

# Il Linguaggio di Programmazione NXC *parte\_1*

## 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 **Legò 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 **Legò 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 scelgono 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.

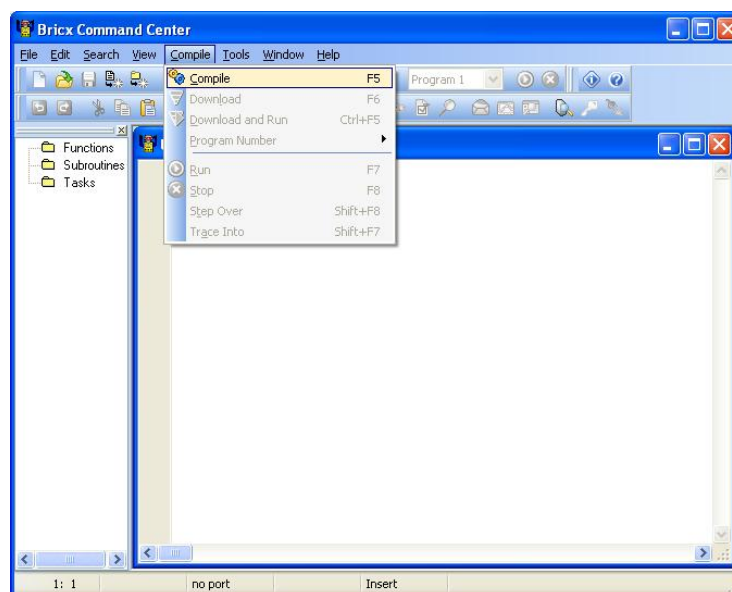
**Tools- Watching the Brick.** Bello in ogni fase. Tiene sotto controllo i valori dell' NXT e riproduce addirittura un grafico della loro variazione nel tempo. Molto comodo per testare il funzionamento dei programmi una volta realizzati.

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 muovono i suoi motori

**Tools-Brick Joystick** 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 musicchette da trasferire e fare eseguire all' NXT Ci sono dei programmi di conversione che trasformano i comuni file .wav 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.

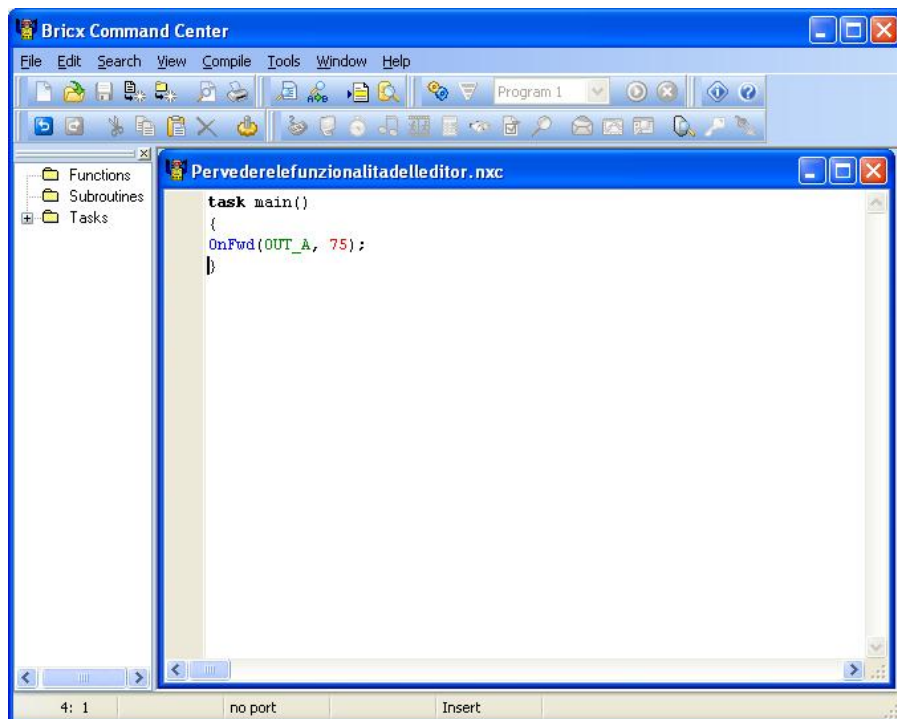
**Tools – Download Firmware** Ci aiuta a scaricare sull' NXT il firmware più aggiornato.



## 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.



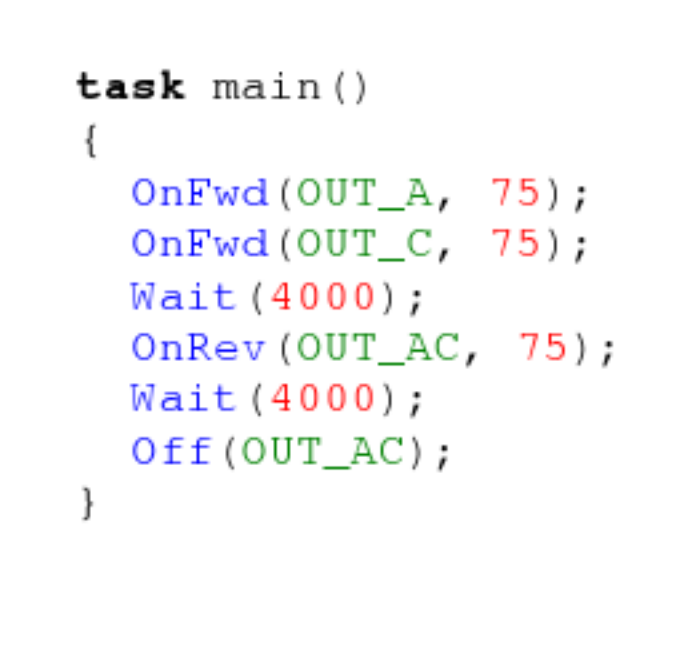
## Scrittura di un primo programma

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
<pre>task main() {   OnFwd(OUT_A, 75);   OnFwd(OUT_C, 75);   Wait(4000);   OnRev(OUT_AC, 75);   Wait(4000);   Off(OUT_AC); }</pre>	 <pre>task main() {   OnFwd(OUT_A, 75);   OnFwd(OUT_C, 75);   Wait(4000);   OnRev(OUT_AC, 75);   Wait(4000);   Off(OUT_AC); }</pre>

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 indicare 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 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 vogliamo 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.

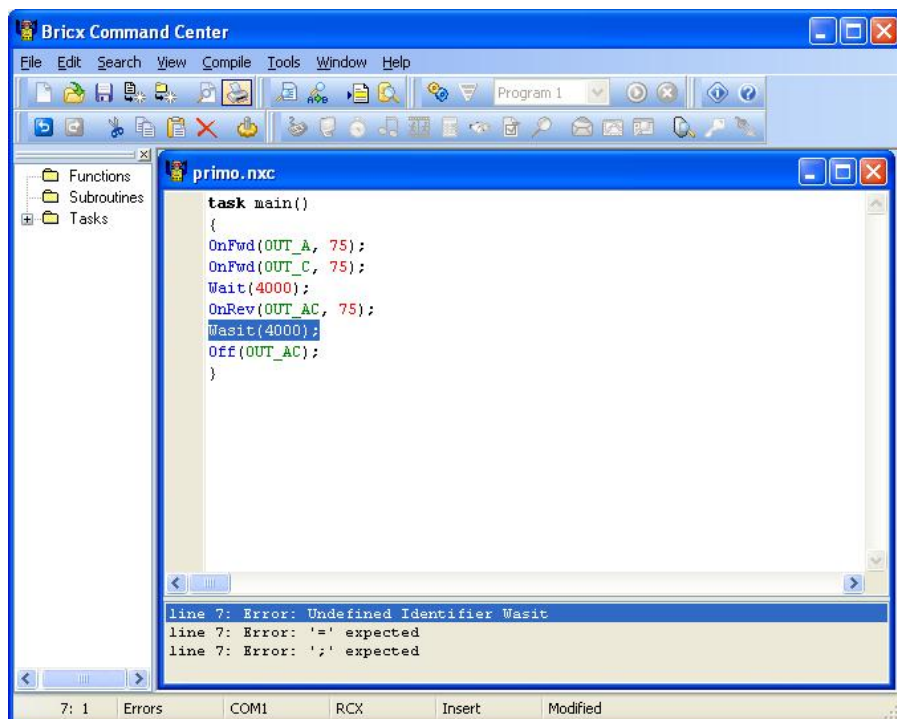
Quando ci sono errori multipli si comincia sempre a risolvere il primo in alto nella finestra, e si continua poi verso il basso.

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.



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

line 7: Error: Undefined Identifier Wasit  
line 7: Error: '=' expected  
line 7: Error: ';' expected

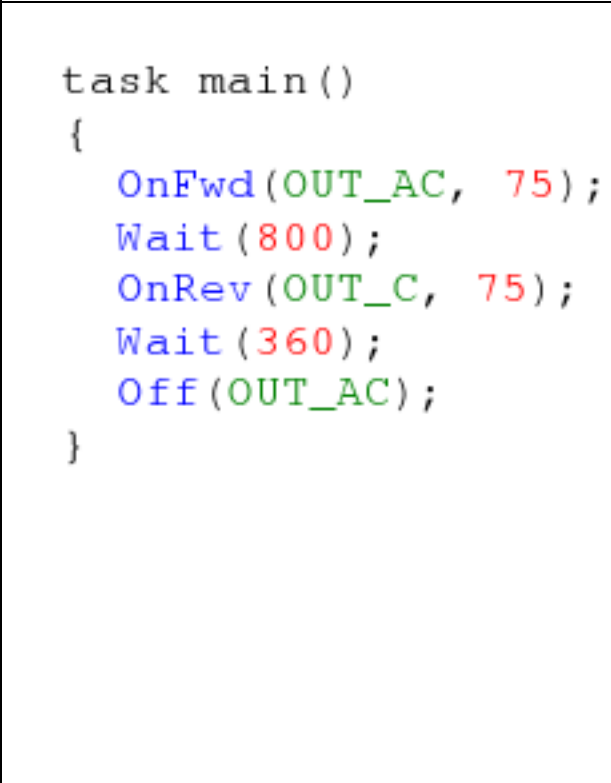
## Un programma più interessante

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 qualcosa 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
<pre>task main() {   OnFwd(OUT_AC, 75);    Wait(800);    OnRev(OUT_C, 75);    Wait(360);    Off(OUT_AC); }</pre>	 <pre>task main() {   OnFwd(OUT_AC, 75);   Wait(800);   OnRev(OUT_C, 75);   Wait(360);   Off(OUT_AC); }</pre>

Analizziamone il codice:

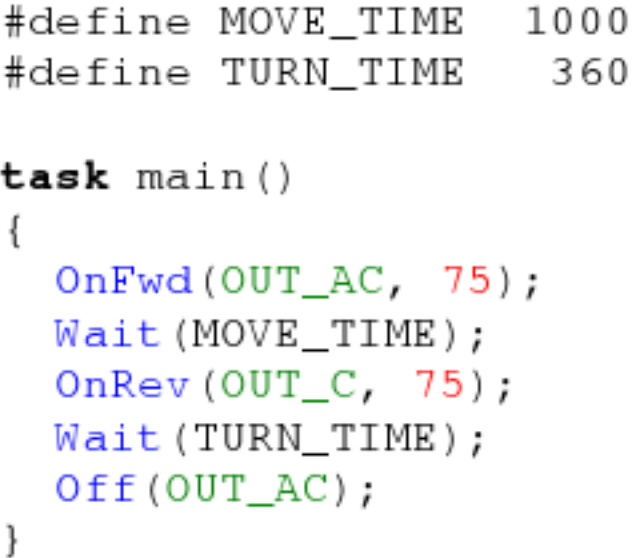
- Si apre il **task main** (dopodichè si apre una parentesi graffa)
- Si fanno girare in avanti i motori alle uscite **A e C** con "velocità" 75 (il 75 % del massimo)
- Si aspettano 800 millesimi di secondo (0,8 secondi)

- 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 superficie 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 possiamo 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
<pre>#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); }</pre>	 <pre>#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); }</pre>

```
}

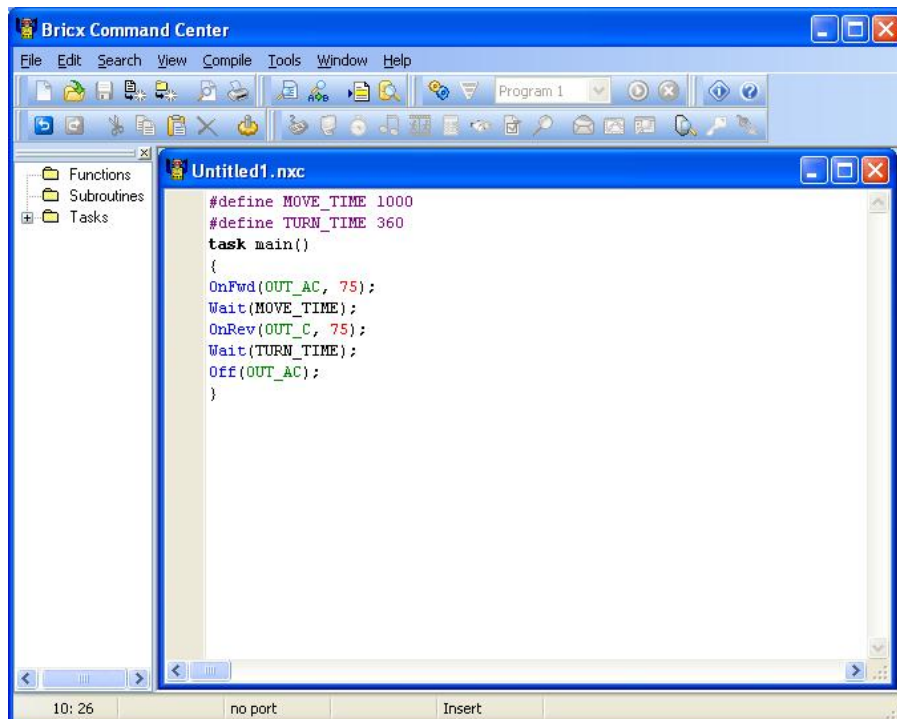
```

In questo codice le prime due righe definiscono due **COSTANTI**. Abbiamo cioè 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.

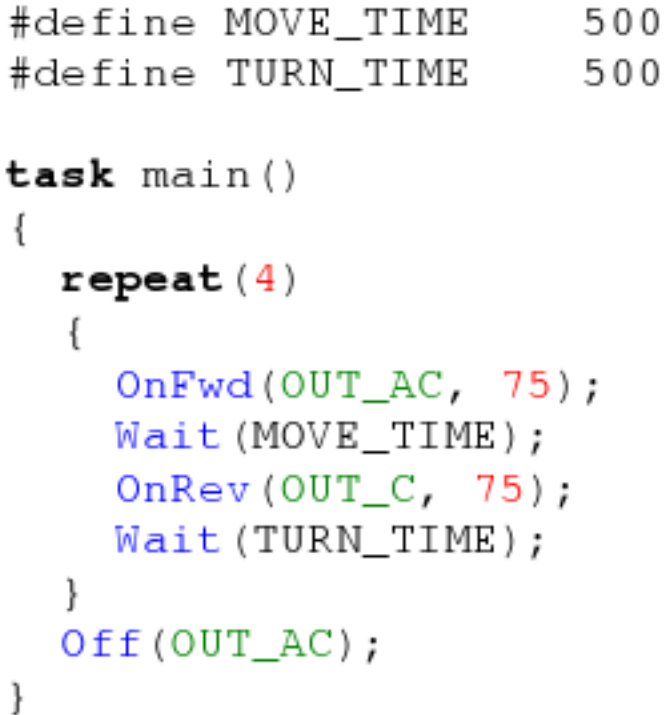
L'uso delle costanti rende il programma più facile da interpretare, ed al contempo da modificare.

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, utilizzare 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
<pre>#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); }</pre>	 <pre>#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); }</pre>

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:

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
-----------------	------------------------

<pre> #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); } </pre>	<pre> #define MOVE_TIME 1000 #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); } </pre>
--	---

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
-----------------	------------------------

<pre> #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);     } </pre>	<pre> #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); } </pre>
---	---

## 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 sé 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. Perché 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 `*/`.

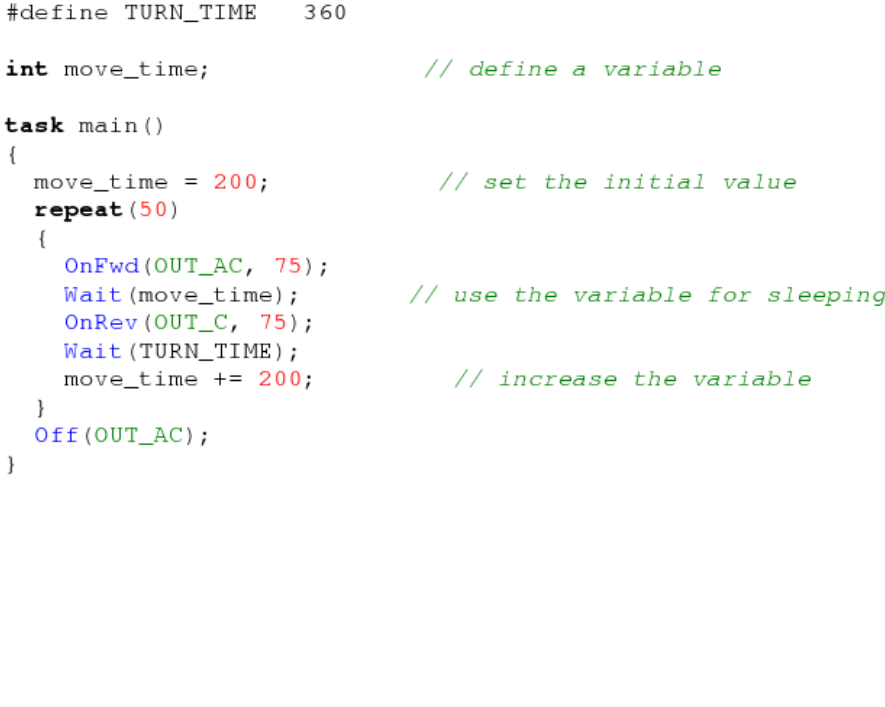
## Uso delle Variabili

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 cassette vuote: 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
<pre>#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); }</pre>	 <pre>#define TURN_TIME 360 int move_time; // define a variable task main() {   move_time = 200; // set the initial value   repeat(50)   {     OnFwd(OUT_AC, 75);     Wait(move_time); // use the variable for sleeping     OnRev(OUT_C, 75);     Wait(TURN_TIME);     move_time += 200; // increase the variable   }   Off(OUT_AC); }</pre>

Le linee di codice interessanti sono indicate da commenti:

- **int move\_time;** all' inizio del codice dichiariamo una variabile. Poniamo attenzione ad ogni dettaglio 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 variabile 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 costanti, 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 eseguito 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 incrementata 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, purché rispettiamo i tipi delle variabili.