utilizzare le subroutine. Così l'esempio precedente, usando le funzioni **inline**,diventa:

Testo copiabile	Screenshot del BricxCC
inline void turn_around() { OnRev(OUT_C, 75); Wait(900); OnFwd(OUT_AC, 75);	

```
        }
    task main()
        {
OnFwd(OUT_AC, 75);
   Wait(1000);
 turn_around();
  Wait(2000);
turn_around();
  Wait(1000);
turn_around();
 Off(OUT_AC);
      }
```

```
inline void turn_around()
{
  OnRev(OUT_C, 75); Wait(900);
  OnFwd(OUT_AC, 75);
}

task main()
{
  OnFwd(OUT_AC, 75);
  Wait(1000);
  turn_around();
  Wait(2000);
  turn_around();
  Wait(1000);
  turn_around();
  Off(OUT_AC);
}
```

Prendendo spunto dall'esempio precedente, possiamo fare del tempo necessario per girarsi (turntime) l'argomento di una funzione, come esemplificato qui di seguito. Fate caso: tra le parentesi, che seguono il nome della funzione **inline**, andiamo a specificare gli argomenti della funzione. In questo caso indichiamo come argomento un intero (ci sono alcune altre possibilità), ed il suo nome è turntime. Se vi sono più argomenti, vanno separati con una virgola.

Testo copiabile	Screenshot del BricxCC

```
inline void turn_around(int pwr, int turntime)
                    {
                  OnRev(OUT_C, pwr);
                    Wait(turntime);
                  OnFwd(OUT_AC, pwr);
                    }
                task main()
                    {
                  OnFwd(OUT_AC, 75);
                      Wait(1000);
                turn_around(75, 2000);
                      Wait(2000);
                 turn_around(75, 500);
                      Wait(1000);
                turn_around(75, 3000);
                     Off(OUT_AC);
                         }
```

```
inline void turn_around(int pwr, int turntime)
{
  OnRev(OUT_C, pwr);
  Wait(turntime);
  OnFwd(OUT_AC, pwr);
}

task main()
{
  OnFwd(OUT_AC, 75);
  Wait(1000);
  turn_around(75, 2000);
  Wait(2000);
  turn_around(75, 500);
  Wait(1000);
  turn_around(75, 3000);
  Off(OUT_AC);
}
```

Notate che in NXC **sub** è lo stesso di **void**; inoltre le funzioni possono avere un tipo di ritorno differente da **void**, possono quindi restituire interi o stringhe al chiamante: per una spiegazione dettagliata di questo argomento fate riferimento alla guida di NXC.

Definizione di macro.

C'è un altro sistema per dare un nome a dei piccoli pezzi di codice: possiamo definire una macro in NXC (non confondete le macro di NXC con quelle di BricxCC). Abbiamo visto in precedenza come si definiscono le costanti, tramite #define, assegnando loro un nome. Ma desso abbiamo bisogno di definire non una singola costante ma un pezzo di codice. Qui di seguito c'è un esempio dello stesso programma di prima, che però usa una macro per far girare il robot.

Testo copiabile	Screenshot del BricxCC

```
          #define turn_around \
OnRev(OUT_B, 75); Wait(3400);OnFwd(OUT_AB, 75);

                task main()
                    {
                OnFwd(OUT_AB, 75);
                    Wait(1000);
                   turn_around;
                    Wait(2000);
                   turn_around;
                    Wait(1000);
```

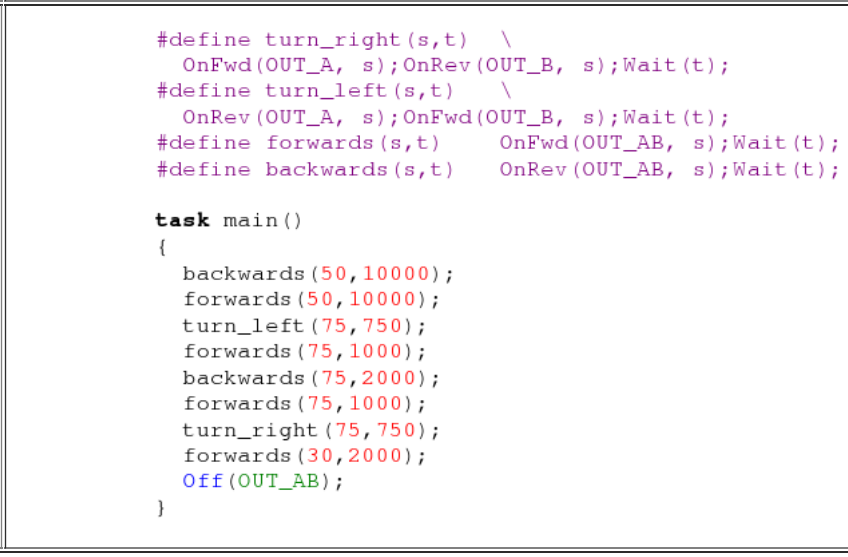
```
turn_around;
Off(OUT_AB);
    }
```

```
#define turn_around  \
  OnRev(OUT_B, 75); Wait(3400);OnFwd(OUT_AB, 75);

task main()
{
  OnFwd(OUT_AB, 75);
  Wait(1000);
  turn_around;
  Wait(2000);
  turn_around;
  Wait(1000);
  turn_around;
  Off(OUT_AB);
}
```

il termine turn_around, posto dopo #define, sostituisce il testo che lo segue nella definizione.
D'ora in poi, dovunque nel programma noi scriviamo turn_around, il termine verrà sosituito con il codice corrispondente. Da notare che il codice deve essere tutto contenuto in una riga. (Ci sono metodi per assegnare ad una macro del codice contenuto in più righe, ma non ve li raccomandiamo.)
Lo statement #define è molto potente. Può anche avere degli argomenti. Per avere un esempio pratico osservate il programma seguente.
In questo programma andiamo a definire quattro macro; una per andare avanti, una per andare indietro, una per andare a destra e l'ultima per andare a sinistra.
Ogni macro ha due parametri che le vengono assegnati: la velocità dei motori (s) ed il tempo per cui viene eseguita (t).

Testo copiabile	Screenshot del BricxCC

`#define turn_right(s,t) \`
`OnFwd(OUT_A, s);OnRev(OUT_B, s);Wait(t);`
`#define turn_left(s,t)`
`\`
`OnRev(OUT_A, s);OnFwd(OUT_B, s);Wait(t);`
`#define forwards(s,t)`
`OnFwd(OUT_AB, s);Wait(t);`
`#define backwards(s,t)`
`OnRev(OUT_AB, s);Wait(t);`
 	
`task main()`
`{`
`backwards(50,10000);`
`forwards(50,10000);`
`turn_left(75,750);`
`forwards(75,1000);`
`backwards(75,2000);`
`forwards(75,1000);`
`turn_right(75,750);`
`forwards(30,2000);`
`Off(OUT_AB);`
`}`
	

E' utile usare le macro. Rendono il codice più leggibile e compatto. Inoltre, potete facilmente modificare il codice, se ad esempio voleste cambiare le porte a cui i motori sono connessi.

Sommario

In questo capitolo avete imparato come usare
- I task,
- Le subroutines,
- Le funzioni inline,
- Le macro.

Le loro funzionalità ed i loro utilizzi sono differenti.

- **I task** di norma vengono eseguiti simultaneamente e si occupano di gestire ciascuno funzionalità differenti che devono essere eseguite contemporaneamente.
- **Le subroutines** sono utili quando , in vari punti dello stesso task, vogliamo eseguire dei corposi pezzi di codice.
  Utilizzare le subroutine ci permette di risparmiare memoria.
- **Le funzioni inline** sono utili quando un pezzo di codice deve essere utilizzato molte volte nello stesso programma (ma rispetto alle subroutine usano maggiore memoria).
- **Le macro**, infine, sono utili per sostituire piccoli pezzi di codice da utilizzare in differenti posizioni del programma.Possono avere anche dei parametri, il chè le rende più utili.

Ora che avete letto e compreso questo capitolo, avete tutte le conoscenze e le capacità per programmare il vostro robot anche per opeazioni complicate e difficili. I capitoli rimanenti del documento servono a capire come programmare il robot per applicazioni che sono importanti solo per alcuni utilizzatori con esigenze particolari.

Fare musica:

L' NXT ha un altoparlante integrato, che permette di riprodurre singoli suoni o file sonori. Questo serve in particolare quando volete che l' NXT vi avverta quando succede qualcosa. Ma può anche essere divertente avere un robot che suona, mentre va in giro :)

Riproduzione di file sonori

BrixCC implementa un convertitore da file **.wav** a file **.rso**, cui si può accedere tramite la voce di Menu **Tools - Sound conversion.**
Una volta convertito il file **.rso** può essere trasferito nella memoria flash dell' NXT mediante un' altra utility, l' **NXT memory browser** **Tools - NXT explorer** e fatto eseguire col comando **PlayFileEx(filename, volume, loop?)**

Gl argomenti del comando sono:

- filename = nome del file
- volume = un numero da zero a 4
- loop = se loop è uguale a 1 il file verrà riprodotto ininterrottamente, se vale 0 il file verrà riprodotto una sola volta.

Testo copiabile	Screenshot del BricxCC

```
#define TIME 200
#define MAXVOL 7
#define MINVOL 1
#define MIDVOL 3
#define pause_4th Wait(TIME)
#define pause_8th Wait(TIME/2)
#define note_4th \
PlayFileEx("! Click.rso",MIDVOL,FALSE); pause_4th
#define note_8th \
PlayFileEx("! Click.rso",MAXVOL,FALSE); pause_8th
task main()
{
PlayFileEx("! Startup.rso",MINVOL,FALSE);
Wait(2000);
note_4th;
note_8th;
note_8th;
note_4th;
note_4th;
pause_4th;
note_4th;
note_4th;
Wait(100);
}
```

Questo programma prima di tutto suona la musichetta di avvio dell' NXT che ormai vi sarà familiare, poi usa il suono standard "click" per suonare il motiveto “Shave and a haircut” che faceva impazzire Roger Rabbit. Le macro qui sono molto utili per semplificare il **task** main.
Provate a modificare il volume per....

Suonare musica

Per far eseguire un tono, potete utilizzare il comando **PlayToneEx(frequency, duration, volume, loop?)** che ha 4 argomenti. Il primo è la frequenza, in Hertz,il secondo la durata (in millesimi di secondo, come per il comando wait),il terzo e il quarto sono volume e loop, come in precedenza.
Potete benissimo usare anche **PlayTone(frequency, duration)**. In questo caso il volume è quello selezionato nel menu dell' NXT e il loop è disabilitato.

Qui di seguito una tabella contenente le frequenze più usate ed utili.

Testo copiabile

```
  Sound 3   4   5    6    7     8    9
    B     247 494 988  1976 3951 7902
    A#    233 466 932  1865 3729 7458
A     220 440 880  1760 3520 7040  14080
G#        415 831  1661 3322 6644  13288
G         392 784  1568 3136 6272  12544
F#        370 740  1480 2960 5920  11840
F         349 698  1397 2794 5588  11176
E         330 659  1319 2637 5274  10548
D#        311 622  1245 2489 4978  9956
D         294 587  1175 2349 4699  9398
C#        277 554  1109 2217 4435  8870
C         262 523  1047 2093 4186  8372
```

Screenshot del BricxCC

Sound	3	4	5	6	7	8	9
B	247	494	988	1976	3951	7902	
A#	233	466	932	1865	3729	7458	
A	220	440	880	1760	3520	7040	14080
G#		415	831	1661	3322	6644	13288
G		392	784	1568	3136	6272	12544
F#		370	740	1480	2960	5920	11840
F		349	698	1397	2794	5588	11176
E		330	659	1319	2637	5274	10548
D#		311	622	1245	2489	4978	9956
D		294	587	1175	2349	4699	9398
C#		277	554	1109	2217	4435	8870
C		262	523	1047	2093	4186	8372

Così come per **PlayFileEx**, l' NXT non aspetta la fine dell esecuzione di una nota.
Perciò, se utilizzate più di una nota in una riga aggiungete dei comandi wait in modo da fermare il programma per il tempo opportuno.

Qui di seguito ecco un esempio:

Testo copiabile

```
#define VOL 3
 task main()
       {
PlayToneEx(262,400,VOL,FALSE); Wait(500);
PlayToneEx(294,400,VOL,FALSE); Wait(500);
PlayToneEx(330,400,VOL,FALSE); Wait(500);
PlayToneEx(294,400,VOL,FALSE); Wait(500);
PlayToneEx(262,1600,VOL,FALSE); Wait(2000);
       }
```

Screenshot del BricxCC

```
#define VOL 3

task main()
{
  PlayToneEx(262,400,VOL,FALSE);  Wait(500);
  PlayToneEx(294,400,VOL,FALSE);  Wait(500);
  PlayToneEx(330,400,VOL,FALSE);  Wait(500);
  PlayToneEx(294,400,VOL,FALSE);  Wait(500);
  PlayToneEx(262,1600,VOL,FALSE); Wait(2000);

}
```

Potete creare pezzi musicali in maniera molto semplice utilizzando **Brick Piano**, che fa parte del BricxCC.
Se volete che l' NXT suoni musica mentre va in giro, usate un task apposito per la musica.

Qui di seguito c'è un esempio di un programma piuttosto stupido che fa andare avanti e indietro l' NXT mentre suona musica.

Testo copiabile

```
task music()
     {
  while (true)
       {
PlayTone(262,400); Wait(500);
PlayTone(294,400); Wait(500);
PlayTone(330,400); Wait(500);
PlayTone(294,400); Wait(500);
       }
     }
task movement()
```

Screenshot del BricxCC

```
        {
      while(true)
        {
OnFwd(OUT_AC, 75); Wait(3000);
OnRev(OUT_AC, 75); Wait(3000);
        }
      }
task main()
      {
Precedes(music, movement);
      }
```

```
task music()
{
  while (true)
  {
    PlayTone(262,400);  Wait(500);
    PlayTone(294,400);  Wait(500);
    PlayTone(330,400);  Wait(500);
    PlayTone(294,400);  Wait(500);
  }
}

task movement()
{
  while(true)
  {
    OnFwd(OUT_AC, 75); Wait(3000);
    OnRev(OUT_AC, 75); Wait(3000);
  }
}

task main()
{
  Precedes(music, movement);
}
```

Sommario

In questo capitolo abbimao imparato come
- Far suonare l'NXT.
Abbiamo pure visto come creare dei task separati per fare musica.

Gestione avanzata dei motori

Ci sono un certo numero di comandi addizionali per la gestione dei motori che possiamo utilizzare per gestirli meglio. In questo capitolo parleremo dei comandi: **ResetTachoCount**, **Coast (Float)**, **OnFwdReg**, **OnRevReg**, **OnFwdSync**, **OnRevSync**, **RotateMotor**, **RotateMotorEx**, e dei concetti basilari del **PID**.

Arresto non brusco

Se utilizzate il comando **Off()** il servomotore si arresta di colpo, arrestando e bloccando il proprio albero. E' possibile fermare il motore più dolcemente, senza usare il freno. Per ottenere ciò, usate i comandi **Float()** oppure **Coast()** (indifferentemente).
L'uso di questi comandi toglie semplicemente la corrente ai motori. L'esempio seguente ferma i motori in entrambi i modi: prima bruscamente , col comando **Off()**, poi gentilmente.
Fate caso: anche se per un robot così piccolo la differnza è piccola, c'è e si nota. Nel caso di robot di maggiori dimensioni,la diferenza aumenta di molto.

Testo copiabile	Screenshot del BricxCC

`task main()`
`{`
`OnFwd(OUT_AC, 75);`
`Wait(500);`
`Off(OUT_AC);`
`Wait(1000);`
`OnFwd(OUT_AC, 75);`
`Wait(500);`
`Float(OUT_AC);`
`}`
	`task main()`
	`{`
	`  OnFwd(OUT_AC, 75);`
	`  Wait(500);`
	`  Off(OUT_AC);`
	`  Wait(1000);`
	`  OnFwd(OUT_AC, 75);`
	`  Wait(500);`
	`  Float(OUT_AC);`
	`}`

Comandi avanzati

I comandi **OnFwd()** e **OnRev()** sono i comandi più semplici per muovere i motori. I servomotori dell' NXT contengono un encoder che permette di controllare con buona precisione la rotazione dell' albero e la velocità;
Il Firmware dell' NXT implementa un controller **PID (Proportional Integrative Derivative)** a ciclo chiuso per controllare i motori, utilizzando la lettura dei dati degli encoder. Se desiderate che il robot vada perfettamente diritto, potete utilzzare una sincronizzazione che fa muovere assieme una coppia di motori, e fa sì che uno dei due "aspetti" l'altro in caso uno dei due vanga rallentato oppure bloccato; in modo simile a questo, potete utilizzare una coppia di motori per far sì che il robot sterzi a destra, a sinistra, o routi sul posto, ma mantenendo i motori sincronizzati.
Ci sono un sacco di comandi per scatenare la potenza dei nostri servomotori! **OnFwdReg('ports','speed','regmode')** fa ruotare i motori specificati alla porta **(ports)** alla velocità **(speed)** applicando il metodo di sincronia **(regmode)** che può essere **OUT_REGMODE_IDLE**, **OUT_REGMODE_SPEED** or **OUT_REGMODE_SYNC**.
Se scegliamo **IDLE**, non verrà utilizzata la regolazione **PID**; se scegliamo **SPEED**, l' NXT manterrà costante la velocità del motore in ogni condizione (ad esempio se aumentiamo il carico del robot o la pendnza che esso supera), infine, se scegliamo **SYNC**, la coppia di motori selezionata tramite **ports** si muoverà in sincronia come spiegato in precedenza. **OnRevReg()** si comporta nello stesso modo, ma facendo girare i motori al contrario.

Testo copiabile	Screenshot del BricxCC

```
task main()
{
OnFwdReg(OUT_AC,50,OUT_REGMODE_IDLE);
Wait(2000);
Off(OUT_AC);
PlayTone(4000,50);
Wait(1000);
ResetTachoCount(OUT_AC);
OnFwdReg(OUT_AC,50,OUT_REGMODE_SPEED);
Wait(2000);
Off(OUT_AC);
PlayTone(4000,50);
Wait(1000);
OnFwdReg(OUT_AC,50,OUT_REGMODE_SYNC);
Wait(2000);
Off(OUT_AC);
}
```

```
task main()
{
  OnFwdReg(OUT_AC,50,OUT_REGMODE_IDLE);
  Wait(2000);
  Off(OUT_AC);
  PlayTone(4000,50);
  Wait(1000);
  ResetTachoCount(OUT_AC);
  OnFwdReg(OUT_AC,50,OUT_REGMODE_SPEED);
  Wait(2000);
  Off(OUT_AC);
  PlayTone(4000,50);
  Wait(1000);
  OnFwdReg(OUT_AC,50,OUT_REGMODE_SYNC);
  Wait(2000);
  Off(OUT_AC);
}
```

Questo programma mostra comportamenti differenti del robot in base alla modalità di gestione dei motori selezionata.
Se cercate di fermare le ruote tenendo il robot in mano:

- Il primo comportamento **(IDLE mode)**: fermando una ruota non accde assolutamente nulla.
- Nella seconda modalità **(SPEED mode)** cercando di rallentare una ruota il robot aumenta la potenza applicatale per neutralizzare l'interferenza esterna e mantenere la velocità invariata.
- Nell' ultima modalità **(SYNC mode)** arrestando una ruota si ferma anche l'altra, in attesa che quella bloccata possa ricominciare a girare.

OnFwdSync('ports','speed','turnpct') è equivalente al comando **OnFwdReg()** in modalità **SYNC**, ma consente di specificare "la percentuale di sterzata" tramite il parametro "turnpct" (da 0 a 100). **OnRevSync()** è lo stesso di prima, ma ovviamente fa andare i motori al contrario.

Il programma seguente esemplifica il funzionamento dei comandi ora menzionati: provate a cambiare la "percentuale di sterzata" ed osservate che cosa accade.

Testo copiabile	Screenshot del BricxCC

```
task main()
{
PlayTone(5000,30);
OnFwdSync(OUT_AC,50,0);
Wait(1000);
PlayTone(5000,30);
OnFwdSync(OUT_AC,50,20);
Wait(1000);
PlayTone(5000,30);
OnFwdSync(OUT_AC,50,-40);
Wait(1000);
PlayTone(5000,30);
OnRevSync(OUT_AC,50,90);
Wait(1000);
Off(OUT_AC);
}
```

```
task main()
{
  PlayTone(5000,30);
  OnFwdSync(OUT_AC,50,0);
  Wait(1000);
  PlayTone(5000,30);
  OnFwdSync(OUT_AC,50,20);
  Wait(1000);
  PlayTone(5000,30);
  OnFwdSync(OUT_AC,50,-40);
  Wait(1000);
  PlayTone(5000,30);
  OnRevSync(OUT_AC,50,90);
  Wait(1000);
  Off(OUT_AC);
}
```

Per finire, i motori possono essere fatti ruotare per un limitato numero di gradi (ricordate che un giro completo equivale a 360 gradi).
In tutti i comandi seguenti, potete agire sulla direzione del motore variando sia il segno della velocità che il segno dell' angolazione.
Così: se l' angolo e la velocità hanno lo stesso segno, il motore girerà in avanti, e se hanno invece segno opposto il motore girerà indietro.
RotateMotor('ports','speed','degrees') ruota il perno del motore specificato da **(ports)** di un angolo specificato da **(degrees)** alla velocità specificata da **(speed)** (da 0 a 100).

Testo copiabile	Screenshot del BricxCC

task main()
{
RotateMotor(OUT_AC, 50,360);
RotateMotor(OUT_C, 50,-360);
}
	task main()
	{
	RotateMotor(OUT_AC, 50,360);
	RotateMotor(OUT_C, 50,-360);
	}

RotateMotorEx('ports','speed','degrees','turnpct','sync', 'stop')è un' estensione del comando precedente, che permette di sincronizzare due motori (ad esempio OUT_AC) specificando una "percentuale di sterzata" **(turnpct)** (da 0 a 100) ed un flag booleano **(sync)** (che può essere vero o falso). Permete inoltre di specificere se il motore deve frenare il proprio asse di rotazione al termine della rotazione desiderata tramite il flag booleano **(stop)**.

Testo copiabile	Screenshot del BricxCC

task main()
{
RotateMotorEx(OUT_AC, 50, 360, 0, true, true);
RotateMotorEx(OUT_AC, 50, 360, 40, true, true);
RotateMotorEx(OUT_AC, 50, 360, -40, true, true);
RotateMotorEx(OUT_AC, 50, 360, 100, true, true);
}
	task main()
	{
	RotateMotorEx(OUT_AC, 50, 360, 0, true, true);
	RotateMotorEx(OUT_AC, 50, 360, 40, true, true);
	RotateMotorEx(OUT_AC, 50, 360, -40, true, true);
	RotateMotorEx(OUT_AC, 50, 360, 100, true, true);
	}

PID control

Il firmware NXT implementa un controller **PID (proportional integrative derivative)** per regolare la posizione e la velocità dei servomotori con precisione. Questo tipo di controller è uno dei più efficaci e più semplici controller a ciclo chiuso esistenti, e viene utilzzato molto spesso. In parole povere funziona così (verifica traduzione):
Il programma dà al controller un punto da raggiungere. Mette in moto i motori col comando **U(t)**, misurando la sua posizione **Y(t)** con l'encoder integrato e calcolando l'errore **e(t)=R8(t) -Y(t)**: in questo senso lo definiamo "closed loop controller" (controller a ciclo chiuso),poichè la posizione di output **Y(t)** viene riportata all' input del controller per fare il calcolo dell' errore.
Il controller trasforma l'errore **E(t)** nel comando **U(t)** in questo modo: **U(t) = P(t) + I(t) + D(t)**, dove **P(t) = KP·E(t)**, **I(t) = KI·(I(t–1) + E(t))** e **D(t) = KD·(E(t) – E(t –1))**.
Può sembrare difficile a prima vista, ma lasciate che vi spieghi il meccanismo. Il comando è la somma di tre contributi, la componente proporzionale **P(t)**, la componente integrata **I(t)** e la componente derivata **D(t)**.
P(t) rende il controller veloce nella risposta, ma non assicura un errore nullo all' eqilibrio;
I(t) dà "memoria" al controller, nel senso che tiene traccia degli errori accumulati e tende a compensarli, garantendo all' equilibrio un errore nullo.
D(t) dà "previsione futura" al controller (così come la derivata in matematica), aumentando la velocità di risposta.
So bene che il concetto può risultare ancora confuso, considerate che sull' argomento sono stati scritti libri accademici!
Ma possiamo sperimentare dal vivo, con il nostro NXT.

Il più semplice programma per fissare il concetto in memoria è il seguente:

Testo copiabile	Screenshot del BricxCC

#define P 50
#define I 50
#define D 50
task main()
{
RotateMotorPID(OUT_A, 100, 180, P, I, D);
Wait(3000);
}
	#define P 50
	#define I 50
	#define D 50
	
	task main(){
	RotateMotorPID(OUT_A, 100, 180, P, I, D);
	Wait(3000);
	}

Il comando **RotateMotorPID(port,speed, angle, Pgain,Igain,Dgain)** fa muovere i motori con dei settaggi del **PID** differenti da quelli di default.
Proviamo coi settaggi seguenti:

- (50,0,0): il motore non ruota esattamente di 180°, visto che rimane un errore non compensato.
- (0,x,x): senza la parte proporzionale , l' errore è enorme.
- (40,40,0): con questo settaggio l'albero del motore si muove troppo in avanti, e si deve correggere l'errore facendolo tornare indietro.
- (40,40,90):buona precisione ma maggiore lentezza.
- (40,40,200): il perno di rotazione del motore oscilla: la componente derivata è troppo elevata.

Provate altri settggi, per scoprire che effetto hanno sul funzionamento del motore.

Sommario

In questo capitolo abbiamo imparato
- I comandi avanzati per la gestione del motore:
Float(),**Coast()**che arrestano dolcemente il motore;
OnXxxReg(), and **OnXxxSync()** che permettono un controllo della velocità del motore e della sicronia;
RotateMotor() e **RotateMotorEx()** utilizzati per far ruotare l'asse del motore di un preciso numero di gradi.
Avete imparato anche qualcosa sui
- PID control; non è stata una spiegazione esaustiva, ma forse sono riuscito a destare in voi un po' di curiosità a riguardo. Date un' occhiata sul web, per approfondire la questione.

Gestione avanzata dei sensori

Nel capitolo 7 abbiamo analizzato la gestione di base dei sensori. Ma coi sensori si può fare ben di più.
In questo capitolo discuteremo della differenza tra **sensor mode** e **sensor type**, verdremo come utilizzare i vecchi (e compatibili) sensori RCX, utlilizzandoli in abbinamento ai cavi di conversione per l' NXT.

Sensor Type

Il comando **SetSensor()** che conosciamo già dal capitolo 7, fa due cose: seleziona il tipo di sensore, e seleziona il modo in cui il sensore opera.
Selezionando però separatamente il tipo di sensore ed il modo in cui esso opera,possiamo controllare il comportamento del sensore con maggiore precisione, il che, per particolari operazioni, è davvero importante.
Il tipo di sensore viene selezionato tramite il comando **SetSensorType()**.Ci sono un sacco di tipi di sensore, ma qui riportiamo i più comunemente usati:

- **SENSOR_TYPE_TOUCH**, è il sensore di tocco,
- **SENSOR_TYPE_LIGHT_ACTIVE**, sensore di luce (con led attivo),
- **SENSOR_TYPE_SOUND_DB**, sensore sonoro,
- **SENSOR_TYPE_LOWSPEED_9V**, sensore di ultrasuoni.

Selezionare il giusto tipo di sensore è importante per indicare se il sensore ha bisogno di una sorgente di alimentazione elettrica (ad esempio per accendere il led del sensore di luce), o per indicare all' NXT che il sensore è digitale e necessita di essere letto tramite il bus seriale I2C.
E' possibile usare i vecchi sensori dell' RCX connessi all' NXT:

- **SENSOR_TYPE_TEMPERATURE**, sensore di temperatura,
- **SENSOR_TYPE_LIGHT** sensore di luce,
- **SENSOR_TYPE_ROTATION** peril sensore di rotazione dell' RCX (di quest' ultimo sensore parleremo più avanti).

Sensor Mode

Il modo di utilizzo dei sensori è gestito dal comando **SetSensorMode()**. Ci sono 8 differenti modalità.
La più importante è

- **SENSOR_MODE_RAW**. In questa modalità il valore restituito dal sensore

sarà un numero compreso tra 0 e 1023. E' il valore "grezzo" restituito dal sensore. L'interpretazione del significato del valore "grezzo" restituito dipende dal tipo di sensore. Ad esempio: per il sensore di tocco, quando non c'è pressione alcuna il valore restituito è 1023. Quando il sensore è premuto del tutto il valore restituito è circa 50. Quando il sensore è parzialmente premuto, esso restiturà un numero compreso tra 1000 e 50. Quindi, se selezioniamo il modo **RAW** per un sensore di tocco, siamo adesso in grado di capire se il sensore è stato premuto parzialmente.
Se,invece, il sensore applicato è un sensore di luce, il valore restituito in modalità **RAW** varia da 300 (molta luce) ad 800 (buio). Questo consente una misura molto più precisa , rispetto all' uso del comando **SetSensor()**.
Per ulteriori dettagli fate riferimento alla guida alla programmazione dell' NXT.

La seconda modalità di utilizzo dei sensori è

- **SENSOR_MODE_BOOL**. In questa modalità il vaolre restituito dai sensori è 0 oppure 1.

Se il valore RAW fosse superiore a 562, il valore restituito in modalità BOOL è 0, altrimenti il valore è 1.

- **SENSOR_MODE_BOOL** è la modalità di utilizzo prestabilita per il sensore di tocco, ma può essere utilizzato anche per altri sensori,se non ci interesa prendere in considerazione i valori analogici restituiti ma se invece ci interessa il raggiungimento di un valore di soglia.

Le modalità

- **SENSOR_MODE_CELSIUS** e
- **SENSOR_MODE_FAHRENHEIT** sono utilizzate per il sensore di temperatura, solamente per indicare al sensore di fornire le temperature in gradi Celsius o Fahrenheit.
- **SENSOR_MODE_PERCENT** converte il valore RAW in un valore percentuale da 0 a 100.E' il valore di default per il sensore di luminosità.
- **SENSOR_MODE_ROTATION** è una modalità adoperata per leggere i sensori di rotazione (vedi sotto).

Altre due interessanti modalità:

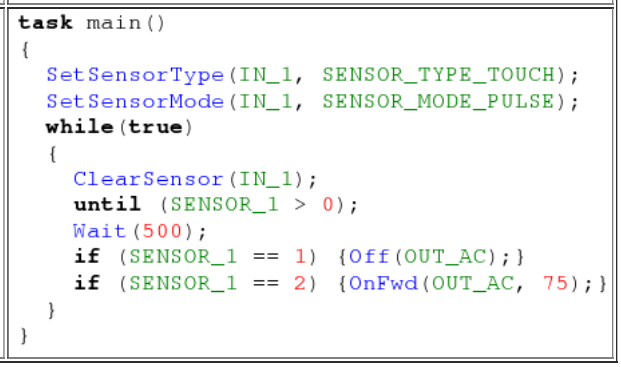
- **SENSOR_MODE_EDGE** e
- **SENSOR_MODE_PULSE**.

Queste modalità contano le transizioni, e cioè il cambiamento di stato di un sensore. Per esempio, se premiamo un sensore di tocco, il suo stato **RAW** passerà da alto a basso. Se lo rilasceremo, cambierà il suo output **RAW** in senso chiaramente opposto. Settando il sensore in modalità **SENSOR_MODE_PULSE**,verranno registrati solo i cambiamenti di fare da basso ad alto, per cui, ad esempio, ogni pressione e conseguente rilascio del sensore di tocco produrrà un cambiamento che verrà conteggiato una sola volta.
SENSOR_MODE_EDGE, viceversa, farà in modo che ogni variazione di stato venga conteggiata per cui, permendo e rilasciando il sensore di tocco avremo due cambiamenti entrambi registrati dal sensore. Sensori utilizzati in queste modalità possono ad esempio servire per capire quanto spesso viene premuto un sensore di tocco, oppure, tramite un sensore di luce, quando viene accesa e spenta una (intensa) lampada.
Ovviamente, quando conteggiate un numero di accensioni o spegnimenti, o un qualsiasi evento, dovete essere in grado di azzerare il contatore.
Per fare ciò esiste il comando **ClearSensor()**,che resetta a zero il valore del contatore del sensore specificato.

Vediamo ora un esempio. Il programma che segue usa un sensore per guidare il robot. Connettete il sensore di tocco tramite un lungo cavo alla porta di ingraesso n°1. Se toccate il sensore rapidamente per due volte consecutive il robot partirà in avanti, se lo toccherete una volta sola cesserà di muoversi. Notate: prima settiamo il tipo di sensore, e poi la modalità di utilizzo. Questo sembra essere essenziale, in quanto il cambio del tipo di sensore influenza anche le sue modalità di utilizzo.

Testo copiabile	Screenshot del BricxCC

```
task main()
{
SetSensorType(IN_1, SENSOR_TYPE_TOUCH);
SetSensorMode(IN_1, SENSOR_MODE_PULSE);
while(true)
{
ClearSensor(IN_1);
until (SENSOR_1 > 0);
Wait(500);
if (SENSOR_1 == 1) {Off(OUT_AC);}
if (SENSOR_1 == 2) {OnFwd(OUT_AC, 75);}
}
}
```



```
task main()
{
  SetSensorType(IN_1, SENSOR_TYPE_TOUCH);
  SetSensorMode(IN_1, SENSOR_MODE_PULSE);
  while(true)
  {
    ClearSensor(IN_1);
    until (SENSOR_1 > 0);
    Wait(500);
    if (SENSOR_1 == 1) {Off(OUT_AC);}
    if (SENSOR_1 == 2) {OnFwd(OUT_AC, 75);}
  }
}
```

Il sensore di rotazione

Il sensore di rotazione è un tipo di sensore davvero utile: è un encoder ottico, più o meno lo stesso che è contenuto nei servomotori dell' NXT.
Questo sensore ha un foro, nel quale si può infilare un asse che più ruotare liberamente,la cui posizione angolare relativa viene misurtata. Una rotazione completa dell' asse equivale a 16 conteggi, (o -16 se giriamo dalla parte opposta), il che significa che ad ogni passo corrisponde una rotazione di 22,5 gradi.
Molto poco rispetto alla precisione di un grado offerta dal servomotore.
Questo vecchio tipo di sensore può comunque risultare utile per contare la rotazione di un asse, senza sprecare un motore per eseguire questo compito. Considerate inoltre che utilizzare un motore come sensore di rotazione, utilizzando il suo encoder interno equivale a perdere un sacco di energia per far ruotare il tutto, mentre il vecchio sensore di rotazione è facilissimo da far ruotare.Se desiderate una precisione maggiore di un passo ogni 22,5 gradi potete sempre aumentare meccanicamente il numero di giri tramite un ingranaggio.
L'esempio seguente è preso dal vecchio tutorial dell' RCX.
Un' applicazione standard consiste nell avere due sensori di rotazione sulle due ruote di un robot, mosse ciascuna da un motore. Se volete far andare il robot diritto in avanti le due ruote si devono muovere alla stessa velocità.Sfortunatamente, i motori non sono uguali e quindi non si muovomno alla stessa velocità.Usando il sensore di rotazione possiamo vedere quale delle due ruote gira più velocemente
Possiamo a questo punto fermare (Meglio usando **Float()**) il motore più veloce, fin quando l'altro motore ha compiuto lo stesso numero di giri. Il programma seguente fa esattamente questo.Semplicemente, fa muovere il robot in linea retta. Per utilizzarlo, cambiata il vostro robot connettendo a ciascuna ruota un sensore di rotazione. Connetete i sensori alle porte 1 e 3.

Testo copiabile	Screenshot del BricxCC

```
task main()
{
SetSensor(IN_1, SENSOR_ROTATION); ClearSensor(IN_1);
SetSensor(IN_3, SENSOR_ROTATION); ClearSensor(IN_3);
while (true)
{
if (SENSOR_1 < SENSOR_3)
{OnFwd(OUT_A, 75); Float(OUT_C);}
else if (SENSOR_1 > SENSOR_3)
{OnFwd(OUT_C, 75); Float(OUT_A);}
```

```
        else
      {OnFwd(OUT_AC, 75);}
       }
     }
```

```
task main()
{
  SetSensor(IN_1, SENSOR_ROTATION); ClearSensor(IN_1);
  SetSensor(IN_3, SENSOR_ROTATION); ClearSensor(IN_3);
  while (true)
  {
    if (SENSOR_1 < SENSOR_3)
      {OnFwd(OUT_A, 75); Float(OUT_C);}
    else if (SENSOR_1 > SENSOR_3)
      {OnFwd(OUT_C, 75); Float(OUT_A);}
    else
      {OnFwd(OUT_AC, 75);}
  }
}
```

Il programma per prima cosa stabilisce che entrambi i sensori sono sensori di rotazione, e resetta il loro valore a zero.Poi inizia un ciclo infinito. All' interno del ciclo si controlla se la lettura di entrambi i sensori è uguale. Se la lettura è la stessa, il robot si muove in avanti in linea retta. Se invece uno dei due è maggiore, il motore corrispondente viene arrestato fin quando la lettura dei due sensori ritorna uno stasso valore. Ovviamente questo è un programma molto semplice; potete modificarlo per far percorrere al robot distanze molto precise, oppure per percorrere dei cerchi perfetti.

Connettere più sensori ad una porta di input

All 'inizio di questa sezione è necessaria una piccola avvertenza:data la nuova struttura dei sensori NXT, migliorati e dotati di connettore a 6 fili, non è più così facile connettere due sensori ad una sola porta com'era per i vecchi sensori dell' RCX.
Secondo me l'unica possibilità concreta (e facile da realizzare) è realizzare un multiplexer analogico da utilizzare in combinazione con un cavo convertitore. L'alternativa è un multiplexer digitale che può gestire la comunicazione tra i sensori e l' NXC tramite il bus I2C, ma certamente non è una soluzione alla portata dei principianti. Come sappiamo l' NXT ha 4 porte di ingresso per i sensori. Se vogliamo costruire robot complessi 4 ingressi potrebbero essere non sufficienti. Per fortuna, con qualche accorgimento, saremo in grado di connettere più sensori ad una singola porta di ingresso. La cosa più semplice è connettere due sensori di tocco alla stessa porta di ingresso. Se uno dei due è premuto (o entrambi), il valore letto sarà 1, altrimenti sarà 0. Non possiamo distinguere quale dei sensori è stato premuto ma spesso questo non è necessario.
Ad esempio, se mettiamo due sensori, uno davanti ed uno dietro al robot, sarà intuitivo comprendere quale dei due è premuto, in base alla direzione in cui sta andando il nostro robot. Se invece decidiamo di leggere i sensori in modalità RAW,disporremo di molte informazioni in più: possiamo probabilmente distinguere quale dei due sensori è stato premuto soltanto leggendo il valore che essi forniscono. Spesso, infatti, differenti sensori danno un differente risultato in modalità RAW anche se si trovano nella stessa condizione. Inoltre, qualora entrambi i sensori fossero premuti simultaneamente,restituiranno un valore molto più basso rispetto a quando uno solo viene premuto (30 contro 50 circa). Potete anche connettere un sensore di tocco ed un sensore di luce alla stessa porta di ingresso (solo sensori RCX). Settate il tipo di sensore come sensore di luce (altrimenti il sensore di luce non funzionerà). Settate la modalità di lettura RAW. Così facendo, se il sensore di tocco è premuto avrete un valore restituito minore di 100 (mentre se ricordate, il sensore di luce dà un valore in modalità RAW compreso tra 300 ed 800).

L'esempio che segue si basa su questa idea: il robot deve essere dotato di un sensore di luce rivolto verso il basso e di un sensore di tocco sul davanti.Connettete entrambi i sensori alla porta 1. Il robot si muoverà a caso sopra ad un' area chiara. Quendo il sensore di luce incontrerà una zona scura (valore RAW letto dal sensore >750) tornerà indietro un poco.Quando il sensore di tocco verrà premuto (valore RAW letto < 100), egualmente il robot indietreggerà.

Ecco il listato del programma.

Testo copiabile	Screenshot del BricxCC

```
mutex moveMutex;
  int ttt,tt2;
task moverandom()
        {
    while (true)
          {
ttt = Random(500) + 40;
    tt2 = Random();
  Acquire(moveMutex);
      if (tt2 > 0)
{ OnRev(OUT_A, 75); OnFwd(OUT_C, 75); Wait(ttt); }
      else
{ OnRev(OUT_C, 75); OnFwd(OUT_A, 75); Wait(ttt); }
     ttt = Random(1500) + 50;
   OnFwd(OUT_AC, 75); Wait(ttt);
     Release(moveMutex);
        }
       }
 task submain()
       {
  SetSensorType(IN_1, SENSOR_TYPE_LIGHT);
   SetSensorMode(IN_1, SENSOR_MODE_RAW);
        while (true)
          {
 if ((SENSOR_1 < 100) || (SENSOR_1 > 750))
            {
     Acquire(moveMutex);
  OnRev(OUT_AC, 75); Wait(300);
     Release(moveMutex);
          }
       }
     }
task main()
```

```
                {
Precedes(moverandom, submain);
                }
```

```
mutex moveMutex;
int ttt,tt2;

task moverandom()
{
  while (true)
  {
    ttt = Random(500) + 40;
    tt2 = Random();
    Acquire(moveMutex);
    if (tt2 > 0)
      { OnRev(OUT_A, 75); OnFwd(OUT_C, 75); Wait(ttt); }
    else
      { OnRev(OUT_C, 75); OnFwd(OUT_A, 75); Wait(ttt); }
    ttt = Random(1500) + 50;
    OnFwd(OUT_AC, 75); Wait(ttt);
    Release(moveMutex);
  }
}

task submain()
{
  SetSensorType(IN_1, SENSOR_TYPE_LIGHT);
  SetSensorMode(IN_1, SENSOR_MODE_RAW);
  while (true)
  {
    if ((SENSOR_1 < 100) || (SENSOR_1 > 750))
    {
      Acquire(moveMutex);
      OnRev(OUT_AC, 75); Wait(300);
      Release(moveMutex);
    }
  }
}

task main()
{
  Precedes(moverandom, submain);
}
```

Spero che il programma sia chiaro. Ci sono due **task**.Il **task** moverandom fa andare in giro il robot a caso. Il **task** principale per prima cosa lancia il **task** moverandom, setta il sensore, e poi aspetta che accada qualcosa.
Se la lettura del sensore cala troppo (il sensore di tocco è premuto) o cresce troppo (il sensore di luce è finito su una zona scura) il moto casuale cessa, il robot indietreggia un po', e poi riparte, ancora con moto casuale.

Sommario

In questo capitolo abbiamo visto una serie di funzionalità aggiuntive riguardanti i sensori. Abbiamo imparato ad impostare
- Il tipo (Sensor type) la
- Mdoalità di funzionamento (Sensor mode) di un sensore e come questo possa essere utilizzato per ottenere dai sensori più informazioni. Abbiamo imparato ad utilizzare i
- Sensori di rotazione. Infine, abbiamo imparato come connettere
- Più sensori alla stessa porta di input. Tutte queste potenzialità sono di grande interesse se si vuol costruire un robot complesso.In questo tipo di applicazioni i sensori giocano spesso un ruolo cruciale.

Task paralleli

Come già detto, I **task** in NXC sono eseguiti simultaneamente, o come si usa dire in parallelo. Questo è davvero interessante: mentre un task si occupa di controllare l'input dei sensori, un secondo può occuparsi di muovere il robot. Un altro ancora può sunare musica. I **task** paralleli sono molto utili, dunque, ma possono anche creare dei problemi; un **task** può interferire con un altro, qualora utilizzi la stessa risorsa di un altro.
Abbbiamo già affrontato questo argomento per sommi capi, mostrato un programma ove non ci sono risorse condivise da seguire (un **task** gestuva il suono e l'altro il movimento), e mostrato un programma ove la gestione dei motori da parte di due diversi **task** imponeva di usare cautela e di fare ricorso alle variabili **mutex**.

Un programma sbagliato

Prendiamo ad esempio il programma seguente: qui un task si occupa di far muovere il robot lungo un quadrato (come spesso abbiamo fatto negli esempi precedenti) mentre il secondo task controlla i sensori. Se un sensore di tocco viene premuto, il robot produce un suono, e fa un lento movimento all' indietro.

Testo copiabile	Screenshot del BricxCC

```
task check_sensors()
{
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      PlayTone(440,500);
      OnRev(OUT_BC, 45);
      Wait(5000);
    }
  }
}
task submain()
{
  while (true)
  {
    OnFwd(OUT_BC, 75); Wait(6000);
    OnRev(OUT_C, 75); Wait(1000);
  }
}
task main()
{
  SetSensor(IN_1,SENSOR_TOUCH);
  Precedes(check_sensors, submain);
}
```

A prima vista questo programma sembra perfetto, ma se lo eseguite otterrete un comportamento del robot davvero inaspettato.
Provate a premere il sensore di tocco: il robot farà un piccolo passo indietro, ma subito riprenderà ad avanzare.
La ragione di questo comportamento va ricercata nell' interferenza tra i due **task**.E' accaduto questo:mentre il **task** submain è in esecuzione ma in pausa (notate che questo task accende i motori con i comandi **OnFwd**, oppure **OnRev** e poi va in pausa **Wait**), viene premuto il sensore ed il controllo dei motori se lo pernde il task check_sensors, causando l'emissione di un tono acustico e poi l'indietreggiare lento del robot.
Ma il primo **task**,terminata l'istruzione di pausa, riprende il comando dei motori, muovendo il robot di nuovo avanti.
E' ovvio che non volevamo ottenere questo comportamento.
Il problema è che mentre eseguiva il secondo **task** il robot non si è accorto che il primo **task** era ancora in esecuzione, ma stava in attesa.

Regione critica, e variabili mutex

Un modo per risolvere questo problema è assicurarsi che in ogni momento solo un **task** possa muovere il robot. Vediamo di modificarel'esempio di prima, utilizzando le variabili **mutex**.

Testo copiabile	Screenshot del BricxCC

```
mutex motoremio;
task check_sensors()
{
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      Acquire(motoremio);
      PlayTone(440,500);
      OnRev(OUT_BC, 45);
      Wait(5000);
      Release(motoremio);
    }
  }
}
task submain()
{
  while (true)
  {
    OnFwd(OUT_BC, 75); Wait(6000);
    OnRev(OUT_C, 75); Wait(1000);
  }
}
task main()
{
  SetSensor(IN_1,SENSOR_TOUCH);
  Precedes(check_sensors, submain);
}
```

Ora il **task** check_sensor non molla più il controllo del motore , fin quando non ha terminato il suo compito. Ha acquisito (**Acquire**) la risorsa motore e fino a quando non la libera tramite **Release** , è il solo **task** a disporne.
Effettuato il **Release**, che libera la variabile mutex in modo che l'altro **task** possa usare la risorsa critica, il motore in questo caso. Il codice tra Acquire e Release è chiamato regione critica: critica significa che viene usata una risorsa condivisa.
Operando in questo modo i **task** non possono interferire tra loro.

Uso dei semafori

C'è un modo artigianale alternativo all' uso delle variabili mutex. Una tecnica standard per risolvere il problema è quello di usare una variabile per indicare quale dei **task** sta usando i motori.
Gli altri **task** non sono abilitati ad usare i motori finché il primo **task** indica, usando la variabile, che ha liberato le risorse. Una variabile di questo tipo è chiamata semaforo.
Facciamo un esempio con "sem" variabile semaforo (equiparabile ad una variabile mutex).
Poniamo che un valore uguale a 0 indica che nessun **task** sta usando i motori (la risorsa è libera).

Testo copiabile	Screenshot del BricxCC

`until (sem == 0);`
`sem = 1; //Acquire(sem);`
`// Do something with the motors`
`// critical region`
`sem = 0;`
`//Release(sem);`
	`until (sem == 0);`
	`sem = 1; //Acquire(sem);`
	
	`// Do something with the motors`
	`// critical region`
	
	`sem = 0; //Release(sem);`

Ora, qualunque **task** voglia usare i motori eseguirà i seguenti comandi:per prima cosa bisogna che nessuno utilizzi la risorsa, ciò significa che bisogna aspettare che il valore della variabile sia 0. Poi, ne prendiamo il controllo ponendo la variabile sem uguale ad 1.Ora possiamo controllare i motori. Alla fine,quando il **task** non ha più bisogno di utilizzare la risorsa, rimette la variabile sem al valore di zero, segnalando così che la risorsa è stata liberata alle altre sezioni del programma.

Qui c'è lo stesso programma di prima, modificato utilizzando una variabile semaforo. Quando il sensore di tocco sente qualcosa, la variabile semaforo è impostata al valore 1 e il **task** submain prende il controllo dei motori. Durante questo lasso di tempo il **task** move_square deve stare in attesa. Quando submain ha finito, la variabile semaforo è impostata a zero e move_square può continuare ad usare le risorse condivise.

Testo copiabile	Screenshot del BricxCC

`int sem;`
`task move_square()`
`{`
`while (true)`
`{`
`until (sem == 0); sem = 1;`
`OnFwd(OUT_AC, 75);`
`sem = 0;`
`Wait(1000);`
`until (sem == 0); sem = 1;`
`OnRev(OUT_C, 75);`
`sem = 0;`
`Wait(850);`
`}`
`}`
`task submain()`
`{`
`SetSensor(IN_1, SENSOR_TOUCH);`
`while (true)`
`{`
`if (SENSOR_1 == 1)`
`{`
`until (sem == 0); sem = 1;`
`OnRev(OUT_AC, 75); Wait(500);`
`OnFwd(OUT_A, 75); Wait(850);`
`sem = 0;`
`}`
`}`
`}`
`task main()`
`{`
`sem = 0;`
`Precedes(move_square, submain);`
`}`	

```
int sem;

task move_square()
{
  while (true)
  {
    until (sem == 0); sem = 1;
    OnFwd(OUT_AC, 75);
    sem = 0;
    Wait(1000);
    until (sem == 0); sem = 1;
    OnRev(OUT_C, 75);
    sem = 0;
    Wait(850);
  }
}

task submain()
{
  SetSensor(IN_1, SENSOR_TOUCH);
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      until (sem == 0); sem = 1;
      OnRev(OUT_AC, 75); Wait(500);
      OnFwd(OUT_A, 75); Wait(850);
      sem = 0;
    }
  }
}

task main()
{
  sem = 0;
  Precedes(move_square, submain);
}
```

Penserete che non è necessario, in move_square, impostare ad 1 la variabile semaforo e poi nuovamente a 0.
E' comunque utile. La ragione è che **OnFwd()** è un doppio comando (vedi Capitolo 10). Non vogliamo che questa sequenza di comandi venga interrotta da un altro **task.**
Le variabili semaforo sono molto utili, e se scrivete programmi complicati, con molti task , sono quasi sempre necessarie.
(C'è comunque una piccola possibilità che non funzionino, cercate di immaginare perché).

Sommario

In questo capitolo abbiamo studiato alcuni dei problemi che vengono posti quando si usano **task** differenti.
Siate sempre attenti a questi possibili malfunzionamenti: molti comportamenti inaspettati ne possono derivare.
La prima soluzione che abbiamo usato ferma e fa ripartire i **task** con
- Acquire e
- Release
in modo da assicurarsi che solo un task sia girando in un dato momento. Il secondo tipo di approccio prevede l'uso delle
- Variabili semaforo per controllare l'esecuzione dei **task**.
Ciò garantisce che in ogni momento venga eseguita la parte critica di un solo **task**.

Comunicazione tra i robot

(VERIFICARE) Se possedete più di un robot NXT questo capitolo è per voi. (Comunque anche se avete un solo NXT potete comunicare con il computer)
I robot possono comunicare tra di loro tramite la tecnologia radio Bluetooth: potete avere molti robot che collaborano (o lottano tra loro), e potete costruire un grande e complesso robot utilizzando due NXT in modo da poter usare 6 motori e 8 sensori.
Per i vecchi RCX la questione è semplice: il robot manda in giro un segnale infrarosso e tutti i robot attorno lo ricevono.
Per L' NXT la faccenda cambia del tutto.
Primo, dovete connettere due o più NXT (o un NXT al PC) tramite il menu onbrick Bluetooth,(verifica traduzione) solo in seguito potete mandare dei messaggi ai dispositivi connessi.

L'NXT che inizia la comunicazione è detto master, e può avere fino a 3 slaves connessi alle linee 1,2,3; Gli Slaves vedono sempre il master connesso alla linea 0. Potete spedire messaggi verso 10 caselle di posta disponibili.

Messaggi Master-Slave

Vedremo due programmi, uno per il master, uno per lo slave. Questi programmi basilari vi insegneranno come un rapido e continuo invio di stringhe può essere utilizzato da un network wireless di due NXT.
Per prima cosa il programma master controlla che sia uno slave sia online sulla linea 1(costante BT_CONN)utilizzando al funzione BluetoohtStatus(conn), quindi crea e spedisce messaggi con il prefisso M ed un numero crescente con **SendRemoteString(conn,queue,string)**, mentre riceve messaggi dallo slave con **ReceiveRemoteString(queue,clear,string)** e mostra i dati.

Testo copiabile	Screenshot del BricxCC

```
//MASTER
#define BT_CONN 1
#define INBOX 1
#define OUTBOX 5
sub BTCheck(int conn)
{
if (!BluetoothStatus(conn)==NO_ERR)
{
TextOut(5,LCD_LINE2,"Error");
Wait(1000);
Stop(true);
}
}
task main()
{
string in, out, iStr;
int i = 0;
BTCheck(BT_CONN); //check slave connection
while(true)
{
iStr = NumToStr(i);
out = StrCat("M",iStr);
TextOut(10,LCD_LINE1,"Master Test");
TextOut(0,LCD_LINE2,"IN:");
TextOut(0,LCD_LINE4,"OUT:");
ReceiveRemoteString(INBOX, true, in);
SendRemoteString(BT_CONN,OUTBOX,out);
TextOut(10,LCD_LINE3,in);
TextOut(10,LCD_LINE5,out);
Wait(100);
i++;
}
}
```

```
//MASTER
#define BT_CONN 1
#define INBOX 1
#define OUTBOX 5

sub BTCheck(int conn){
  if (!BluetoothStatus(conn)==NO_ERR){
    TextOut(5,LCD_LINE2,"Error");
    Wait(1000);
    Stop(true);
  }
}

task main(){
  string in, out, iStr;
  int i = 0;
  BTCheck(BT_CONN); //check slave connection
  while(true){
    iStr = NumToStr(i);
    out = StrCat("M",iStr);
    TextOut(10,LCD_LINE1,"Master Test");
    TextOut(0,LCD_LINE2,"IN:");
    TextOut(0,LCD_LINE4,"OUT:");
    ReceiveRemoteString(INBOX, true, in);
    SendRemoteString(BT_CONN,OUTBOX,out);
    TextOut(10,LCD_LINE3,in);
    TextOut(10,LCD_LINE5,out);
    Wait(100);
    i++;
  }
}
```

Il programma slave è molto simile ma utilizza **SendResponseString(queue,string)** invece di **SendRemoteString** in quanto lo slave scambia dati solo col master, che è sempre sulla linea 0.

Testo copiabile	Screenshot del BricxCC

```
//SLAVE
#define BT_CONN 1
#define INBOX 5
#define OUTBOX 1
sub BTCheck(int conn)
{
if (!BluetoothStatus(conn)==NO_ERR)
{
TextOut(5,LCD_LINE2,"Error");
Wait(1000);
Stop(true);
}
}
task main()
{
string in, out, iStr;
int i = 0;
BTCheck(0); //check master connection
while(true)
{
iStr = NumToStr(i);
out = StrCat("S",iStr);
TextOut(10,LCD_LINE1,"Slave Test");
TextOut(0,LCD_LINE2,"IN:");
```

```
      TextOut(0,LCD_LINE4,"OUT:");
  ReceiveRemoteString(INBOX, true, in);
     SendResponseString(OUTBOX,out);
       TextOut(10,LCD_LINE3,in);
      TextOut(10,LCD_LINE5,out);
             Wait(100);
                i++;
                 }
              }
```

```
//SLAVE
#define BT_CONN 1
#define INBOX 5
#define OUTBOX 1

sub BTCheck(int conn){
   if (!BluetoothStatus(conn)==NO_ERR){
      TextOut(5,LCD_LINE2,"Error");
      Wait(1000);
      Stop(true);
   }
}

task main(){
   string in, out, iStr;
   int i = 0;
   BTCheck(0); //check master connection
   while(true){
     iStr = NumToStr(i);
     out = StrCat("S",iStr);
     TextOut(10,LCD_LINE1,"Slave Test");
     TextOut(0,LCD_LINE2,"IN:");
     TextOut(0,LCD_LINE4,"OUT:");
     ReceiveRemoteString(INBOX, true, in);
     SendResponseString(OUTBOX,out);
     TextOut(10,LCD_LINE3,in);
     TextOut(10,LCD_LINE5,out);
     Wait(100);
     i++;
   }
}
```

Se uno dei programmi cessa di girare, l' altro continuerà a spedire dati in sua direzione con numeri via via maggiori, senza sapere che ogni messaggio viene perso, poichè nessuno ascolta più.
Per ovviare all' inconveniente possiamo creare un protocollo migliore, con notifica di ricezione.

Spedizione con ricevuta di ricezione.

In questa sezione troviamo altri due programmi: questa volta il master spedisce i dati con **SendRemoteNumber(conn,queue,number)** e rimane in attesa di una risposta dallo slave (ciclo **until**, in cui c'è un' istruzione **ReceiveRemoteString**); solo se lo slave è in ascolto e spedisce notifica di ricezione, il master continuerà la sua trasmissione. Lo slave semplicemente ascolta tramite **ReceiveRemoteNumber(queue,clear,number)** e spedisce notifiche di ricezione tramite **SendResponseNumber**.

Bisogna comunicare alla coppia di programmi master-slave quale deve essere la notifica di ricezione: in questo caso ho scelto il carattere esadecimale **0xFF**
Il master manda numeri a caso, ed attende risposta dallo slave. Ogni volta che riceve una notifica di ricezione deve svuotare la variabile in cui la immagazzina, altrimenti il master potrebbe ingannarsi, vedendo la variabile contenere il giusto valore, e continare a spedire numeri anche in assenza di rapporti di ricezione.
Lo slave controlla continuamente la casella di posta. Se questa non è vuota visualizza il valore letto e spedisce un rapporto di avvenuta ricezione al master.
All' inizio del programma, ho scelto di far inviare un messaggio di avvenuta ricezione anche senza aver ricevuto nessun dato, per sbloccare la trasmissione da parte del master.
Senza questo stratagemma, se il programma del master parte per primo, rimarrà inattivo, anche se il programma slave verrà fatto partire.Con lo stratagemma, invece, i primi messaggi andranno persi, ma potremo far partire il programma master ed il programma slave in tempi diversi, senza rischio che si blocchino.

Testo copiabile	Screenshot del BricxCC

```
                //MASTER
              #define BT_CONN 1
               #define OUTBOX 5
               #define INBOX 1
 #define CLEARLINE(L)                \
TextOut(0,L,"                         ");


        sub BTCheck(int conn)
                {
   if (!BluetoothStatus(conn)==NO_ERR)
                {
     TextOut(5,LCD_LINE2,"Error");
              Wait(1000);
              Stop(true);
                  }
                }
            task main()
                  {
                int ack;
                 int i;
            BTCheck(BT_CONN);
  TextOut(10,LCD_LINE1,"Master sending");
```

```
                while(true)
                    {
                  i = Random(512);
                CLEARLINE(LCD_LINE3);
               NumOut(5,LCD_LINE3,i);
                      ack = 0;
          SendRemoteNumber(BT_CONN,OUTBOX,i);
                  until(ack==0xFF)
                        {
until(ReceiveRemoteNumber(INBOX,true,ack) == NO_ERR);
                        }
                   Wait(250);
                    }
                  }
```

```
//MASTER
#define BT_CONN 1
#define OUTBOX 5
#define INBOX 1
#define CLEARLINE(L)  \
  TextOut(0,L,"                    ");

sub BTCheck(int conn){
   if (!BluetoothStatus(conn)==NO_ERR){
      TextOut(5,LCD_LINE2,"Error");
      Wait(1000);
      Stop(true);
   }
}

task main(){
   int ack;
   int i;
   BTCheck(BT_CONN);
   TextOut(10,LCD_LINE1,"Master sending");
   while(true){
     i = Random(512);
     CLEARLINE(LCD_LINE3);
     NumOut(5,LCD_LINE3,i);
     ack = 0;
     SendRemoteNumber(BT_CONN,OUTBOX,i);
     until(ack==0xFF) {
       until(ReceiveRemoteNumber(INBOX,true,ack) == NO_ERR
     }
     Wait(250);
   }
}
```

Testo copiabile	Screenshot del BricxCC

```
                  //SLAVE
               #define BT_CONN 1
              #define OUT_MBOX 1
               #define IN_MBOX 5

             sub BTCheck(int conn)
                    {
          if (!BluetoothStatus(conn)==NO_ERR)
                    {
               TextOut(5,LCD_LINE2,"Error");
                 Wait(1000);
                 Stop(true);
                   }
                  }

              task main()
                   {
                  int in;
                BTCheck(0);
       TextOut(5,LCD_LINE1,"Slave receiving");
   SendResponseNumber(OUT_MBOX,0xFF); //unblock master
                  while(true)
                    {
if (ReceiveRemoteNumber(IN_MBOX,true,in) != STAT_MSG_EMPTY_MAILBOX)
                    {
                 TextOut(0,LCD_LINE3,"
                        ");
                NumOut(5,LCD_LINE3,in);
             SendResponseNumber(OUT_MBOX,0xFF);
                    }
           Wait(10); //take breath (optional)
                   }
                  }
```

```
//SLAVE
#define BT_CONN 1
#define OUT_MBOX 1
#define IN_MBOX 5

sub BTCheck(int conn){
   if (!BluetoothStatus(conn)==NO_ERR){
      TextOut(5,LCD_LINE2,"Error");
      Wait(1000);
      Stop(true);
   }
}

task main(){
   int in;
   BTCheck(0);
   TextOut(5,LCD_LINE1,"Slave receiving");
   SendResponseNumber(OUT_MBOX,0xFF); //unblock ma
   while(true){
     if (ReceiveRemoteNumber(IN_MBOX,true,in) != S
       TextOut(0,LCD_LINE3,"                    ");
       NumOut(5,LCD_LINE3,in);
       SendResponseNumber(OUT_MBOX,0xFF);
     }
     Wait(10); //take breath (optional)
   }
}
```

Controllo diretto

C'è un' altra interessante funzione della comunicazione Bluetooth: il master può dare ordini diretti agli slave. Nell'esempio che segue, il master manda i comandi agli slave per muovere i motori o per riprodurre dei suoni.
Per far ciò, non è necessario utilizzare un programma slave, dato che il firmware dell'NXT si occupa di gestire direttamente gli ordini del master.

Testo copiabile	Screenshot del BricxCC

```
//MASTER
#define BT_CONN 1
#define MOTOR(p,s) RemoteSetOutputState(BT_CONN, p, s, \
  OUT_MODE_MOTORON+OUT_MODE_BRAKE+OUT_MODE_REGULATED, \
    OUT_REGMODE_SPEED, 0, OUT_RUNSTATE_RUNNING, 0)

sub BTCheck(int conn)
{
 if (!BluetoothStatus(conn)==NO_ERR)
 {
  TextOut(5,LCD_LINE2,"Error");
  Wait(1000);
  Stop(true);
 }
}

task main()
{
 BTCheck(BT_CONN);
 RemotePlayTone(BT_CONN, 4000, 100);
 until(BluetoothStatus(BT_CONN)==NO_ERR);
 Wait(110);
 RemotePlaySoundFile(BT_CONN, "! Click.rso", false);
 until(BluetoothStatus(BT_CONN)==NO_ERR);
 //Wait(500);
 RemoteResetMotorPosition(BT_CONN,OUT_A,true);
 until(BluetoothStatus(BT_CONN)==NO_ERR);
 MOTOR(OUT_A,100);
 Wait(1000);
 MOTOR(OUT_A,0);
}
```

```
//MASTER
#define BT_CONN 1
#define MOTOR(p,s) RemoteSetOutputState(BT_CO
  OUT_MODE_MOTORON+OUT_MODE_BRAKE+OUT_MODE_RE
  OUT_REGMODE_SPEED, 0, OUT_RUNSTATE_RUNNING,

sub BTCheck(int conn){
   if (!BluetoothStatus(conn)==NO_ERR){
      TextOut(5,LCD_LINE2,"Error");
      Wait(1000);
      Stop(true);
   }
}

task main(){
   BTCheck(BT_CONN);
   RemotePlayTone(BT_CONN, 4000, 100);
   until(BluetoothStatus(BT_CONN)==NO_ERR);
   Wait(110);
   RemotePlaySoundFile(BT_CONN, "! Click.rso"
   until(BluetoothStatus(BT_CONN)==NO_ERR);
   //Wait(500);
   RemoteResetMotorPosition(BT_CONN,OUT_A,tru
   until(BluetoothStatus(BT_CONN)==NO_ERR);
   MOTOR(OUT_A,100);
   Wait(1000);
   MOTOR(OUT_A,0);
}
```

Sommario

In questo capitolo abbiamo imparato alcuni aspetti basilari della
- Comunicazione Bluetooth
tra robot: connettere due NXT, spedire e ricevere stringhe, numeri, e utilizzare le
- Notifiche di ricezione. Quest' ultimo aspetto è molto importante quando è necessario un protocollo di comunicazione sicura tra i robot. L'ultima funzione appresa è quella di
- Spedire direttamente dei comandi dal robot master allo slave.

Ulteriori comandi

NXC ha un certo numero di comandi addizionali. In questo capitolo ne vedremo di tre tipi:

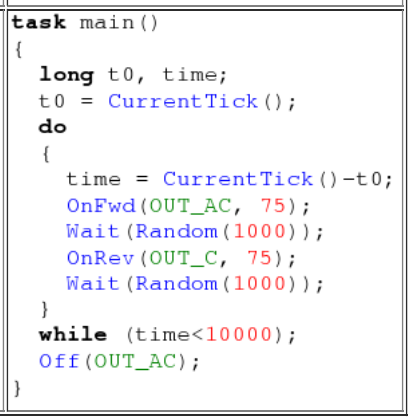
- Uso del timer
- Comandi per controllare il display,e
- Comandi per usare il filesystem di NXT.

Timer

L'NXT ha un timer che gira continuamente.Il timer conta unità (tick) di 1/1000 di secondo. Potete ottenere il numero corrente di tick con l'istruzione CurrentTick().

Testo copiabile	Screenshot del BricxCC

```
task main()
{
long t0, time;
t0 = CurrentTick();
do'
{
time = CurrentTick()-t0;
OnFwd(OUT_AC, 75);
Wait(Random(1000));
OnRev(OUT_C, 75);
Wait(Random(1000));
}
while (time<10000);
Off(OUT_AC);
}
```

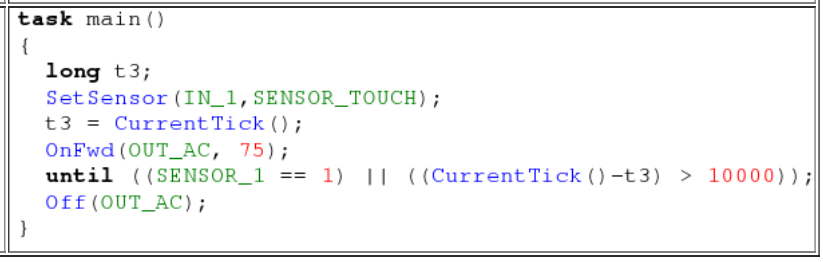


Qui c'è un esempio dell' utilizzo del timer: il robot si muve di moto casuale per dieci secondi. Paragoniamo questo programma con quello visto nel capitolo 6, che faceva esattamente la stessa cosa. Il programma che usa il timer è decisamente più semplice. Il timer è molto utile come alternativa all' istruzione Wait(). Potete arrestare l'esecuzione per un determinato periodo di tempo resettando un timer, e poi aspettare che raggiunga un determinato valore. Ma, durante l'attesa,il nostro robot può reagire ad un particolare evento (per esempio un segnale proveniente dai sensori).

Il programma che segue è un esempio di questo comportamento.Fa andare il robot per dieci secondi, oppure finchè il sensore di tocco ha sentito una pressione.

Testo copiabile	Screenshot del BricxCC

```
task main()
{
long t3;
SetSensor(IN_1,SENSOR_TOUCH);
t3 = CurrentTick();
OnFwd(OUT_AC, 75);
until ((SENSOR_1 == 1) || ((CurrentTick()-t3) > 10000));
Off(OUT_AC);
}
```



Non dimenticate: il timer lavora con passi di un millesimo di secondo, esattamente come il comando **Wait()**

Display a matrice di pixel

L'NXT ha un display monocromatico a matrice di pixel con una risoluzione di 100 x 64. Ci sono parecchie funzioni API per disegnare sullo schermo

- Stringhe di testo,
- Numeri,
- Punti,
- Linee,
- Rettangoli,
- Cerchi, ed anche
- Immagini bitmap (files con estensione .ric).

L'esempio che segue tenterà di esemplificare tutto ciò. Il pixel di coordinate (0,0) è in fondo a sinistra dello schermo.

Testo copiabile	Screenshot del BricxCC

```
#define X_MAX 99
#define Y_MAX 63
#define X_MID (X_MAX+1)/2
#define Y_MID (Y_MAX+1)/2
task main()
{
int i = 1234;
TextOut(15,LCD_LINE1,"Display", true);
NumOut(60,LCD_LINE1, i);
PointOut(1,Y_MAX-1);
```

```
              PointOut(X_MAX-1,Y_MAX-1);
                   PointOut(1,1);
                PointOut(X_MAX-1,1);
                     Wait(200);
                RectOut(5,5,90,50);
                     Wait(200);
                LineOut(5,5,95,55);
                     Wait(200);
                LineOut(5,55,95,5);
                     Wait(200);
           CircleOut(X_MID,Y_MID-2,20);
                     Wait(800);
                   ClearScreen();
GraphicOut(30,10,"faceclosed.ric"); Wait(500);
                   ClearScreen();
       GraphicOut(30,10,"faceopen.ric");
                     Wait(1000);
                         }
```

```
#define X_MAX 99
#define Y_MAX 63
#define X_MID (X_MAX+1)/2
#define Y_MID (Y_MAX+1)/2

task main(){
   int i = 1234;
   TextOut(15,LCD_LINE1,"Display", true);
   NumOut(60,LCD_LINE1, i);
   PointOut(1,Y_MAX-1);
   PointOut(X_MAX-1,Y_MAX-1);
   PointOut(1,1);
   PointOut(X_MAX-1,1);
   Wait(200);
   RectOut(5,5,90,50);
   Wait(200);
   LineOut(5,5,95,55);
   Wait(200);
   LineOut(5,55,95,5);
   Wait(200);
   CircleOut(X_MID,Y_MID-2,20);
   Wait(800);
   ClearScreen();
   GraphicOut(30,10,"faceclosed.ric");    Wait(500);
   ClearScreen();
   GraphicOut(30,10,"faceopen.ric");
   Wait(1000);
}
```

Tutte le funzioni utilizzate sono autodescrittive, ma vediamo di descriverne i parametri in dettaglio.

- **ClearScreen()** cancella lo schermo;
- **NumOut(x, y, number)** stampa un numero alle coordinate specificate;
- **TextOut(x, y, string)** stampa un testo alle coordinate specificate;
- **GraphicOut(x, y, filename)** mostra un file di bitmap (.ric)
- **CircleOut(x, y, radius)** disegna un cerchio con centro corrispondente alle coordinate specificate,e con un dato raggio;
- **LineOut(x1, y1, x2, y2)** disegna una linea dal punto (x1,x2) al punto (x2,y2)
- **PointOut(x, y)** accende il pixel di coordinate (x,y)
- **RectOut(x, y, width, height)** disegna un rettangolo con il vertice in basso a sinistra alle coordinate (x,y) e con le dimensioni specificate;
- **ResetScreen()** resetta lo schermo.

File system

L'NXT può leggere e scrivere dei files, memorizzandoli sulla sua memoria flash. Così,ad esempio, possimo salvare una sequenza di valori letti da un sensore, o leggere dei numeri da un file durante l'esecuzione di un programma. L' unico limite al numero ed alle dimensioni dei files è l'estensione della memoria flash.
Le funzioni API dell' NXT consentono di gestire i files (creare, rinominare, cancellare, trovare), permettono di leggere e scrivere stringhe di testo numeri, e singoli bytes.
Nell' esempio che segue, impareremo a creare un file, scriverci dentro delle stringhe di testo, e rinominarlo.Prima di tutto, il programma cancella gli eventuali files che abbiano il nome uguale a quello che vogliamo creare. Nonè una buona abitudine (dovremmo controllare se ne esiste uno e cancellarlo manualmente, oppure scegliere un altro nome per il nostro file) ma in un caso semplice come il nostro non ci sono problemi.
Poi creiamo il file **Createfile("Danny.txt",512,fileHandle**), specificando quindi nome, grandezza, e handle, dove il firmware dell' NXT scriverà il mumero che gli serve per suo uso.
Ancora, crea le stringhe e le scrive sul file separandole con un invio tramite lo statement **WriteLnString(fileHandle,string,bytesWritten)** ove ogni parametro deve essere una variabile.
Infine, il file viene chiuso e rinominato.
Ricordate: il file deve essere chiuso prima di iniziare qualunque altra opearazione.
Quindi se create un file ci potete scrivere, ma prima di poter leggere dovete chiudere il file e riaprirlo col comando **OpenFileRead()**;per cancellare o rinominare un file, questo deve essere chiuso.

Testo copiabile	Screenshot del BricxCC

`#define OK LDR_SUCCESS`
`task main()`
`{`
`byte fileHandle;`
`short fileSize;`
`short bytesWritten;`
`string read;`
`string write;`
`DeleteFile("Danny.txt");`
`DeleteFile("DannySays.txt");`
`CreateFile("Danny.txt", 512, fileHandle);`
`for(int i=2; i<=10; i++ )`
`{`
`write = "NXT is cool ";`	

```
            string tmp = NumToStr(i);
        write = StrCat(write,tmp," times!");
    WriteLnString(fileHandle,write, bytesWritten);
                    }
            CloseFile(fileHandle);
    RenameFile("Danny.txt","DannySays.txt");
                    }
```

```
#define OK LDR_SUCCESS

task main(){
   byte fileHandle;
   short fileSize;
   short bytesWritten;
   string read;
   string write;
   DeleteFile("Danny.txt");
   DeleteFile("DannySays.txt");
   CreateFile("Danny.txt", 512, fileHandle);
   for(int i=2; i<=10; i++ ){
      write = "NXT is cool ";
      string tmp = NumToStr(i);
      write = StrCat(write,tmp," times!");
      WriteLnString(fileHandle,write, bytesWritten);
   }
   CloseFile(fileHandle);
   RenameFile("Danny.txt","DannySays.txt");
}
```

Per controllare il risultato, andate ,in Brixcc, nella voce di menu **Tools-NXT Explorer** , caricate **Danny.txt** sul PC e date un' occhiata.

Pronti per l'esempio successivo? Creeremo una tabella di caratteri ASCII.

Testo copiabile	Screenshot del BricxCC

```
                task main()
                    {
                 byte handle;
if (CreateFile("ASCII.txt", 2048, handle) == NO_ERR)
                    {
            for (int i=0; i < 256; i++)
                     {
                 string s = NumToStr(i);
                  int slen = StrLen(s);
              WriteBytes(handle, s, slen);
                  WriteLn(handle, i);
                     }
                CloseFile(handle);
                    }
                }
```

```
task main(){
   byte handle;
   if (CreateFile("ASCII.txt", 2048, handle) == NO_ERR) {

   for (int i=0; i < 256; i++) {
      string s = NumToStr(i);
      int slen = StrLen(s);
      WriteBytes(handle, s, slen);
      WriteLn(handle, i);
    }
    CloseFile(handle);
  }
}
```

Questo programma davvero semplice crea un file, e se non ci sono stati errori, ci scrive un numero da 0 a 255 (convertendo prima il numero in stringa) con **WriteBytes(handle, s, slen)** che è un altro sistema di scrivere stringe senza farle seguire da un INVIO (carriage return).
Poi scrive il numero non convertito, con **WriteLn(handle, value)** che aggiunge un INVIO (carriage return). Il risultato che potete visionare aprendo il file ASCII.txt con un editor di testi (come ad esempio Windows Notepad) è questo: il numero scritto come stringa è scritto in maniera da renderlo comprensibile agli umani, mentre il numero scritto come valore esadecimale è interpretato e visualizzato come un codice ASCII.
Non mi resta che mostrarvi le ultime due importanti funzioni: **ReadLnString** per leggere stringhe dai files, e **ReadLn** per leggere dei numeri.
Per esemplificare la prima funzione ecco un programma: il **task** main chiama la subroutine CreateRandomFile che crea un file contenente una quantità casuale di numeri (scritti come stringhe).
Potete anche commentare questa riga ed utilizzare per i vostri scopi un altro file creato a mano.
Poi il **task** main apre questo file per la lettura, legge una riga alla volta, fino alla fine del file, usando la funzione **ReadLnString** e mostra il testo letto.
Nella subroutine **CreateRandomFile** generiamo una sequenza di numeri casuali, li convertiamo in stringa, e li scriviamo sul file.
La funzione **ReadLnString** accetta come argomenti una variabile stringa e l'handle del file.
Dopo la sua chiamata la stringa conterrà una riga di testo, e la funzione restituirà un codice d'errore che ci sarà utile per capire se abbiamo raggiunto la fine del file.

Testo copiabile	Screenshot del BricxCC

```
            #define FILE_LINES 10
    sub CreateRandomFile(string fname, int lines)
                    {
                  byte handle;
                    string s;
                 int bytesWritten;
                DeleteFile(fname);
                int fsize = lines*5;
            //create file with random data
     if(CreateFile(fname, fsize, handle) == NO_ERR)
                     {
                    int n;
                repeat(FILE_LINES)
                      {
                 int n = Random(0xFF);
```

```
                    s = NumToStr(n);
                WriteLnString(handle,s,bytesWritten);
                          }
                  CloseFile(handle);
                        }
                      }
                    task main()
                          {
                     byte handle;
                      int fsize;
                     string buf;
                   bool eof = false;
         CreateRandomFile("rand.txt",FILE_LINES);
    if(OpenFileRead("rand.txt", fsize, handle) == NO_ERR)
                           {
               TextOut(10,LCD_LINE2,"Filesize:");
                   NumOut(65,LCD_LINE2,fsize);
                            Wait(600);
                   until (eof == true)
                            {
              // read the text file till the end
        if(ReadLnString(handle,buf) != NO_ERR) eof = true;
                            ClearScreen();
                  TextOut(20,LCD_LINE3,buf);
                           Wait(500);
                            }
                          }
                  CloseFile(handle);
                          }
```

```
#define FILE_LINES 10

sub CreateRandomFile(string fname, int lines){
   byte handle;
   string s;
   int bytesWritten;
   DeleteFile(fname);
   int fsize = lines*5;
   //create file with random data
   if(CreateFile(fname, fsize, handle) == NO_ERR) {
      int n;
      repeat(FILE_LINES) {
         int n = Random(0xFF);
         s = NumToStr(n);
         WriteLnString(handle,s,bytesWritten);
      }
      CloseFile(handle);
   }
}

task main(){
   byte handle;
   int fsize;
   string buf;
   bool eof = false;
   CreateRandomFile("rand.txt",FILE_LINES);
   if(OpenFileRead("rand.txt", fsize, handle) == NO_ERR) {
     TextOut(10,LCD_LINE2,"Filesize:");
     NumOut(65,LCD_LINE2,fsize);
     Wait(600);
     until (eof == true){   // read the text file till the
         if(ReadLnString(handle,buf) != NO_ERR) eof = true;
         ClearScreen();
         TextOut(20,LCD_LINE3,buf);
         Wait(500);
     }
   }
   CloseFile(handle);
}
```

Nell' ultimo esempio vi mostrerò come leggere dei numeri da un file.
Colgo l'occasione per farvi un piccolissimo esempio di compilazione condizionata.
All' inizio del codice, c'è una definizione che non è usata nè per una macro, nè per un alias: la chiamiamo semplicemente INT.(verifica traduzione) Segue una istruzione al preprocessore

```
#ifdef INT
 ...Codice...
#endif
```

che dice semplicemente al compilatore di compilare il codice compreso tra gli statements if solo se INT è stata definita.
In questo modo, se definiamo INT, verrà compilata la sezione di codice all' inizio del programma, mentre se invece fosse definita LONG verrebbe compilata la seconda versione. Questo sistema mi permette di mostrare come con la stessa funzione **ReadLn(handle, val)** posso leggere alternativamente da un file sia numeri INT (a 16 bit) sia numeri LONG (a 32 bit).
Come prima, la funzione accetta un handle ed una variabile numerica e restituisce un nomero ed un codice d'errore.
Se la variabile numerica è INT, la funzione legge 2 byte dal file, se la variabile è LONG ne legge 4.
Nella stesa maniera possiamo leggere anche le variabili bool.

Testo copiabile	Screenshot del BricxCC

```
        #define INT // INT or LONG
                #ifdef INT
              task main ()
                    {
           byte handle, time = 0;
            int n, fsize,len, i;
                  int in;
           DeleteFile("int.txt");
     CreateFile("int.txt",4096,handle);
    for (int i = 1000; i<=10000; i+=1000)
                    {
               WriteLn(handle,i);
                    }
             CloseFile(handle);
```

```
    OpenFileRead("int.txt",fsize,handle);
      until (ReadLn(handle,in)!=NO_ERR)
                    {
                 ClearScreen();
            NumOut(30,LCD_LINE5,in);
                   Wait(500);
                    }
             CloseFile(handle);
                    }
                 #endif
               #ifdef LONG
                task main ()
                    {
              byte handle, time = 0;
               int n, fsize,len, i;
                   long in;
            DeleteFile("long.txt");
       CreateFile("long.txt",4096,handle);
  for (long i = 100000; i<=1000000; i+=50000)
                    {
                  WriteLn(handle,i);
                    }
              CloseFile(handle);
     OpenFileRead("long.txt",fsize,handle);
       until (ReadLn(handle,in)!=NO_ERR)
                    {
                 ClearScreen();
            NumOut(30,LCD_LINE5,in);
                   Wait(500);
                    }
              CloseFile(handle);
                    }
                 #endif
```

```
#define INT //  INT or LONG

#ifdef INT
task main () {
   byte handle, time = 0;
   int n, fsize,len, i;
   int in;
   DeleteFile("int.txt");
   CreateFile("int.txt",4096,handle);
   for (int i = 1000; i<=10000; i+=1000){
      WriteLn(handle,i);
   }
   CloseFile(handle);
   OpenFileRead("int.txt",fsize,handle);
   until (ReadLn(handle,in)!=NO_ERR){
      ClearScreen();
      NumOut(30,LCD_LINE5,in);
      Wait(500);
   }
   CloseFile(handle);
}
#endif

#ifdef LONG
task main () {
   byte handle, time = 0;
   int n, fsize,len, i;
   long in;
   DeleteFile("long.txt");
   CreateFile("long.txt",4096,handle);
   for (long i = 100000; i<=1000000; i+=50000){
      WriteLn(handle,i);
   }
   CloseFile(handle);
   OpenFileRead("long.txt",fsize,handle);
   until (ReadLn(handle,in)!=NO_ERR){
      ClearScreen();
      NumOut(30,LCD_LINE5,in);
      Wait(500);
   }
   CloseFile(handle);
}
#endif
```

Sommario

In questo capitolo abbiamo preso in considerazione le funzioni avanzate dell' NXT:
---- Il timer ad alta risoluzione,
---- Il display a matrice di punti, ed
---- Il filesystem

Considerazioni finali

Se avete completato questo tutorial, potete considerarvi degli esperti di NXC. Se ancora non l'avete fatto, è tempo di mettervi a sperimentare da voi.
Con creatività, nell'assemblaggio e nella programmazione,potrete far compiere all'NXT operazioni incredibili.
Questo tutorial non copre tutti gli aspetti della programmazione del BricxCC. Vi raccomando di leggere interamente la guida alla programmazione dell NXC. NXC è in continuo sviluppo, future versioni implementeranno certamente funzionalità maggiori.
Molti concetti riguardanti la programmazione non sono stati qui trattati.
In particolare non abbiamo considerato il comportamento intelligente dei robot capaci di imparare dall' esperienza, ed altri aspetti dell' intelligenza artificiale.
E' inoltre possibile pilotare il robot direttamente tramite il PC. Questo richiede che scriviate un programma in un linguaggio come C++, Visual Basic, Java o Delphi.
E' possibile far interagire un simile programma con un programma che gira su un robot NXT, e questa combinazione risulta davvero potente.
Se siete interessati a questo tipo di programmazione del vostro robot, incominciate con lo scaricare i Fantom SDK e i documenti Open Source, dalla sezione NXTreme del sito web di Lego Mindstorm. http://mindstorms.lego.com/Openview/NXTreme.aspx
Il web è una ottima sorgente ove attingere ulteriori informazioni. Altri importanti punti di partenza sono sul LUGNET the Lego Users Groups Network (sito non ufficiale) http://www.lugnet.com/robotics/nxt

Visite