



# UNIVERSITÀ DEGLI STUDI DI PADOVA

Facoltà di Ingegneria

Corso di Laurea Triennale in Ingegneria Informatica

## **Realizzazione di un simulatore semplificato del robot educativo Mindstorm NXT**

Relatore: Prof. Michele Moro

Laureando: Filippo Buletto  
Matricola: 560362

Anno Accademico 2008-2009



*Ad Alice per avermi sostenuto,  
alla mia famiglia per aver creduto in me,  
allo scoutismo per avermi reso più forte.*



# Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
1.1	TERECop . . . . .	2
1.2	Strumenti . . . . .	2
1.2.1	Java . . . . .	3
1.2.2	NetBeans . . . . .	3
1.2.3	NXT-G . . . . .	3
1.2.4	NXC . . . . .	4
1.3	Definizioni . . . . .	5
<b>2</b>	<b>Mindstorm NXT</b>	<b>7</b>
2.1	Kit simulato . . . . .	7
2.1.1	Brick NXT . . . . .	7
2.1.2	Motori . . . . .	8
2.1.3	Sensori . . . . .	9
2.2	Linguaggi di programmazione utilizzati . . . . .	11
2.2.1	NeXT Byte Codes . . . . .	11
2.2.2	Not eXactly C . . . . .	13
2.2.3	NXT-G graphical language . . . . .	15
2.3	Altri Linguaggi disponibili . . . . .	15
2.3.1	Microsoft Robotic Studio . . . . .	15
2.3.2	leJOS . . . . .	16
2.3.3	Ruby-NXT . . . . .	17
2.3.4	NXT-Python . . . . .	17
<b>3</b>	<b>NXTSimulator</b>	<b>19</b>
3.1	Premessa . . . . .	19
3.2	Guida all'utilizzo dell'ambiente simulato . . . . .	20
3.2.1	L'interfaccia . . . . .	20
3.2.2	Guida . . . . .	24
3.3	Manuale tecnico . . . . .	27
3.3.1	Bug segnalati e limitazioni preesistenti . . . . .	27
3.3.2	Integrazione dei sensori di input . . . . .	60
3.3.3	Bug noti . . . . .	67
<b>4</b>	<b>Conclusioni e lavoro futuro</b>	<b>69</b>

A Istruzioni per il controllo di flusso	71
---	----

# Elenco delle figure

1.1	Esempio di programma creato utilizzando NXT-G. . . . .	4
2.1	Brick NXT. . . . .	8
2.2	Servomotore. . . . .	8
2.3	Sensore di luce. . . . .	9
2.4	Sensore di suono. . . . .	10
2.5	Sensore di tatto. . . . .	10
2.6	Sensore di prossimità. . . . .	11
2.7	Finestra di compilazione codice NXC tramite NBC. . . . .	14
2.8	MSRS Visual Programming Language. . . . .	16
3.1	Finestra principale di NXT Simulator. . . . .	20
3.2	Finestra per la scelta del file RXE. . . . .	21
3.3	Menù File e Aggiungi. . . . .	22
3.4	Menù Rimuovi, Simulazione e Lingua. . . . .	23
3.5	Comportamento del simulatore al variare del valore di prossimità. . . . .	27
3.6	Disposizione degli Array di Cluster. . . . .	34
3.7	Disposizione degli Array di numeri all'interno della struttura "array". . . . .	35
3.8	Blocco Move di NXT-G con fattore di sterzata modificato. . . . .	51
3.9	Rapporto tra le velocità angolari al variare di Steering. . . . .	52
3.10	Programma NXT-G per il movimento di tre motori. . . . .	57
4.1	Assemblaggio del kit LEGO® Mindstorm® NXT . . . . .	70





## Elenco delle tabelle

3.1	Struttura dei file RXE. . . . .	28
3.2	Struttura header del file RXE. . . . .	28
3.3	Struttura dei campi della DSTOC. . . . .	29
3.4	Struttura di un Dope Vector. . . . .	31
3.5	Struttura di un Clump Record. . . . .	32
3.6	Esempio di cluster con un elementi di tipo array. . . . .	38
3.7	Esempio di array con un elementi di tipo array. . . . .	38
3.8	Elenco proprietà dei sensori. . . . .	48



# Elenco dei listati codice

1.1	Codice NXC di esempio: esegue le stesse azioni del programma in figura 1.1 . . . . .	5
2.1	Codice NeXT Byte Codes di esempio . . . . .	11
2.2	arraybuild.nxc . . . . .	14
2.3	arraybuild.nxc decompilato in NBC . . . . .	14
3.1	UltrasonicoPorta3_MotoreRallentaAvvicinandosiStoppaA5cm.nxc . . . . .	24
3.2	OP_SHORT_MOV errato . . . . .	36
3.3	OP_SHORT_MOV corretto . . . . .	36
3.4	Caso array di cluster, OP_INDEX errato . . . . .	37
3.5	Caso array di array, OP_INDEX errato . . . . .	37
3.6	Caso array di cluster, OP_INDEX corretto . . . . .	39
3.7	Caso array di array, OP_INDEX corretto . . . . .	39
3.8	Caso array di array, OP_REPLACE corretto . . . . .	40
3.9	OP_ARRBUILD corretto . . . . .	42
3.10	OP_ARRSUBSET corretto . . . . .	43
3.11	OP_ARRINIT corretto . . . . .	45
3.12	OP_NUMTOSTRING . . . . .	46
3.13	OP_SETIN . . . . .	47
3.14	OP_GETIN . . . . .	47
3.15	OP_SETOUT . . . . .	48
3.16	OP_GETOUT . . . . .	48
3.17	OutputPortConfigurationProperties.java . . . . .	52
3.18	Spostamento corretto del puntatore dei valori di default . . . . .	55
3.19	Esempio del caso array nell'array di cluster . . . . .	55
3.20	Frammento di codice NeXT Byte Codes del RXE anormale . . . . .	56
3.21	Codice NXC equivalente al programma in figura 3.10. . . . .	57
3.22	Classe SensorData . . . . .	60
3.23	Classe InputPortConfigurationProperties . . . . .	61
3.24	Metodo stopSim() della classe NXTSView . . . . .	65
3.25	Metodo setStatus(String str) della classe SensorTouchPanel . . . . .	66



# Capitolo 1

## Introduzione

Il progetto NXT Simulator è cominciato nell'anno 2008 presso il dipartimento di Ingegneria dell'informazione dell'Università di Padova, con lo scopo di produrre un simulatore funzionale del robot contenuto nel kit LEGO<sup>®</sup> Mindstorm<sup>®</sup> NXT<sup>1</sup>. L'intento è di simulare al meglio il comportamento del robot tramite un ambiente grafico che mostri all'utente/programmatore le reazioni dei vari componenti collegati a seconda degli input e della programmazione dello stesso.

Questo elaborato descrive come sono state risolte molte delle lacune presenti nella versione su cui ho lavorato, in particolare per quel che riguarda il motore del simulatore: l'interprete dei comandi. In molti punti infatti erano presenti errori logici non identificabili in fase di compilazione ma solo a run-time con prove specifiche e mirate. Come tutti i bug, le cause delle problematiche nel software ed i sintomi correlati sono "geograficamente" separati nel codice ed a scomparsa/ricomparsa temporanea a seconda delle modifiche che si effettuano. Saranno presentate inoltre le nuove funzioni introdotte e la nuova guida all'utente per l'utilizzo del programma.

La situazione attuale del progetto è quella di completezza delle funzioni basilari di input (quindi sensori standard presenti nel kit NXT) e di output (simulazione del movimento dei motori collegati al robot), lo sviluppo potrà quindi proseguire con l'integrazione di funzioni avanzate quali l'interpretazione delle istruzioni di conversione e le chiamate di sistema, le quali permettono di sfruttare l'LCD integrato del robot, il sistema audio, ecc...

---

<sup>1</sup>Ulteriori dettagli nel capitolo 2

## 1.1 TERECoP

TERECOP, acronimo di *Teacher Education on Robotics-Enhanced Constructivist Pedagogical Methods*, è un progetto nel quale è inserito anche il Dipartimento di Ingegneria dell'Informazione. TERECoP trae ispirazione da:

- le teorie costruttiviste di Jean Piaget che sostengono che l'apprendimento non sia tanto il risultato di un passaggio di conoscenze quanto un processo attivo di costruzione della conoscenza basato su esperienze ricavate dal mondo reale e collegate a preconoscenze uniche e personali (Piaget 1972)<sup>2</sup>.
- la filosofia didattica costruzionista di S. Papert che introduce l'idea che il processo di apprendimento risulta decisamente più efficace quando vengano introdotti artefatti cognitivi, ovvero oggetti e dispositivi che si basino su concetti familiari allo studente.

Più di recente l'attenzione si è concentrata sulla costruzione di modelli informatici. Il Computer-aided modelling nell'ambito dell'apprendimento è considerato un valido strumento per migliorare l'acquisizione e lo sviluppo del pensiero autonomo degli studenti. [...] Attualmente si ritiene che la programmazione, intesa come un ambito educativo generale per la costruzione di modelli e strumenti, possa sostenere un apprendimento costruzionista lungo lo sviluppo del curriculum scolastico (Papert, 1992)<sup>3</sup>.

Il sistema Lego Mindstorm si dimostra essere uno strumento flessibile per l'apprendimento costruzionista offrendo l'opportunità di progettare e costruire strutture robotiche con tempo e fondi limitati. Il progetto si pone come obiettivo complessivo quello di sviluppare una struttura di supporto per corsi di formazione degli insegnanti, al fine di aiutarli a realizzare attività formative di tipo costruttivista con l'uso della robotica e dar loro la possibilità di divulgare attraverso questa struttura le proprie esperienze. [1]

La disponibilità in questo ambito di un simulatore ancorché semplificato consente di disporre di uno strumento rappresentante un valido aiuto per la costruzione ed il test delle attività riguardanti il robot NXT.

## 1.2 Strumenti

Gli strumenti utilizzati per sviluppare NXTSimulator sono molteplici: per la programmazione è stato utilizzato il linguaggio Java e l'ambiente di sviluppo gratuito NetBeans, per la parte di test e analisi del codice binario del robot, il linguaggio grafico NXT-G ed il linguaggio a riga di comando NXC.

Segue quindi una breve descrizione dei suddetti strumenti.

---

<sup>2</sup>Piaget, J. (1972). *The Principles of Genetic Epistemology*. N. Y.: Basic Books.

<sup>3</sup>Papert, S. (1992). *The Children's Machine*. N.Y.: Basic Books.

### 1.2.1 Java

Java è un linguaggio di programmazione orientato agli oggetti ad uso gratuito<sup>4</sup> e di proprietà di Sun Microsystems<sup>®</sup>. La sintassi deriva da quella del C e C++, ma a differenza di questi Java consente di creare programmi eseguibili su molte piattaforme tramite la sua struttura "a macchina virtuale"<sup>5</sup>. Grazie alla possibilità di un uso su più sistemi operativi senza dover tradurre il codice sorgente, alla relativa semplicità rispetto ad altri linguaggi di programmazione più complessi ed alla vasta disponibilità di librerie e supporto, Java è stato individuato come linguaggio di programmazione che più si addice allo sviluppo del progetto NXT Simulator. La versione del kit di sviluppo scelta è la 1.6, la più aggiornata e presente sulle maggiori piattaforme<sup>6</sup>. L'eseguibile è stato sviluppato su piattaforma Macintosh ma è stato testato anche su piattaforma Windows (anche nella sua ultima versione Windows 7); non sono stati riscontrati problemi nel passaggio quindi si può affermare che la scelta di utilizzare Java è stata totalmente soddisfacente.

### 1.2.2 NetBeans

NetBeans è un ambiente di sviluppo multi-linguaggio scritto interamente in Java. È l'ambiente scelto da Sun Microsystems<sup>®</sup> come IDE ufficiale, in contrapposizione al più diffuso Eclipse.

Le versioni utilizzate per lo sviluppo di NXT Simulator sono state molteplici durante il ciclo di vita del programma, in particolare per questo elaborato sono state utilizzate la versione 6.6 e la più recente 6.7.

NetBeans offre un ambiente completo per lo sviluppo di applicazioni grafiche (e non) e si è rivelato un valido strumento soprattutto per quanto riguarda il debugging "al volo" del codice; utilizzando la tecnica JIT (Just In Time) è possibile infatti controllare ogni aspetto del Thread in corso di esecuzione: le variabili ed il loro contenuto, il flusso di istruzioni ed i salti condizionati, l'interazione tra la parte simulatore e la parte di interfaccia grafica, ecc...

### 1.2.3 NXT-G

NXT-G è un linguaggio grafico per programmare il robot LEGO Mindstorm NXT, esso è basato sulla tecnologia LabVIEW: un ambiente di sviluppo grafico elaborato da National Instruments, appositamente modificato per rispettare le specifiche del firmware NXT. [2]

Questo tipo di programmazione visuale è definito G-Language<sup>7</sup>, esso si caratterizza per l'assenza di codice sotto forma di testo, il quale viene invece salvato in codice binario compatibile esclusivamente con LabVIEW. La definizione degli algoritmi del programma avviene tramite icone ed oggetti grafici (definiti

---

<sup>4</sup>Licenza GNU General Public License

<sup>5</sup>JVM, Java Virtual Machine

<sup>6</sup>Microsoft Windows, Linux ed Apple Mac OS X (solo 64bit)

<sup>7</sup>Graphic Language

”blocchi”) mentre lo scambio di informazioni fra blocchi avviene tramite linee di congiunzione, disposte a formare una specie di diagramma di flusso.

In figura 1.1 è possibile notare l'estrema semplicità della programmazione grafica, ma al contempo si osserva come già un piccolo programma richieda una strutturazione mediamente complessa, cosa che all'aumentare della dimensione del programma può creare difficoltà di gestione dell'ambiente e della struttura del diagramma stesso.

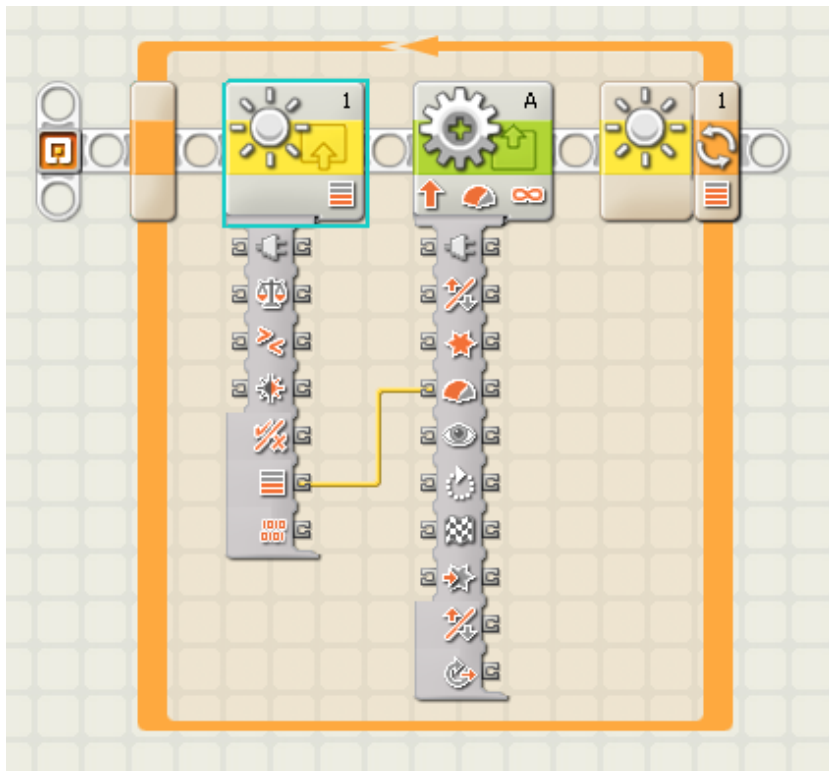


Figura 1.1: Esempio di programma creato utilizzando NXT-G.

Di norma NXT-G produce il file eseguibile caricandolo direttamente nel robot (via cavo o tramite wireless Bluetooth), per poterlo quindi salvare direttamente nel computer è stato necessario utilizzare un plug-in di terze parti.

Una nota negativa di NXT-G è anche quella di utilizzare un compilatore di file RXE (vedi definizione alla sezione 1.3) piuttosto avido di risorse, se paragonato ad altri compilatori come NBC<sup>8</sup>. Infatti esso utilizza molte strutture complesse ed un alto numero di variabili.

### 1.2.4 NXC

NXC è l'acronimo di Not eXactly C; esso è un linguaggio di programmazione simile al C come sintassi ma totalmente dedicato alla creazione di file RXE per

---

<sup>8</sup>vedi paragrafo 1.2.4



Mindstorm NXT. Questo linguaggio sfrutta il compilatore NBC<sup>9</sup> per la creazione dei file RXE, il quale traduce le istruzioni ad alto livello NXC in codice oggetto eseguibile per il robot. [3]

Questo linguaggio Open Source è quindi associato ad un compilatore/decompilatore che abbia la possibilità di mostrare, sotto forma di testo, le istruzioni del codice binario corrispondente al codice NXT. L'uso di questo strumento è stato determinante nei molti casi in cui il debug dei file RXE era molto difficoltoso tramite l'analisi del simulatore in Java.

Codice 1.1: Codice NXC di esempio: esegue le stesse azioni del programma in figura 1.1

```
task main(){
    SetSensorLight(0);
    int val = SensorScaled(0);
    OnFwd(0, val);
    while(val < 75){
        val = SensorScaled(0);

        SetOutput(0, 2, val);
    }
}
```

Questo codice dimostra la reazione del robot agli stimoli luminosi. La velocità del motore collegato alla porta A è direttamente proporzionale alla luminosità misurata dal sensore di luce alla porta 1; quando il valore di luce supera il 75% il programma termina.

## 1.3 Definizioni

Elenco dei termini e delle frasi tecniche utilizzate:

- **RXE**: file contenente codice binario eseguibile dal Brick NXT.
- **Brick NXT**: parte primaria del robot, esso contiene il microcomputer che esegue la macchina virtuale atta ad interpretare i file RXE, possiede porte per il controllo di motori e sensori, una porta USB, un controller wireless Bluetooth, è dotato inoltre di un display LCD, un piccolo altoparlante, vari tasti e innesti per collegarlo alle altre parti LEGO Technic.

---

<sup>9</sup>NeXT Byte Codes



# Capitolo 2

## Mindstorm NXT

### 2.1 Kit simulato

Il sistema LEGO<sup>®</sup> Mindstorm<sup>®</sup> NXT è un kit composto da vari elementi utili alla costruzione del robot, fra questi quelli interessati alla simulazione sono: i motori, i sensori (di tipo tattile, luminoso, sonoro e un sensore di prossimità a ultrasuoni) ed il Brick NXT che rappresenta quello che si può definire il cervello operativo del robot.

NXT ha rimpiazzato da 3 anni il kit RCX sempre della linea LEGO<sup>®</sup> Mindstorm<sup>®</sup>.

#### 2.1.1 Brick NXT

La traduzione di "Brick" è mattoncino: infatti l'aspetto del componente è quello di un grande mattoncino LEGO con annessi display pulsanti e porte di collegamento.

All'interno è presente un micro computer, il quale permette l'esecuzione dell'interprete dei comandi ed il controllo di tutte le periferiche ad esso connesso. Le specifiche del Brick sono:

- Microcontrollore 32-bit ARM7
- 256 Kbytes FLASH, 64 Kbytes RAM
- 8-bit AVR microcontrollore
- 4 Kbytes FLASH, 512 Byte RAM
- Bluetooth (Bluetooth Classe II V2.0 compatibile)
- Porta USB full speed (12 Mbit/s)
- 4 porte di input
- 3 porte di output
- Display LCD da 100 x 64 pixel

- Speaker - 8 bit fino a 16 KHz
- Fonte energetica: 6 batterie tipo AA



Figura 2.1: Brick NXT.

### 2.1.2 Motori

In particolare si tratta di servomotori, cioè a differenza dei motori tradizionali questi devono possedere bassa inerzia, linearità di coppia e velocità, rotazione uniforme e capacità di sopportare picchi di potenza.

I servomotori forniti in kit integrano inoltre un sensore di rotazione che dà la possibilità di tracciare la posizione dell'asse esterno del motore in gradi o rotazioni complete (con incertezza di circa un grado). Una rotazione equivale a 360 gradi.

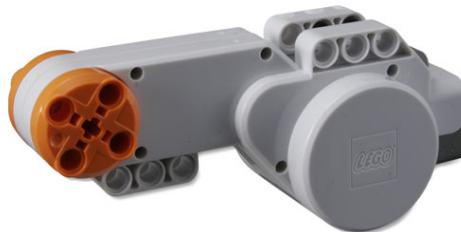


Figura 2.2: Servomotore.

### 2.1.3 Sensori

I sensori permettono al robot di ricevere informazioni riguardanti l'ambiente circostante, e rendono possibile l'interazione del robot con l'esterno. I sensori sono di vario tipo e produttore: qui saranno brevemente introdotti quelli contenuti nel kit Mindstorm<sup>®</sup> NXT, che sono poi resi disponibili nel simulatore.

**Sensore di luce** Questo sensore restituisce al Brick informazioni riguardanti la quantità di luce presente nella zona frontale al sensore stesso. Non potendo distinguere i vari colori il sensore rileva il livello di grigio valutando la luce su una scala che va da 0 a 100 (dal buio alla massima intensità misurabile). È anche dotato di un led rosso che emette luce per poterne leggere la quantità riflessa da una superficie.



Figura 2.3: Sensore di luce.

**Sensore di suono** Il sensore di suono possiede un microfono e può essere utilizzato per misurare l'ampiezza dei suoni (misurata come volume o rumorosità). Le unità di misura disponibili sono i dB ed i dBA (con dBA si intende rappresentare la risposta in frequenza dell'orecchio umano ai suoni, quindi in uno spettro di frequenze mirato). I valori restituiti dal sensore sono in percentuale e vanno dal 4-5% (silenzio totale in una stanza) al 30-100% (rilevato in caso di musica riprodotta ad alto volume o da delle persone che urlano).



Figura 2.4: Sensore di suono.

**Sensore di tatto** Composto da un singolo pulsante, il sensore di tatto ha tre possibili valori: premuto, rilasciato e "cliccato". Quest'ultimo valore è dato quando il bottone viene premuto e successivamente rilasciato in rapida successione, analogamente al click del mouse su un PC.



Figura 2.5: Sensore di tatto.

**Sensore di prossimità** Questo sensore permette di rilevare la distanza di un oggetto solido posto di fronte ad esso; per far ciò viene utilizzato un sistema ad ultrasuoni. Il robot riesce quindi ad elaborare informazioni riguardanti gli oggetti presenti nello spazio circostante, con un range che va dalla prossimità totale (intorno ai 4 cm) ad un massimo di 255 cm.



Figura 2.6: Sensore di prossimità.

## 2.2 Linguaggi di programmazione utilizzati

I principali linguaggi di programmazione descritti nella sezione 1.2, sono stati utilizzati per molteplici scopi nello sviluppo di NXT Simulator. Ogni programma produce in uscita un file binario eseguibile<sup>1</sup> secondo le specifiche del firmware 1.03, le quali sono descritte da LEGO nel documento: "*LEGO® MINDSTORMS® NXT Executable File Specification*". [4]

### 2.2.1 NeXT Byte Codes

Questo termine è utilizzato per riferirsi sia al linguaggio assembly interpretato dal Brick NXT, sia al compilatore che viene utilizzato per produrre i file binari scritti in tale linguaggio. [5]

In questo progetto è stato utilizzato un decompilatore di file RXE in linguaggio NBC, infatti analizzare la struttura del linguaggio omonimo è servito a scopi didattici per conoscere al meglio ogni possibile aspetto sui file binari RXE. Dati i gravi problemi che affliggevano il simulatore nella precedente versione, il miglior sistema di debug è stato quello di creare tramite NBC semplici file con alcuni listati di prova per le singole istruzioni, avendo quindi la possibilità di studiare e successivamente correggere il comportamento del simulatore in quelle determinate modalità.

Codice 2.1: Codice NeXT Byte Codes di esempio

```
//-----  
// Loops forever with motor turning back and forth by 360 degrees  
//-----  
dseg segment  
  
// Motor values  
theUF byte  
thePower sbyte  
theOM byte
```

---

<sup>1</sup>Vedi file RXE sezione 1.3

```

theRM byte
theRS byte OUT_RUNSTATE_RUNNING
theAngle ulong

rs byte
OldRotCount sword
RotCount sword
// timer vars
thenTick dword
nowTick dword

dseg ends

thread main

    // Run the motor A for till angle reached, power80
    set thePower, 80
Forever:
    set theAngle, 360
    set theOM, OUT_MODE_MOTORON+OUT_MODE_BRAKE
    set theRM, OUT_REGMODE_IDLE

    set theUF, UF_UPDATE_RESET_BLOCK_COUNT+UF_UPDATE_RESET_COUNT+
        UF_UPDATE_TACHO_LIMIT+UF_UPDATE_SPEED+UF_UPDATE_MODE
    setout OUT_A, OutputMode, theOM, RegMode, theRM, TachoLimit,
        theAngle, RunState, theRS, Power, thePower, UpdateFlags, theUF

    // Waits till angle reached
Running:
    getout rs, OUT_A, RunState
    brcmp EQ, Running, rs, OUT_RUNSTATE_RUNNING

    // Regulates for speed = 0
    set theOM, OUT_MODE_MOTORON+OUT_MODE_BRAKE+OUT_MODE_REGULATED
    set theRM, OUT_REGMODE_SPEED
    set theUF, UF_UPDATE_SPEED+UF_UPDATE_MODE
    setout OUT_A, OutputMode, theOM, RegMode, theRM, RunState, theRS,
        Power, 0, UpdateFlags, theUF

    // Verifies that motor doesn't rotate for 100ms, else loops
    getout RotCount, OUT_A, RotationCount
Stabilize:
    mov OldRotCount, RotCount
    gettick nowTick
    add thenTick, nowTick, 100 // wait 100ms

Waiting:
    getout RotCount, OUT_A, RotationCount
    brcmp NEQ, Stabilize, OldRotCount, RotCount
    gettick nowTick
    brcmp LT, Waiting, nowTick, thenTick

    // Change direction and loops
    neg thePower, thePower

```



```
    jmp Forever  
  
endt
```

### 2.2.2 Not eXactly C

Il potente linguaggio di programmazione C-Like NXC è stato profusamente utilizzato per lo studio del comportamento del simulatore sia per quanto riguarda l'implementazione delle nuove funzioni, quali la simulazione dei sensori di input, sia per la correzione delle istruzioni mal implementate.

Not eXactly C (NXC) è un linguaggio di programmazione ad alto livello simile ad NQC<sup>2</sup>. La versione utilizzata è quella per Mac OS X ma è disponibile anche per Windows e Linux. NXC è in fase di beta dato che non è tuttora completo, ma comprende una vasta serie di API per NXT ed è stato dichiarato come ben funzionante.

Il linguaggio supporta i seguenti costrutti: *if/else*, *while*, *do-while*, *repeat*, *for*, *switch*, *until*, *goto*, ed *asm {}*; "break" e "continue" sono utilizzabili all'interno dei cicli, anche "return" è disponibile per uscire da una routine o per restituire dati dalla stessa.

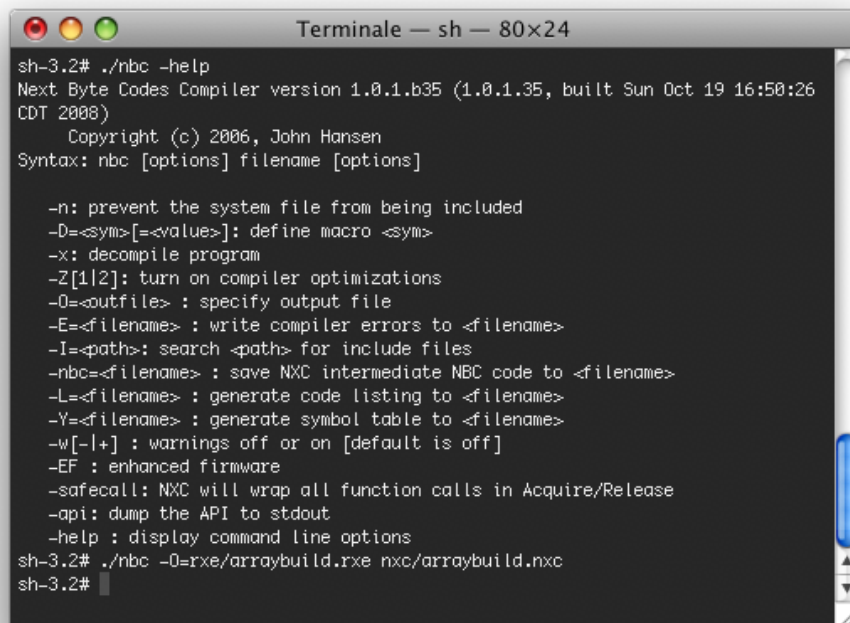
NXC supporta variabili globali e locali, ad ora supporta i seguenti tipi: *int*, *short*, *long*, *byte*, *char*, *bool*, *unsigned short*, *unsigned long*, *unsigned int*, *mutex*, *string*, ed *array* di qualsiasi tipo a parte *mutex* (*int* e *short* sono compilati nello stesso tipo di dato).

Gli array possono essere di qualsiasi dimensione e dichiarati utilizzando la sintassi a parentesi quadre (*int X[]*). Per array a dimensione fissa si utilizza la dichiarazione seguente: *int X[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}*; . Possono essere dichiarati anche array multidimensionali utilizzando l'API *ArrayInit*.

Basato sul compilatore/decompilatore NBC, non è previsto nessun IDE di sviluppo: infatti il file di testo con estensione NXC viene direttamente compilato in RXE (a meno di errori di sintassi) tramite linea di comando.

---

<sup>2</sup>Linguaggio utilizzato per il precedente modello RXC e non supportato da NXT



```

Terminale — sh — 80x24
sh-3.2# ./nbc -help
Next Byte Codes Compiler version 1.0.1.b35 (1.0.1.35, built Sun Oct 19 16:50:26
CDT 2008)
Copyright (c) 2006, John Hansen
Syntax: nbc [options] filename [options]

-n: prevent the system file from being included
-D=<sym>[=<value>]: define macro <sym>
-x: decompile program
-Z[1|2]: turn on compiler optimizations
-O=<outfile> : specify output file
-E=<filename> : write compiler errors to <filename>
-I=<path>: search <path> for include files
-nbc=<filename> : save NXC intermediate NBC code to <filename>
-L=<filename> : generate code listing to <filename>
-Y=<filename> : generate symbol table to <filename>
-w[+|-] : warnings off or on [default is off]
-EF : enhanced firmware
-safecall: NXC will wrap all function calls in Acquire/Release
-api: dump the API to stdout
-help : display command line options
sh-3.2# ./nbc -O=rxn/arraybuild.rxn nxc/arraybuild.nxc
sh-3.2#

```

Figura 2.7: Finestra di compilazione codice NXC tramite NBC.

**Esempio di utilizzo di NXC per il debug di un'istruzione** L'istruzione viene inserita in un programma NXC:

Codice 2.2: arraybuild.nxc

```

int X[] = {0, 1, 2, 3, 4, 5, 6, 7};
int Y[] = {10, 11, 12, 13, 14, 15, 16, 17};
int R[] = {6, 7, 8, 9, 10, 11, 12, 13};
int S[] = {13, 14, 15, 16, 17, 18, 19, 20};
int Z[];
int K[];

task main()
{
    ArrayBuild(Z, X, Y);
    ArrayBuild(K, R, S);
}

```

Codice 2.3: arraybuild.nxc decompilato in NBC

```

; arraybuild.rxn
; ----- variable declarations -----
dseg      segment
;----- definitions -----
;----- declarations -----
ub0000    byte
sb0001    sbyte    16

```

```

sb0002  sbyte
a0003   sword[]  0xD, 0xE, 0xF, 0x10, 0x11, 0x12, 0x13, 0x14
a0005   sword[]
a0007   sword[]
a0009   sword[]  0x6, 0x7, 0x8, 0x9, 0xA, 0xB, 0xC, 0xD
a000B   sdword[]
a000D   sword[]  0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7
a000F   sword[]  0xA, 0xB, 0xC, 0xD, 0xE, 0xF, 0x10, 0x11
dseg    ends
; ----- program code -----
        thread t000
        arrinit a000B, sb0002, sb0001
        arrbuild      a0005, a000D, a000F
        arrbuild      a0007, a0009, a0003
        exit          -1, -1
        endt

```

### 2.2.3 NXT-G graphical language

Questo ambiente è stato utilizzato per verificare la compatibilità di simulazione dei blocchi "Move" (di movimento dei motori) con il simulatore. Il codice eseguibile prodotto da NXT-G è molto complesso e dispendioso di risorse, ad esempio un solo blocco "Move" fa produrre al compilatore mediamente 450 linee di codice NBC diviso in 11 Thread, lo stesso programma scritto in NXC e seguentemente compilato non supera le 74 linee e 2 Thread.

L'alta complessità dei file RXE creati con NXT-G è stata utile strumento di verifica, per accertare la robustezza e la compatibilità del simulatore, e di individuazione di molti bug (che in programmi semplici come quelli prodotti dal compilatore NBC non erano visibili).

## 2.3 Altri Linguaggi disponibili

Altri linguaggi possono essere utilizzati per creare file RXE eseguibili dal Brick NXT, di seguito sono presentati i più importanti.

### 2.3.1 Microsoft Robotic Studio

MRS<sup>3</sup> è un ambiente di sviluppo creato da microsoft per piattaforme Windows, non è specifico per MINDSTORMS<sup>®</sup> NXT dato che promette supporto per molti tipi di robot diversi, ma ne è compatibile. MRS permette la programmazione sia testuale, tramite i linguaggi C# o VB.NET, che visuale (in maniera analoga a NXT-G), tramite MSRS Visual Programming Language. [6]

Per la creazione di un programma è necessario inserire le informazioni dettagliate riguardo tutte le periferiche collegate al robot che devono essere utilizzate, perciò si riconosce che questo non sia un ambiente per principianti o inesperti.

---

<sup>3</sup>Microsoft Robotic Studio

MRS utilizza una connessione diretta con il Brick NXT tramite collegamento Bluetooth per il controllo ed il caricamento del programma nella memoria del robot, fornisce quindi la possibilità di un controllo remoto tramite un'interfaccia web creata ad-hoc.

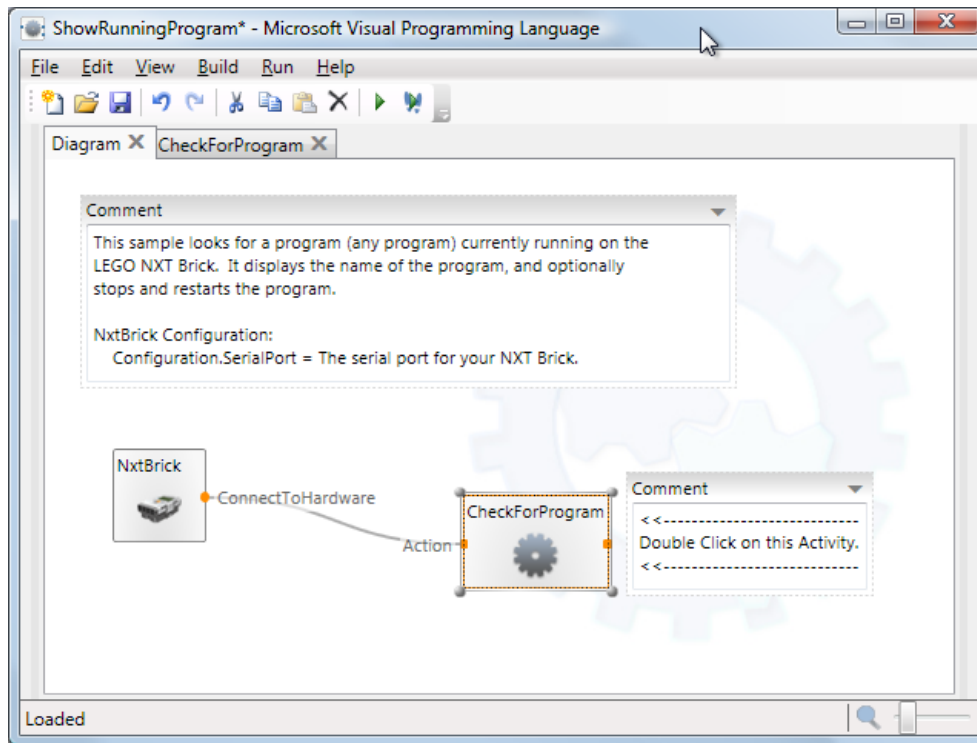


Figura 2.8: MSRS Visual Programming Language.

### 2.3.2 leJOS

leJOS<sup>4</sup> è il nome di una piccola JVM<sup>5</sup> adattata per essere eseguita all'interno del Brick NXT.

Le possibilità aperte da questo ambiente operativo sono molte, in primis l'accesso alla programmazione in Java del robot ed alle API leJOS NXJ, le quali comprendono:

- Thread di tipo Preemptive (tasks)
- Arrays anche multi-dimensionali
- Ricorsioni
- Sincronizzazione

<sup>4</sup>Chiamato così per il gioco di parole che avviene tra la pronuncia simile a LEGO e il significato della parola in lingua spagnola "distante"

<sup>5</sup>Java Virtual Machine

- Eccezioni
- I tipi di dato Java, molto utili per il supporto ai numeri reali Float e Stringhe
- L'accesso alle classi `java.lang`, `java.util` e `java.io`
- API ben documentate<sup>6</sup>

Purtroppo, essendo il codice prodotto in Java e non binario RXE, non risulta compatibile per l'uso con NXT Simulator.

### 2.3.3 Ruby-NXT

Ruby-NXT è un'interfaccia per Ruby che aggiunge il supporto al controllo del NXT via Bluetooth. La libreria è ancora nello stadio iniziale di sviluppo ma consente già un buon supporto al controllo sia a basso che alto livello del sistema.

Ruby è un linguaggio di scripting orientato agli oggetti, che possiede però caratteristiche tipiche dei paradigmi imperativo e funzionale<sup>7</sup>. Essendo un linguaggio che comunica via wireless con il robot, non è interessato alla simulazione tramite NXT Simulator.

### 2.3.4 NXT-Python

NXT-Python è un'interfaccia driver Python per NXT, utilizza la libreria `NXT_Python` (che ne costituisce la base) scritta da Douglas P Lau back nel 2007. Grazie a queste librerie è possibile dotare del supporto all'NXT il linguaggio Python, la comunicazione può avvenire sia tramite cavo che wireless (USB e Bluetooth).

Python è un linguaggio al alto livello interpretato e spesso viene classificato anche come linguaggio di scripting; si può definire molto potente soprattutto per la possibilità di utilizzarlo secondo vari paradigmi di programmazione (come ad esempio la programmazione strutturata, ad oggetti, o funzionale).

---

<sup>6</sup><http://lejos.sourceforge.net/nxt/pc/api/index.html>

<sup>7</sup>Inteso come stile di programmazione



# Capitolo 3

## NXT Simulator

### 3.1 Premessa

La creazione del simulatore è stata dettata dalla volontà di realizzare uno strumento che, a disposizione di utenti e sviluppatori dell'ambiente Mindstorm<sup>®</sup> NXT, desse la possibilità di studiare il comportamento del robot senza un accesso fisico allo stesso. Il tipo di simulazione è del tipo azione-reazione, cioè il programma dà la possibilità di caricare file eseguibili ed osservarne l'evoluzione (anche tramite interazioni con lo stesso, es. motori in modalità programmata o manuale, sensori con parametri variabili). Tutto questo si sviluppa su un'interfaccia bidimensionale tramite la visualizzazione di immagini e controlli che rappresentano le varie parti del kit NXT collegate al Brick. L'obiettivo ultimo è quello di poter creare un simulatore tridimensionale che possa visualizzare la costruzione ed il movimento del robot in un ambiente grafico 3D, ma ciò sarà materia di sviluppo per successive evoluzioni.

NXT Simulator si basa su varie componenti:

- un analizzatore di file RXE, che gestisce quella che si potrebbe definire la "macchina virtuale" poiché si fa carico di: gestire le strutture dati e la struttura del file, l'interpretazione delle istruzioni ed il loro effetto sulle variabili o per la comunicazione di input/output;
- un'interfaccia grafica che permetta di interagire con il sistema per: la scelta del file RXE, la connessione dei motori e sensori (virtuali), l'avvio o l'arresto della simulazione, la scelta della localizzazione, l'interazione con le componenti di input;
- classi speciali per l'intercomunicazione delle suddette parti, per fornire supporto all'interprete dei comandi, per la conversione tra i vari tipi di variabili, per la sincronizzazione dei motori ed il sistema di logging integrato, per l'integrazione del sistema di localizzazione a più lingue per l'interfaccia.

Tutte queste parti non saranno descritte in dettaglio poiché sono state ampiamente sviluppate e descritte durante la loro realizzazione nelle tesi precedenti,

saranno però approfonditi quegli aspetti che sono stati interessati da modifiche, correzioni e nuove implementazioni.

## 3.2 Guida all'utilizzo dell'ambiente simulato

I requisiti dell'applicazione sono i seguenti:

- Sistema operativo Windows (versione minima: XP), Mac OS X 10.5, Linux
- Java Runtime Environment 6 o superiore (per lo sviluppo Java Development Kit 6)<sup>1</sup>

I file RXE da simulare vanno prodotti come spiegato nella sezione 2.2. Se NXT Simulator non si avvia con l'interfaccia in lingua desiderata è necessario modificare l'opzione che si trova nel menù "Lingua" o "Language".

### 3.2.1 L'interfaccia

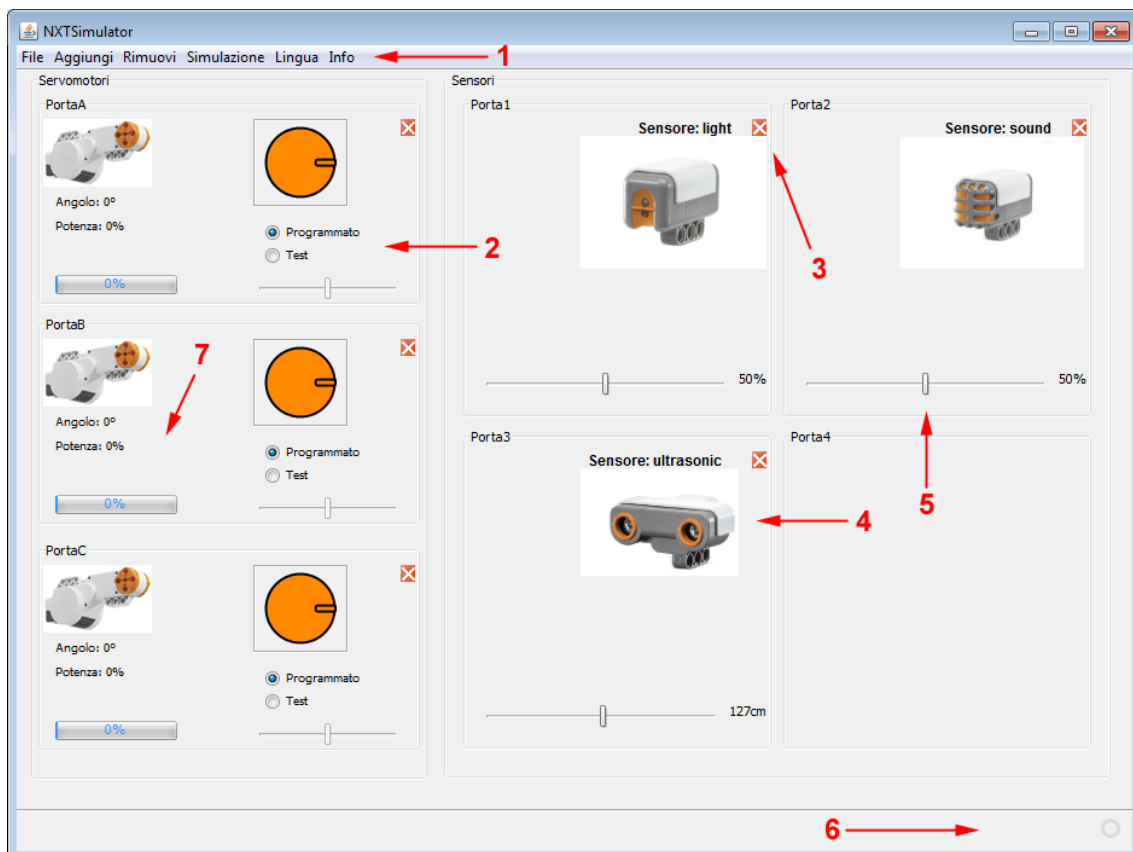


Figura 3.1: Finestra principale di NXT Simulator.

<sup>1</sup><http://java.sun.com/javase/downloads/index.jsp>



Le parti principali dell'interfaccia grafica sono:

1. Menù contestuali;
2. Pannello per il controllo dei servomotori;
3. Pannello per il controllo dei sensori, in particolare per rimuovere il sensore;
4. Immagine che indica il tipo di sensore inserito;
5. Slider utilizzato per il controllo manuale del sensore;
6. Zona per l'etichetta che indica lo stato di funzionamento del simulatore (es. simulazione attiva, terminata, ecc...);
7. Informazioni in tempo reale sul servomotore attivato alla relativa porta.

### Menù e pannelli

L'accesso ai menù avviene, come di consueto, tramite l'apposita barra situata nella parte superiore della finestra; l'uso degli stessi è altamente intuitivo poiché i nomi individuano esattamente la funzione svolta.

Dal menù file (figura 3.3(a)) è possibile:

- caricare il file RXE da utilizzare: l'effetto è quello di ottenere una finestra che consente di navigare il proprio file system per la scelta del file;

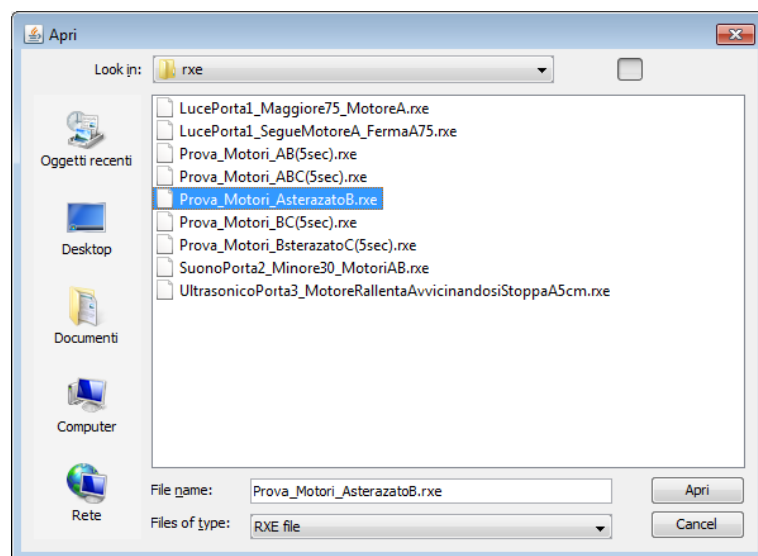
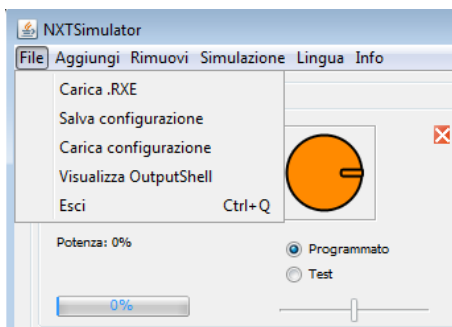


Figura 3.2: Finestra per la scelta del file RXE.

- salvare la configurazione dei pannelli attivi nel simulatore: in un file vengono automaticamente memorizzate le informazioni utili al recupero della posizione dei servomotori e sensori come sono visualizzati al momento del salvataggio;
- caricare la configurazione precedentemente salvata (è possibile salvare/caricare una sola configurazione alla volta);
- il sottomenù VisualizzaOutputShell apre una finestra che mostra l'output dell'interprete dei comandi;
- uscire dal programma.



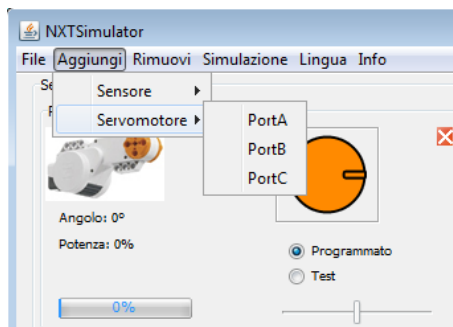
(a) Menù file



(b) Menù aggiungi sensore



(c) Pannello scelta della porta



(d) Menù aggiungi servomotore

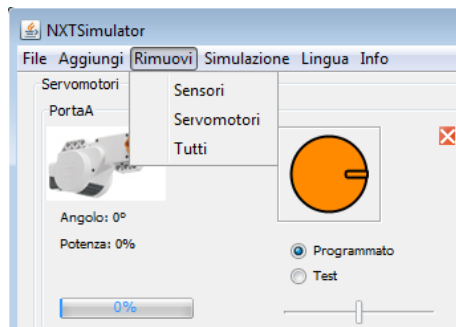
Figura 3.3: Menù File e Aggiungi.

Il menù "aggiungi" è diviso in due parti fondamentali: sensori (figura 3.3(b)) e servomotori (figura 3.3(d)).

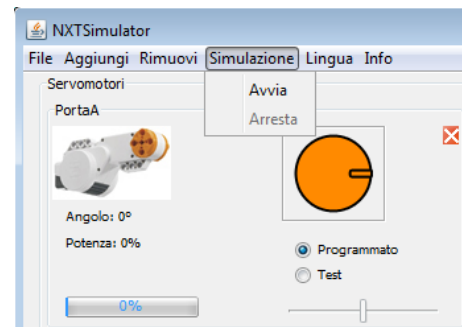
Essendo i sensori di vario tipo, il secondo menù a cascata del sottomenù "Sensore" dà la possibilità di scegliere quale componente collegare al Brick virtuale; una volta effettuata la scelta è possibile selezionare, tramite il pannello 3.3(c), la porta nella quale sarà simulato.

D'altra parte i servomotori sono di un solo tipo ed è quindi possibile scegliere direttamente dal menù a cascata in quale porta collegare il servomotore.

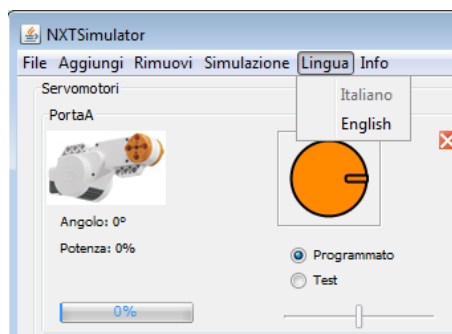
Ogni componente è rimovibile sia tramite il menù 3.4(a), che tramite l'apposito bottone rosso presente in ogni pannello con un componente attivo. Una volta che il file è caricato è possibile utilizzare il menù "Simulazione" 3.4(b) per avviare o arrestare la stessa, le opzioni sono attivate/disattivate a seconda dello stato della simulazione. Dal menù Lingua è possibile selezionare quale localizzazione utilizzare per l'uso dell'interfaccia: l'elenco mostra tutte le possibili scelte.



(a) Menù rimuovi



(b) Menù simulazione



(c) Menù lingua

Figura 3.4: Menù Rimuovi, Simulazione e Lingua.

Note di versione e informazioni sugli autori sono raggiungibili dal menù Info.



### 3.2.2 Guida

#### Interazioni utente

Ogni menù si attiva/disattiva a seconda dello stato della simulazione. Nota per l'uso è quella di impostare con precisione le porte ed il tipo di sensori e servomotori che verranno utilizzati. Infatti il simulatore non è in grado di correggere eventuali inesattezze della configurazione, ed è possibile che si verifichino anomalie di funzionamento.

Questa problematica non è correggibile dato che la posizione (e tipo) di eventuali motori o sensori utilizzati nei file RXE non è determinabile a priori poiché la configurazione degli stessi avviene tramite particolari istruzioni eseguite a runtime. D'altra parte questo può accadere anche con il robot reale se si collegassero alle porte motori e sensori in modo non corretto.

Una normale sequenza d'uso è così sviluppata:

1. Tramite menù "File"  $\Rightarrow$  Carica .RXE, sfogliare il contenuto del file system ed aprire il file designato alla simulazione.
2. Inserire nelle porte corrette i servomotori ed i sensori tramite il menù "Aggiungi"<sup>2</sup>.
3. Se vi sono parametri da impostare prima di avviare la simulazione è necessario farlo utilizzando gli appositi slider (punto 5 della figura 4.1).
4. Avviare la simulazione tramite il menù "Simulazione"  $\Rightarrow$  Avvia.
5. Durante lo svolgimento è possibile utilizzare gli slider dei sensori (o dei motori in modalità test), oppure arrestare manualmente la simulazione dal menù "Simulazione"  $\Rightarrow$  Arresta.<sup>3</sup>
6. Al termine della simulazione, ripetere dal punto 3. (o dal punto 1. se è necessario cambiare RXE) i passaggi elencati o terminare il programma.

#### Esempio pratico d'uso

Il file RXE da utilizzare nel simulatore è stato creato utilizzando il linguaggio NXC, il sorgente è mostrato di seguito:

Codice 3.1: UltrasonicoPorta3\_MotoreRallentaAvvicinandosiStoppaA5cm.nxc

```
task main(){
    SetSensorType(2, IN_TYPE_REFLECTION); // Ultrasonico porta 3
    int val = SensorScaled(2);
    OnFwd(0, 75); // Avvio del motore porta A potenza 75
    while(val > 5){
        val = SensorScaled(2); // Lettura valore di distanza
```

<sup>2</sup>I punti 1. e 2. possono anche essere invertiti.

<sup>3</sup>Al termine della simulazione vengono resettati i parametri di ogni componente attivo.

```

    if(val<50){ // Se sotto i 50 cm si rallenta
        int a = val*1666/1000; // Calcolo nuova
            potenza motore
        SetOutput(0, 2, a); // Imposto nuova potenza
    }
    else SetOutput(0, 2, 75); // Se oltre i 50 cm
        imposto potenza massima a 75
    } // Sotto i 5 cm il programma termina
}

```

Il listato presenta un programma che avvia il motore della porta A ed utilizza il sensore Ultrasonico per capire quando far fermare il robot adoperando un ciclo per poter rallentare la corsa man mano che la distanza diminuisce.

Una volta salvato il sorgente nel file *"UltrasonicoPorta3\_MotoreRallenta AvvicinandosiStoppaA5cm.nxc"*, si utilizza il prompt dei comandi (a seconda del sistema operativo il nome varia, per Windows è cmd.exe, per Mac OS X è Terminale e per Linux è shell o bash) e si utilizza il comando:

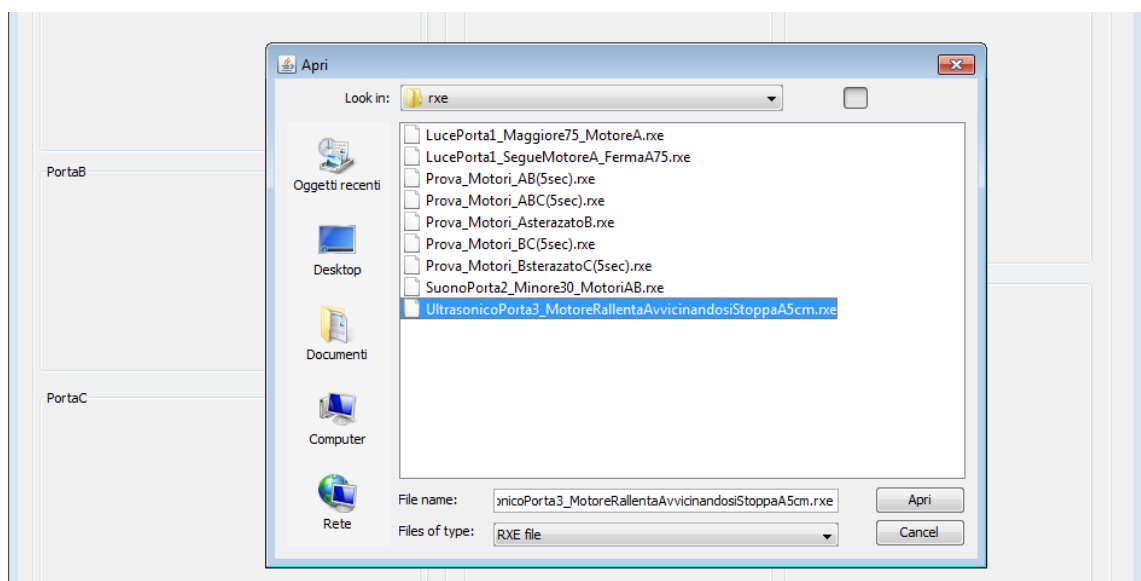
```

"nbc.exe -O=UltrasonicoPorta3_MotoreRallentaAvvicinandosiStoppaA5cm.rxe
UltrasonicoPorta3_MotoreRallentaAvvicinandosiStoppaA5cm.nxc"

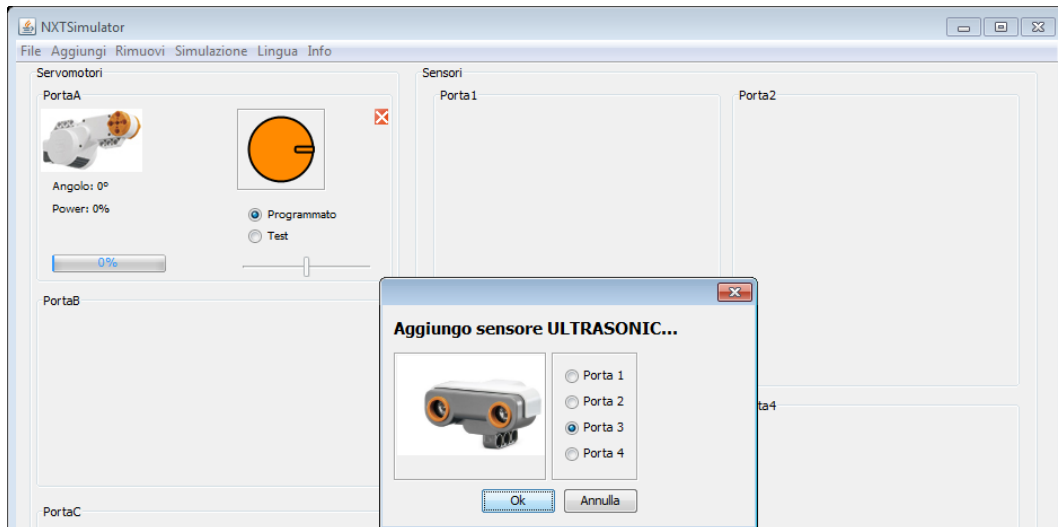
```

Ora si può procedere avviando il simulatore e seguendo i passi descritti precedentemente:

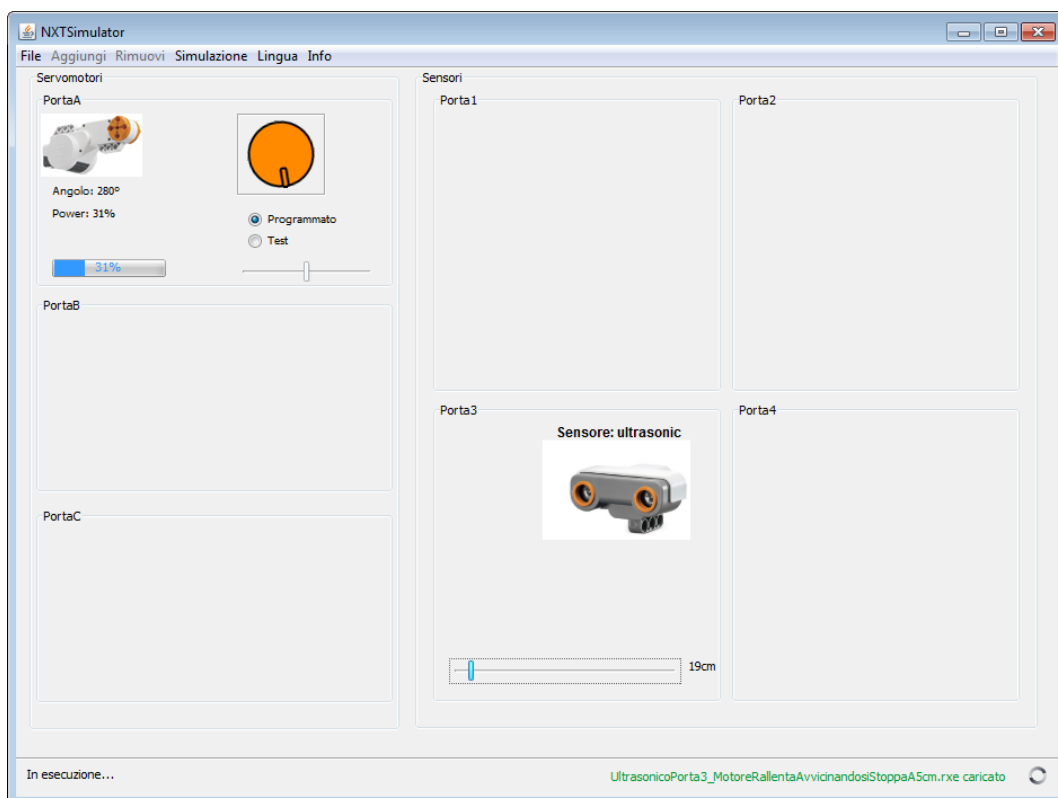
1. Tramite menù "File" ⇒ Carica .RXE, sfogliare il contenuto del file system ed aprire il file *"UltrasonicoPorta3\_MotoreRallenta AvvicinandosiStoppaA5cm.nxc"*



2. Inserire nella porta A un motore ed il sensore ultrasonico nella porta 3:



3. Utilizzare lo slider del sensore ultrasonico per impostare la distanza iniziale desiderata;
4. Avviare la simulazione tramite il menù "Simulazione" ⇒ Avvia;
5. Durante lo svolgimento utilizzare lo slider per vedere il comportamento del motore in funzione del cambio di distanza;



6. Si noti come, impostando un valore di prossimità alto come in figura 3.5(a), l'immagine del motore ruoti più velocemente (e la barra della potenza applicata sia più grande), mentre impostando un valore più basso come in figura 3.5(b), l'immagine ruoti più lentamente (con conseguente valore più basso di potenza applicata).

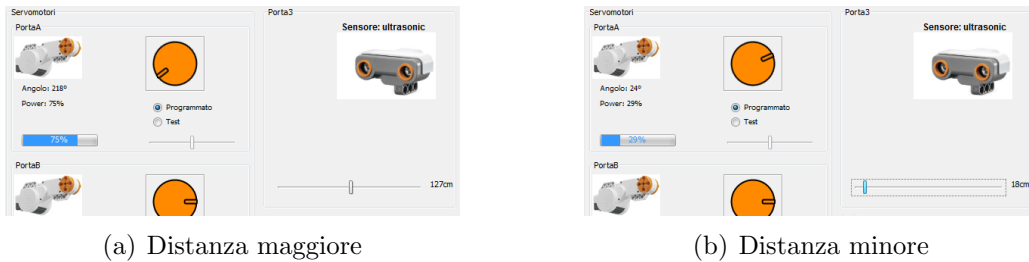


Figura 3.5: Comportamento del simulatore al variare del valore di prossimità.

7. Terminare il programma portando lo slider in un punto  $\leq$  a 5 cm o tramite il menù "Simulazione"  $\Rightarrow$  Arresta.
8. Al termine della simulazione, ripetere dal punto 3. (o dal punto 1. se è necessario cambiare RXE) i passaggi elencati o terminare il programma.

## 3.3 Manuale tecnico

In questa sezione saranno analizzate le problematiche del software risolte e strumenti e tecniche utilizzate al fine di portare il simulatore ad uno stato di completa correttezza e completezza.

### 3.3.1 Bug segnalati e limitazioni preesistenti

#### Introduzione

Per poter analizzare la struttura del simulatore è stato necessario uno studio approfondito della composizione del file RXE e, più in generale, del comportamento della virtual machine<sup>4</sup> all'interno del Brick.

Prerequisito fondamentale per la lettura di questa sezione è la conoscenza di semplici concetti di programmazione e teoria degli elaboratori.

Il file RXE è diviso in quattro parti principali, come mostrato in tabella 3.1.

Le varie parti del file sono esplicitamente descritte nel documento *Executable File Specification* [4] e da precedenti progetti [7] [8], ma è necessario sviscerare alcuni concetti necessari per questo elaborato.

<sup>4</sup>Abbreviata in VM d'ora in poi.

Segmento	Dimensione	Tipo
Header	38 Byte	Descrive gli altri segmenti del file.
Dataspace	variabile	Descrive le variabili e le strutture dati, nello specifico i tipi ed i valori di default.
Clump records	variabile	Informazioni riguardanti il momento in cui è prevista l'esecuzione di un Clump (vedi paragrafo Clump Records).
Codespace	variabile	Bytecode vero e proprio delle istruzioni.

Tabella 3.1: Struttura dei file RXE.

**Header** In questo segmento sono descritti gli offset utilizzati per determinare la dimensione variabile di quelli successivi, i quattro campi presenti sono:

Dimensione	Tipo	Descrizione
16 Byte	String	Questa stringa è caratteristica dei file RXE compilati per il firmware 1.03 e 1.05, l'esatto contenuto è la rappresentazione ASCII della stringa 'MindstormsNXT'.
18 Byte	Dataspace header	Descrive grandezza e disposizione dei dati all'interno del file. Questi dati possono essere statici o dinamici.
2 Byte	Clump count	Rappresentato in memoria da un Word da 16-bit senza segno, descrive quanti Clump sono presenti
2 Byte	Code word count	Rappresentato in memoria da un Word da 16-bit senza segno, descrive il numero di parole di codice contenute nel Codespace

Tabella 3.2: Struttura header del file RXE.

Il Dataspace header è un segmento complesso dell'header, inizia al sedicesimo Byte dall'inizio del file ed è rappresentato tramite Word da 16-bit senza segno, in particolare contiene:

- **Count:** il numero di campi presenti nella tabella DSTOC;
- **Initial Size:** grandezza in Byte del Dataspace completo;
- **Static Size:** grandezza in Byte dei soli dati statici del Dataspace;
- **Default Data Size:** grandezza in Byte dei valori di default (sia statici che dinamici);
- **Dynamic Default Offset:** offset in Byte dell'inizio dei valori di default dinamici;
- **Dynamic Default Size:** grandezza in Byte dei dati dinamici;



- **Memory Manager Head:** indice del primo elemento dell'array DVA;
- **Memory Manager Tail:** indice dell'ultimo elemento in coda del medesimo array;
- **Dope Vector Offset:** offset in Byte della locazione iniziale del DV relativo al Dataspace.

**Dataspace e Strutture dati** Questo segmento del file RXE contiene tutte le informazioni utili all'inizializzazione delle strutture del Dataspace in memoria durante l'attivazione del programma. Vi sono tre parti principali:

- tabella DSTOC: Dataspace Table Of Contents, contiene le informazioni riguardanti le variabili del programma, come ad esempio il tipo;
- Default Static Data: contiene i valori delle variabili statiche del programma (sono dati della DSTOC);
- Default Dynamic Data: contiene i valori dinamici delle variabili (sono anch'essi parte della DSTOC).

**Dataspace Table Of Contents** La tabella DSTOC<sup>5</sup> descrive i tipi di dati delle variabili e la loro locazione all'interno del programma. Si potrebbe meglio definire come il centro delle strutture dati della VM, infatti sono i suoi campi ad essere utilizzati come riferimento per le variabili da parte delle istruzioni. Se i dati possono variare durante il tempo di esecuzione, la struttura della DSTOC è invariabile.

La struttura dei campi della DSTOC è definita come da tabella 3.3:

DSTOC Record			
Campo	Tipo	Flags	Data Descriptor
Bit	0..7	8..15	16..31

Tabella 3.3: Struttura dei campi della DSTOC.

Gli 8 bit "Tipo", utilizzati per identificare il codice rappresentano un numero intero senza segno che identifica il tipo di variabile. Di seguito un elenco dei possibili tipi:

0. TC\_VOID: tipo di variabile non utilizzabile nel codice.
1. TC\_UBYTE: intero di 8bit, senza segno.
2. TC\_SBYTE: intero di 8bit, con segno.
3. TC\_UWORD: intero di 16bit, senza segno.

<sup>5</sup>Abbreviata solo in DSTOC d'ora in poi.

4. TC\_SWORD: intero di 16bit, con segno.
5. TC\_ULONG: intero di 32bit, senza segno.
6. TC\_SLONG: intero di 32bit, con segno.
7. TC\_ARRAY: Array, il tipo è definito più avanti.
8. TC\_CLUSTER: Struttura complessa composta da più campi.
9. TC\_MUTEX: Variabile utilizzata per lo scheduling dei Clump.

Gli ulteriori 8 bit del campo "Flags" indicano se la variabile è da istanziare utilizzando un valore di default: se è presente il valore "1" vuol dire che la variabile va istanziata con il valore zero, altrimenti, in caso di valore "0", deve essere istanziata utilizzando il valore di default presente nel segmento dati corrispondente.

Le istruzioni per manipolare i dati delle variabili devono riferirsi, attraverso un codice univoco, direttamente ad un indice della DSTOC, ma non tutti i dati possono essere riferiti direttamente (ad esempio per gli elementi degli array è necessario prima puntare al dato TC\_ARRAY e successivamente all'elemento desiderato).

Il campo Data Descriptor, infine, assume scopi diversi a seconda del tipo di dato della variabile.

La tabella DSTOC è compilata seguendo determinate specifiche a seconda del tipo di dato. Quando è dichiarata una struttura complessa, quali gli array ed i cluster, questa segue sempre la regola secondo la quale il tipo della struttura è il primo elemento, seguito poi dai tipi degli elementi che la compongono. Più rigorosamente:

- Una variabile semplice occupa una riga di DSTOC.
- Una variabile di tipo array occupa da due a più righe secondo questa disposizione: la prima riga, di tipo TC\_ARRAY, descrive l'indice del Dope Vector a cui si riferisce l'array, la riga successiva può essere o di tipo semplice o nuovamente di tipo array. Nel caso di array di array la regola si applica nuovamente costruendo così array multi-dimensionali.
- Una variabile di tipo cluster occupa anch'essa da due a più righe secondo la disposizione: il campo Data Descriptor della prima riga indica di quanti elementi è composto il cluster, le variabili successive specificano i tipi dei campi dello stesso. Tutti i tipi sono ammessi, ivi inclusi anche array o cluster stessi.

Nota: gli offset contenuti nei Data Descriptor dei tipi semplici o degli array indicano entrambi degli offset ma da posizioni differenti: quelli specificati dai tipi semplici isolati o array sono riferiti al "dataspace offset", mentre quelli degli

elementi successivi ad un array sono riferiti al "array data offset". Ciò è dovuto al fatto che gli array sono situati in posizioni dinamiche nella memoria, quindi utilizzare un sistema di offset risulta la scelta migliore, sia per indicizzare gli array stessi che per i loro elementi.

**Dati statici e dinamici** I valori di default sono accodati al precedente segmento, in particolare prima vi sono i dati statici. La dimensione occupata da ogni dato dipende dal tipo della variabile, il loro ordine dipende da quello della DSTOC e la loro presenza dipende dal valore del campo "Flag". La dimensione totale di questi dati viene ricavata dai campi dell'header: Default Data Size e Dynamic Defalut Offset.

I dati dinamici rappresentano unicamente array ed i loro valori. Il memory manager (nella VM del Brick) si avvale di una struttura di supporto per la gestione di questi valori, denominata **Dope Vector**<sup>6</sup>. Ad ogni array viene associato un DV, costituito da un record di cinque campi:

Campi	Descrizione
Offset	Riferito all'inizio dei dati nella memoria, è relativo all'inizio dei dati utente.
Dimensione	Grandezza in Byte degli elementi dell'array.
Numero elementi	Numero di elementi dell'array.
Back pointer	Non usato nel firmware 1.03.
Link index	Indice riferito al DV successivo.

Tabella 3.4: Struttura di un Dope Vector.

Tutti i DV sono memorizzati in una struttura chiamata **Dope Vector Array**<sup>7</sup> con le seguenti caratteristiche: è un oggetto singolo e quindi ne esiste una sola copia in memoria, il primo DV del DVA descrive il DVA stesso, non è manipolabile dalle istruzioni ma soltanto dal memory manager, è rappresentata da una *linked list* (utilizzando i campi *Link Index* dei DV).

**Clump records** Prima di analizzare questo segmento è necessario definire il concetto di **Clump**. Ogni Clump rappresenta una porzione di codice e, se chiamato direttamente da un altro Clump, è definibile come una *subroutine*. I Clump sono utilizzabili per l'ambiente multitasking. La VM del Brick prende le decisioni sulla pianificazione dei vari Clump durante il tempo di esecuzione, basate sulle informazioni contenute nei Clump Records. Un Clump si definisce dipendente quando deve essere eseguito al termine di un altro Clump. Non essendo la VM una macchina dotata di Stack non è possibile applicare la rientranza: quindi non sono ammesse *subroutine* ricorsive e, per *subroutine* comuni è necessario utilizzare

<sup>6</sup>Abbreviata solo in DV d'ora in poi.

<sup>7</sup>Abbreviata solo in DVA d'ora in poi.

i mutex per evitare le interferenze.

Tabella 3.5: Struttura di un Clump Record.

Campo	Dimensione	Descrizione
Fire Count	1 Byte	Byte senza segno, indica quando il Clump è pronto ad essere eseguito.
Dependent Count	1 Byte	Byte senza segno, indica il numero di Clump dipendenti da questo Clump.
Code Start Offset	2 Byte	Word senza segno, indica l'offset dal Codespace rappresentante l'inizio del Clump.
Dependent List	Variabile	Array di Byte senza segno, puntano agli indici dei Clump record che dipendono da questo.

Il numero di Clump è definito nell'header. Ogni Clump record ha una lunghezza fissa di 4 Byte ed una parte variabile (la lista delle dipendenze, tab. 3.5), quindi  $n$  Clump record hanno una parte fissa formata da  $4 * n$  Byte ed una parte variabile che deriva dalla quantità di dipendenze.

Il massimo ammesso dal firmware 1.03 (e 1.05) è di 255 Clump per programma.

**Istruzioni** La VM interpreta le istruzioni presenti nel file RXE, in accordo con le regole di interpretazione delle stesse vengono manipolate le strutture dati nella memoria e comandate le periferiche di I/O.

Le istruzioni disponibili sono di vario tipo:

- Matematiche: eseguono operazioni matematiche sugli oggetti del dataspace.
- Logiche: eseguono operazioni di logica binaria sugli oggetti del dataspace.
- Di confronto: confrontano i valori del dataspace fra di loro, oppure con lo zero, e producono risultati booleani.
- Manipolazione dati: muovono e manipolano gli oggetti del dataspace.
- Controllo di flusso: modificano la pianificazione dei Clump e delle istruzioni all'interno degli stessi<sup>8</sup>.
- Di sistema e I/O: interagiscono con le periferiche di I/O del Brick o altri servizi di sistema.

---

<sup>8</sup>Approfondita in Appendice A.

**Codespace** È l'ultimo segmento del file RXE, contiene le istruzioni che la VM deve eseguire. È composto da Word da 16 Bit<sup>9</sup>, i quali vengono interpretati come istruzioni (che possono anche essere di lunghezza variabile da uno a più codeword). Le istruzioni possono essere codificate in due modi: *short instruction* e *long instruction*. La codifica dipende da un flag posto al Bit 12 del Codeword (se uguale a 1 allora è short inst., altrimenti è long inst.).

La codifica ***long instruction*** è la più usata: infatti molte istruzioni supportano solo questa. Il primo codeword contiene il codice dell'istruzione, la dimensione totale dell'istruzione e alcuni flag (riservati ad alcune istruzioni). Successivi al primo codeword vi sono quelli che rappresentano gli operandi, i quali possono essere valori immediati (cioè codificati nel codeword) o puntatori alla DSTOC.

La codifica ***short instruction*** è riservata ad alcune istruzioni (le più comuni, risparmiando così spazio di memoria). L'istruzione può venire interpretata in due modi: se possiede un solo parametro, rappresentabile con un solo Byte o se, possedendo due parametri, è possibile determinare il primo par. grazie ad un offset (rispetto al secondo) di dimensione massima di un Byte.

---

<sup>9</sup>Definite Codeword, parole di codice.

### Correzioni a Interpreter.java

**Premessa** Le seguenti correzioni sono state applicate al simulatore seguendo le specifiche del documento *Executable File Specification* [4]. La precedente situazione era costituita da uno stato di imprecisione per la maggior parte delle istruzioni e inadeguatezza per l'interazione con i motori. Le convenzioni utilizzate nel codice sono: la tabella DSTOC, rappresentata dalla variabile "table" e i DV, divisi per tipo di array che rappresentano:

- per gli array di cluster viene utilizzata la matrice "arrayCluster" definita come segue:

```
arrayCluster[CLA][P].get(CL)
```

I due indici CLA e P puntano ad una "cella" della matrice, la quale è costruita per colonne: in ogni colonna vi sono gli identificatori ed i valori dei Cluster come spiegato di seguito in figura 3.6

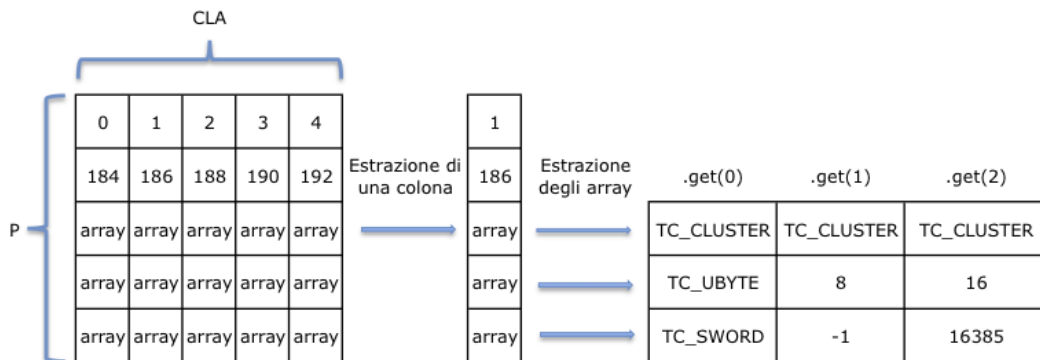


Figura 3.6: Disposizione degli Array di Cluster.

CLA quindi, è un indice che rappresenta la posizione dell'array di cluster all'interno della matrice.

P rappresenta un indice che viene utilizzato per tre funzioni: al valore 0 è memorizzato l'indice dell'array nella matrice (valore ridondante, equivale a CLA), al valore 1 è memorizzato l'indice del DV, dal valore 2 in poi sono memorizzati degli array che contengono gli elementi dei cluster. L'elemento 0-esimo è riservato.

Per accedere agli elementi del i-esimo cluster si utilizza la funzione `.get(CL)`, dove CL rappresenta il CL-esimo cluster.

Ad esempio, un array di cluster con elementi di tipo byte senza segno e word con segno avrà nella cella `arrayCluster[CLA][2]` un array di elementi del tipo TC\_CLUSTER, nella `arrayCluster[CLA][3]` elementi di tipo TC\_UBYTE e nella `arrayCluster[CLA][4]` elementi di tipo TC\_SWORD. Per ottenere il 3° cluster sono necessarie le seguenti istruzioni:

```
arrayCluster[1][2].get(4); arrayCluster[1][3].get(4);
arrayCluster[1][4].get(4);
```

- per gli array di tipi semplici (non cluster) viene utilizzato l'array "array" definito come segue:

```
array[A].get(I)
```

L'indice A identifica l'array scelto, mentre l'indice I punta ai dati dell'array, come spiegato in figura 3.7

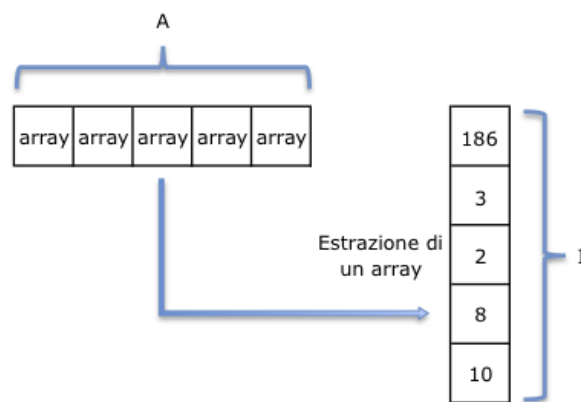


Figura 3.7: Disposizione degli Array di numeri all'interno della struttura "array".

A quindi, rappresenta la posizione dell'array all'interno della struttura.

I viene utilizzato per recuperare le seguenti informazioni: al valore 0 è memorizzato l'indice del DV, al valore 1 è memorizzato l'indice dell'array nella struttura (valore ridondante, equivale a A), dal valore 2 in poi vi sono gli elementi dell'array. Per ottenere, ad esempio 3 elementi di un array di byte senza segno sono necessarie le istruzioni:

```
array[3].get(2); array[3].get(3); array[3].get(4);
```

Di seguito saranno esposte le modifiche più rilevanti effettuate.

**OP\_SHORT\_MOV** Parametri: Destinazione, Sorgente.

Versione codificata in *short instruction* dell'istruzione MOV. Viene copiato il dato sorgente nella posizione specificata in destinazione.

Nella precedente versione veniva commesso un errore nella gestione degli array:

Codice 3.2: OP\_SHORT\_MOV errato

```
dest = calcolaIndexArray(array, arg1); // Indice destinazione
source2 = calcolaIndexArray(array, source1); // Indice sorgente
offset = array[source2].size();
for (int i = 2; i < offset; i++){
    System.out.println("Array "+ arg1+" <= element "+array[source2].
        get(i));
    outGen.println("Array "+ arg1+" <= element "+array[source2].get(
        i));
    outClump[clumpNow].println("Array "+ arg1+" <= element "+array[
        source2].get(i));
    // Funzione .add(int i, object o); inserisce l'oggetto "o" nella
        posizione "i" dell'array
    array[dest].add(i, array[source2].get(i));
}
```

L'array di destinazione infatti non veniva sovrascritto da una copia esatta, ma "mesccolato" con i dati dell'array sorgente. L'operazione risulterebbe valida solo nel caso in cui la destinazione avesse un numero di elementi minore o uguale alla sorgente, ma nel caso di più elementi, quelli oltre la quantità della sorgente sarebbero rimasti.

La soluzione proposta è di cancellare l'array destinazione (mantenendo le informazioni utili del DV) e copiarne successivamente il contenuto della sorgente.

Codice 3.3: OP\_SHORT\_MOV corretto

```
dest = calcolaIndexArray(array, arg1);
source2 = calcolaIndexArray(array, source1);
offset = array[source2].size();
ArrayList tempor = new ArrayList();
// Copia degli identificatori dell'array di destinazione
tempor.add(array[dest].get(0));
tempor.add(array[dest].get(1));
// Impostazione tipo e dati dell'array sorgente
for (int i = 2; i < offset; i++) {
    System.out.println("Array "+ arg1 + " <= element " + array[
        source2].get(i));
    outGen.println("Array "+ arg1 + " <= element " + array[source2
        ].get(i));
    outClump[clumpNow].println("Array "+ arg1 + " <= element " +
        array[source2].get(i));
    // Spostamento dei dati
    tempor.add(array[source2].get(i));
}
```



```

}
// Copia array temporaneo
array[dest] = tempor;

```

**OP\_MOV** Parametri: Destinazione, Sorgente.

Versione codificata in *long instruction* dell'istruzione MOV. Viene copiato il dato sorgente nella posizione specificata in destinazione.

Le correzioni apportate a questa istruzione sono le stesse della versione *short*, tenendo conto della diversa codifica dei parametri.

**OP\_INDEX** Parametri: Destinazione, Sorgente, Indice.

Copia l'elemento dell'array sorgente nella destinazione della DSTOC.

L'errore consiste nell'errata gestione dei casi in cui l'array è composto di cluster o altri array:

Codice 3.4: Caso array di cluster, OP\_INDEX errato

```

// Calcolo dell'indice che identifica l'array di cluster nella
// struttura di memoria ove sono memorizzati
source1 = calcolaIndexArrayCluster(arrayCluster, source1);
// Caso in cui l'array sorgente sia non vuoto
if(arrayCluster[source1][3].size()>1){
    for(int i = 0; i < table[dest].getDataDescriptor(); i++){
        System.out.println(arrayCluster[source1][i+2].get(
            index+1));
        tmp = (Integer)arrayCluster[source1][i+2].get(index
            +1);

        System.out.println("dest "+dest+"<-"+arrayCluster[
            source1][i+2].get(index+1));
        outGen.println("dest "+dest+"<-"+arrayCluster[
            source1][i+2].get(index+1));
        outClump[clumpNow].println("dest "+dest+"<-"+
            arrayCluster[source1][i+2].get(index+1));
        // Nella riga di destinazione nella DSTOC vengono
        // copiati i valori del cluster
        table[dest+i+1].setValDefault(tmp);
    }
}else{
    for(int i=0; i< table[dest].getDataDescriptor(); i++)
        table[dest+i+1].setValDefault(0);
}

```

Codice 3.5: Caso array di array, OP\_INDEX errato

```

// Calcolo dell'indice che identifica l'array nella struttura di
// memoria ove sono memorizzati
source1 = calcolaIndexArray(array, source1);

```

```
// Lettura del dato all'indice scelto
tmp = (Integer)array[source1].get(index+3);
System.out.println("source1 "+source1+"tmp "+tmp);
outGen.println("source1 "+source1+" tmp "+ tmp);
// Scrittura del dato nella riga di destinazione nella DSTOC
table[dest].setValDefault(tmp);
```

In queste situazioni viene omesso di gestire il caso in cui nei cluster o negli array semplici vi sono elementi del tipo "array": infatti nella definizione di DSTOC gli array non occupano una sola riga della tabella, ma due.

Per capire al meglio la situazione è necessario analizzare degli esempi in proposito.

Tabella 3.6: Esempio di cluster con un elementi di tipo array.

DSTOC			
Indice record	Tipo	Flags	Data Descriptor
168	TC_CLUSTER	0x00	0x0003
169	TC_UBYTE	0x01	0x01C2
170	TC_ARRAY	0x00	0x0202
171	TC_UBYTE	0x00	0x0001
172	TC_SBYTE	0x01	0x01C4

Nel caso del cluster rappresentato in tabella 3.6, il primo record, nel campo Data Descriptor, indica la quantità di elementi che lo compongono: 3; ma nella DSTOC vi sono 4 elementi. La spiegazione è data dal record 170 il quale logicamente rappresenta un unico elemento, ma è fisicamente composto dai record 170 e 171 assieme (il 171° indica il tipo di dati contenuti nell'array al 170° record).

È necessario quindi che il codice provveda a copiare il numero esatto di record, contando un record in più ogni qual volta si presenta un elemento di tipo array.

Tabella 3.7: Esempio di array con un elementi di tipo array.

DSTOC			
Indice record	Tipo	Flags	Data Descriptor
...	...	...	...
180	TC_ARRAY	0x00	0x0800
181	TC_ARRAY	0x00	0x0000
182	TC_UBYTE	0x00	0x0000
...	...	...	...

Simil caso quello degli array di array, in particolare l'esempio fornito dalla tabella 3.7 mostra come in questi casi, per copiare un array di array, non sia sufficiente il solo record che dovrebbe contenere il tipo (nell'esempio il 181), ma sia necessario copiare anche il record che completa l'array secondario (in questo caso il 182).

Di seguito il codice corretto:

Codice 3.6: Caso array di cluster, OP\_INDEX corretto

```
// Calcolo dell'indice che identifica l'array di cluster nella
// struttura di memoria ove sono memorizzati
source1 = calcolaIndexArrayCluster(arrayCluster, source1);
// Caso in cui l'array sorgente non sia vuoto, il controllo deve
// essere effettuato nel secondo campo
if (arrayCluster[source1][2].size() > 1) {
    // Copia del cluster, compresi eventuali array
    int ko = 0; // fattore di aggiustamento in caso di array
    for (int i = 0; i < table[dest].getDataDescriptor(); i++) {
        // Copia dell'elemento
        table[dest + 1 + i + ko] = (DSTOC) arrayCluster[source1][i +
            2 + ko].get(index + 1);
        if (table[dest + 1 + i + ko].getType().equals("TC_Array")) {
            ko++;
            // Copia del tipo dell'array
            table[dest + 1 + i + ko] = (DSTOC) arrayCluster[source1
                ][i + 2 + ko].get(index + 1);
        }
        System.out.println("dest " + dest + "<-" + arrayCluster[
            source1][i + 2 + ko].get(index + 1));
        outGen.println("dest " + dest + "<-" + arrayCluster[source1
            ][i + 2 + ko].get(index + 1));
        outClump[clumpNow].println("dest " + dest + "<-" +
            arrayCluster[source1][i + 2 + ko].get(index + 1));
    }
} else {
    int ko = 0;
    for (int i = 0; i < table[dest].getDataDescriptor(); i++) {
        // Copia dell'elemento
        table[dest + i + 1 + ko].setValDefault(0);
        if (table[dest + 1 + i + ko].getType().equals("TC_Array")) {
            ko++;
            // Copia del tipo dell'array
            table[dest + 1 + i + ko].setValDefault(0);
        }
    }
}
```

Codice 3.7: Caso array di array, OP\_INDEX corretto

```
// Caso array di array
if (table[dest].getType().equals("TC_Array")) {
    dest = calcolaIndexArray(array, dest);
    ArrayList tempor = new ArrayList();
    // Salvataggio intestazione array di destinazione
    for (int as = 0; as < 3; as++) {
        tempor.add(array[dest].get(as));
    }
}
```

```

    array[dest] = (ArrayList) array[source1].get(index + 3);

    // Ripristino dell'intestazione
    for (int as = 0; as < 3; as++) {
        array[dest].set(as, tempor.get(as));
    }
} else { // Caso array di tipi semplici
    tmp = (Integer) array[source1].get(index + 3);
    System.out.println("source1 " + source1 + "tmp " + tmp);
    outGen.println("source1 " + source1 + " tmp " + tmp);
    table[dest].setValDefault(tmp);
}

```

**OP\_REPLACE** Parametri: Destinazione, Sorgente, Indice, NuovoValore.

Rimpiazza un sottoinsieme dell'array sorgente a partire dall'indice index e memorizza il risultato nell'array destinazione. Nel caso in cui l'indice fornito sia fuori dal range massimo della sorgente viene effettuato un move dell'intero array.

Questa istruzione è stata interamente mal interpretata: infatti le azioni che venivano svolte erano quelle di sostituire il solo elemento puntato dall'indice nell'array sorgente e la copia errata sulla destinazione (commettendo lo stesso errore fatto in OP\_MOV).

Per brevità viene proposto il solo codice corretto:

Codice 3.8: Caso array di array, OP\_REPLACE corretto

```

// dest corrisponde all'array di destinazione
// source1 corrisponde all'array sorgente
// index corrisponde all'indice
// source2 corrisponde al nuovo valore
index = table[index].getValDefault();
if (table[index].getValDefault() == 65535) {
    index = 0;
}
// Caso array di cluster non era gestito! Ora si
if (table[dest + 1].getType().equals("TC_Cluster")) {
    dest = calcolaIndexArrayCluster(arrayCluster, dest);
    source1 = calcolaIndexArrayCluster(arrayCluster, source1);
    // Caso in cui Index e' nei limiti dell'array
    if (!(index >= arrayCluster[source1][2].size())) {
        // Carico il cluster
        int ko = 0;
        for (int j = 0; j < table[source2].getDataDescriptor(); j++) {
            // Caso in cui si voglia aggiungere un elemento a fondo dell'array
            if (index < arrayCluster[source1][2 + j + ko].size() - 1) {
                arrayCluster[dest][2 + j + ko].set(index + 1, table[source2 + 1
                    + j + ko]);
            } else {
                arrayCluster[dest][2 + j + ko].add(table[source2 + 1 + j + ko]);
            }
        }
        if (table[source2 + 1 + j + ko].getType().equals("TC_Array")) {
            ko++;
            if (index < arrayCluster[source1][2 + j + ko].size() - 1) {
                arrayCluster[dest][2 + j + ko].set(index + 1, table[source2
                    + 1 + j + ko]);
            }
        }
    }
}

```

```

        } else {
            arrayCluster[dest][2 + j + ko].add(table[source2 + 1 + j +
                ko]);
        }
    }
}
if (dest != source1) {
    for (int j = 2; j < maxCluster + 2; j++) {
        arrayCluster[dest][j] = arrayCluster[source1][j];
    }
}
} else { // Caso in cui index va oltre la dimensione dell'array
    for (int j = 2; j < maxCluster + 2; j++) {
        arrayCluster[dest][j] = arrayCluster[source1][j];
    }
}
} else {
    dest = calcolaIndexArray(array, dest);
    source1 = calcolaIndexArray(array, source1);
    // Index e' nei limiti dell'array
    if (index < array[source1].size() - 3) {
        // Caso array di numeri
        if (!table[source2].getType().equals("TC_Array")) {
            source2 = table[source2].getValDefault();
            array[source1].set(index + 3, source2);
        } else { // Caso array di array
            ArrayList sourceX = (ArrayList) array[calcolaIndexArray(array,
                source2)];
            array[source1].set(index + 3, sourceX);
        }
        ArrayList tempor = new ArrayList();
        // Salvataggio intestazione array di destinazione
        for (int as = 0; as < 3; as++) {
            tempor.add(array[dest].get(as));
        }
        // Copia dell'array sorgente
        for (int i = 3; i < array[source1].size(); i++) {
            tempor.add(array[source1].get(i));
        }
        // Scrittura nella destinazione
        array[dest] = tempor;
    } else { // Caso in cui index va oltre la dimensione dell'array
        ArrayList tempor = new ArrayList();
        // Salvataggio intestazione array di destinazione
        for (int as = 0; as < 3; as++) {
            tempor.add(array[dest].get(as));
        }
        for (int i = 3; i < array[source2].size(); i++) {
            tempor.add(array[source2].get(i));
        }
        array[dest] = tempor;
    }
}
}
}

```

**OP\_ARRSIZE** Parametri: Destinazione, Sorgente.

Memorizza il numero di elementi dell'array sorgente nella destinazione della DSTOC.

In questa istruzione sono stati utilizzati indici errati per effettuare il calcolo del numero di elementi nell'array di cluster. Il codice corretto è molto semplice e viene omissso.

**OP\_ARRBUILD** Parametri: DimensioneIstruzione, Destinazione, Sorgente1, Sorgente2, ... SorgenteN.

Costruisce l'array di destinazione utilizzando gli oggetti sorgente (possono essere di qualsiasi tipo) che saranno concatenati.

La versione precedente peccava di una sostanziale incompletezza: veniva infatti gestito soltanto un caso, quello di array di tipi semplici. Inoltre la dimensione dell'istruzione soffriva di problemi se l'array di destinazione era di generose dimensioni. L'istruzione è stata così riscritta comprendendo i casi in cui vi erano elementi del tipo:

- Array di cluster (compresa la gestione del caso in cui il cluster è composto anche da elementi di tipo array);
- Cluster;
- Array di tipi semplici;
- Array di array;
- Tipi semplici.

Codice 3.9: OP\_ARRBUILD corretto

```
// dest corrisponde all'array di destinazione
// size corrisponde alla dimensione dell'istruzione
// source0 rappresenta la sorgente (diversa per ogni parametro)

// Pulizia dell'array di destinazione
if (!table[dest + 1].getType().equals("TC_Cluster")) // La destinazione e' un
    Array
{ // La sorgente e' un dato/array
    ArrayList temp = new ArrayList();
    temp.add(array[calcolaIndexArray(array, dest)].get(0));
    temp.add(array[calcolaIndexArray(array, dest)].get(1));
    temp.add(array[calcolaIndexArray(array, dest)].get(2));

    array[calcolaIndexArray(array, dest)] = temp;
} else { // La destinazione e' un array di cluster
    for (int i = 0; i < maxCluster; i++) {
        Object tipo = arrayCluster[calcolaIndexArrayCluster(arrayCluster, dest)
            ][i + 2].get(0);
        arrayCluster[calcolaIndexArrayCluster(arrayCluster, dest)][i + 2].clear
            ();
        arrayCluster[calcolaIndexArrayCluster(arrayCluster, dest)][i + 2].add(
            tipo);
    }
}

// Controllo delle sorgenti: cluster, dati o array.
// Ci possono essere piu' d'una sorgente.
for (int i = 2; i < size - 4; i = i + 2) {
    coppia[0] = words[i];
    coppia[1] = words[i + 1];
    int source = conv.getPosInt(coppia);
    if (table[source].getType().equals("TC_Array")) { // E' un array
        if (table[source + 1].getType().equals("TC_Cluster")) { // Di cluster
            int dest0 = calcolaIndexArrayCluster(arrayCluster, dest);
            int source0 = calcolaIndexArrayCluster(arrayCluster, source);
            // Copia dei dati da un array all'altro
            for (int y = 0; y < maxCluster; y++) {
                for (int k = 1; k < arrayCluster[source0][2].size(); k++) {
                    arrayCluster[dest0][y + 2].add(arrayCluster[source0][y + 2].
                        get(k));
                }
            }
        }
    }
}
```

```

    }
  }
} else { // Array normale
  int dest0 = calcolaIndexArray(array, dest);
  int source0 = calcolaIndexArray(array, source);
  // Destinazione array di numeri
  if (!array[dest0].get(2).equals("TC_Array")) {
    // Copia dei dati da un array all'altro
    for (int j = 3; j < array[source0].size(); j++) {
      array[dest0].add(array[source0].get(j));
    }
  } else { // Destinazione array di array
    array[dest0].add(array[source0]);
  }
}
} else { // E' un dato
  if (table[source].getType().equals("TC_Cluster")) { // E' un cluster
    int dest1 = calcolaIndexArrayCluster(arrayCluster, dest);
    int ko = 0; // Fattore di spostamento nel caso di elementi di tipo
    array
    for (int x = 0; x < table[source].getDataDescriptor(); x++) {
      arrayCluster[dest1][x + 2 + ko].add(table[source + 1 + x + ko]);
      if (table[source + 1 + x + ko].getType().equals("TC_Array")) {
        ko++;
        arrayCluster[dest1][x + 2 + ko].add(table[source + 1 + x +
        ko]);
      }
    }
  } else { // E' un dato non cluster
    int dest1 = calcolaIndexArray(array, dest);
    array[dest1].add(table[source].getValDefault());
  }
}
}
}

```

**OP\_ARRSUBSET** Parametri: Destinazione, Sorgente, Indice, Contatore.

Memorizza una parte dell'array sorgente nell'array destinazione; vengo copiati  $n$  gli elementi (dove  $n$  = "Contatore", se "Contatore" non è un numero valido verranno copiati  $n = [( \text{lunghezza sorgente} ) - \text{Indice}]$  elementi).

In questa istruzione non venivano effettuati i controlli sul valore di "Contatore" ed inoltre non era contemplato il caso di array di cluster o di array di array. La copia dei valori da un array all'altro avveniva inoltre con l'errata modalità già vista per OP\_MOV. Il codice è stato quindi completamente riscritto.

Codice 3.10: OP\_ARRSUBSET corretto

```

// dest rappresenta la destinazione
// source2 rappresenta il contatore
// source1 rappresenta la sorgente
// index rappresenta l'indice
if (index == 65535) {
  index = 0;
} else {
  index = table[index].getValDefault();
}
if (!table[dest + 1].getType().equals("TC_Cluster")) { // Caso array
  non di cluster

```

```

// Segue il controllo per il calcolo di Contatore
if (source2 == 65535) {
    source2 = array[calcolaIndexArray(array, source1)].size() -
        index - 3;
} else {
    source2 = table[source2].getValDefault();
}
source1 = calcolaIndexArray(array, source1);
dest = calcolaIndexArray(array, dest);

ArrayList temp2 = new ArrayList();
temp2.add(array[dest].get(0));
temp2.add(array[dest].get(1));
temp2.add(array[dest].get(2));
array[dest] = temp2;

for (int i = index; i < (index + source2); i++) {
    array[dest].add(array[source1].get(i + 3));
}
} else { // Caso array di cluster
    // Segue il controllo per il calcolo di Contatore
    if (source2 == 65535) {
        source2 = arrayCluster[calcolaIndexArrayCluster(arrayCluster,
            source1)][2].size() - index - 2;
    } else {
        source2 = table[source2].getValDefault();
    }

    source1 = calcolaIndexArrayCluster(arrayCluster, source1);
    dest = calcolaIndexArrayCluster(arrayCluster, dest);

    // Copia degli elementi dall'array sorgente
    for (int j = 0; j < maxCluster; j++) {
        arrayCluster[dest][j + 2].clear(); // Svuotamento array di
            destinazione
        arrayCluster[dest][j + 2].add(arrayCluster[source1][j + 2].
            get(0)); // Copia del tipo di dato
        for (int i = index; i < (index + source2); i++) {
            arrayCluster[dest][j + 2].add(arrayCluster[source1][j +
                2].get(i + 2)); // Copia dei cluster
        }
    }
}
}

```

**OP\_ARRINIT** Parametri: Destinazione, NuovoValore, Contatore.

Inizializza l'array sorgente con  $n$  copie (dove  $n$  = "Contatore") di NuovoValore. Nel caso in cui "Contatore" è = 0 o nel caso non sia un numero valido, l'array di destinazione sarà vuoto.

Nella versione precedente vi era il problema che nel caso in cui "Contatore" non fosse stato valido, l'interprete non avrebbe fatto avanzare il puntatore



program counter, cadendo in loop infinito. Inoltre non erano gestiti gli array di cluster.

Codice 3.11: OP\_ARRINIT corretto

```
// dest rappresenta la destinazione
// source1 rappresenta il nuovo valore
// source2 rappresenta il contatore
if ((table[source2].getValDefault() == 65535) || source2 == 65535) {
    source2 = 0;
} else {
    source2 = table[source2].getValDefault();
}
// Gestito ora anche il caso di Array di Cluster
if (!table[source1].getType().equals("TC_Cluster")) {
    dest = calcolaIndexArray(array, dest);
    ArrayList tempi = new ArrayList();

    // Copia degli identificatori dell'array di destinazione
    tempi.add(array[dest].get(0));
    tempi.add(array[dest].get(1));
    tempi.add(array[dest].get(2));
    array[dest] = tempi;

    // Ciclo per l'inserimento dei valori
    for (int i = 0; i < source2; i++) {
        if (!array[dest].get(2).equals("TC_Array")) {
            array[dest].add(table[source1].getValDefault());
        } else {
            array[dest].add(array[calcolaIndexArray(array, source1)]);
        }
    }
} else {
    dest = calcolaIndexArrayCluster(arrayCluster, dest);
    // Fattore di aggiustamento nel caso vi siano valori di tipo array
    int ko = 0;
    for (int i = 0; i < table[source1].getDataDescriptor(); i++) {
        Object zero = arrayCluster[dest][i + 2 + ko].get(0);
        arrayCluster[dest][i + 2 + ko].clear();
        arrayCluster[dest][i + 2 + ko].add(zero);
        for (int j = 0; j < source2; j++) {
            arrayCluster[dest][i + 2 + ko].add(table[source1 + 1 + i + ko]);
            if (table[source1 + 1 + i + ko].getType().equals("TC_Array")) {
                ko++;
                arrayCluster[dest][i + 2 + ko].add(table[source1 + 1 + i + ko]);
            }
        }
    }
}
```

**OP\_NUMTOSTRING** Parametri: Destinazione, Sorgente.

Converte l'intero situato nella posizione specificata dalla sorgente in una stringa contenente il numero stesso, memorizza il risultato in destinazione.

Questa istruzione non era implementata nella precedente versione del simulatore. La convenzione, utilizzata dal firmware 1.03 per la formattazione delle stringhe, è quella dello standard ANSI C. Vengono memorizzate come sequenze di caratteri in array di interi utilizzando la codifica ASCII; ogni stringa viene conclusa dal carattere terminatore denominato *NULL* (equivalente all'intero "0").

Codice 3.12: OP\_NUMTOSTRING

```
// dest rappresenta l'array destinazione
// source1 rappresenta la sorgente

dest = calcolaIndexArray(array, dest);
ArrayList tempi = new ArrayList();
// Copia degli identificatori dell'array di destinazione
tempi.add(array[dest].get(0));
tempi.add(array[dest].get(1));
tempi.add(array[dest].get(2));
array[dest] = tempi;

// Conversione tramite libreria standard Java del valore sorgente
String toConvert = String.valueOf((int) table[source1].getValDefault());
// Creazione array
for (int i = 0; i < toConvert.length(); i++) {
    array[dest].add(toConvert.getBytes()[i]);
}
// Inserimento del carattere di terminazione
array[dest].add((byte) 0);
```

### Istruzioni per l'interazione con le periferiche esterne

Le correzioni effettuate a questo tipo di istruzioni non riguardano un'errata precedente implementazione, ma un adattamento a: la nuova struttura dati, per quanto riguarda l'interazione con i sensori, ed alla rivisitazione della classe, per il controllo dei motori.

Le classi *InputPortConfigurationProperties* ed *OutputPortConfigurationProperties* sono atte all'interfacciamento fra l'interprete dei comandi e le strutture dati di input e di output per l'accesso in scrittura delle proprietà dei dispositivi. Tramite le informazioni passate dalle istruzioni, esse convertono i dati (se necessario) ed accedono alle proprietà da modificare. Il comportamento di queste classi

è fedele alla descrizione che si trova nel documento Executable File Specification di LEGO [4].

**OP\_SETIN** Parametri: Sorgente, Porta, IDProprietà.

Imposta una proprietà identificata da "IDProprietà" del sensore inserito nella porta "Porta".

Codice 3.13: OP\_SETIN

```
// dest rappresenta Porta
// source1 rappresenta la sorgente
// source2 rappresenta l'ID della proprietà
in = new InputPortConfigurationProperties(table[source1].
    getValDefault(), table[dest].getValDefault(), source2, clumpNow);
in.device();
```

La classe *InputPortConfigurationProperties* verrà descritta nella sezione 3.3.2<sup>10</sup>.

**OP\_GETIN** Parametri: Destinazione, Porta, IDProprietà.

Memorizza in destinazione la proprietà identificata da "IDProprietà" del sensore inserito nella porta "Porta".

Codice 3.14: OP\_GETIN

```
// dest rappresenta destinazione
// source1 rappresenta la porta
// source2 rappresenta l'ID della proprietà
table[dest].setValDefault(NXTSView.getSensorController(table[source1]
    .getValDefault()).getValue(source2));
```

Per la scrittura di questa istruzione e l'implementazione del controllo dei sensori, è stata necessaria la costruzione di una nuova struttura dati denominata **SensorData**. Essa racchiude tutte le proprietà e impostazioni che rappresentano lo stato dei sensori secondo le specifiche del firmware 1.03.

Questa classe è utilizzata sia dall'interprete dei comandi sia dai pannelli dedicati al controllo dei sensori da parte dell'utente. Ogni istanza rappresenta un sensore e ve ne è uno per porta: *s1*, *s2*, *s3*, *s4* riferiti alle porte da 0 a 3 secondo la convenzione del firmware.

<sup>10</sup>Per dettagli vedere il documento [4] alla bibliografia ed il codice sorgente di NXTSimulator.

Valore ID	ID Proprietà	Tipo
0x0	IO_IN_TYPE	TC_UBYTE
0x1	IO_IN_MODE	TC_UBYTE
0x2	IO_IN_ADRAW	TC_UWORD
0x3	IO_IN_NORMRAW	TC_UWORD
0x4	IO_IN_SCALED_VAL	TC_SWORD
0x5	IO_IN_INVALID_DATA	TC_UBYTE

Tabella 3.8: Elenco proprietà dei sensori.

La classe `SensorData` rispetta il tipo e gli identificatori delle proprietà elencate in tabella 3.8 e verrà descritta nella sezione 3.3.2<sup>11</sup>.

**OP\_SETOUT** Parametri: Dimensione, Porta/ListaPorte, IDProp1, Sorgente1, ... IDPropN, SorgenteN.

Imposta una o più proprietà, identificate da "IDPropX", del servomotore/i inserito/i nella porta "Porta" (o set di porte "ListaPorte"). Il parametro "Dimensione" specifica la lunghezza totale in Byte dell'istruzione; nel caso in cui venga specificata una sola porta, il valore è un Byte, altrimenti è un puntatore ad un array contenente i numeri di porta.

L'istruzione è stata modificata per poter adattarsi ai cambiamenti apportati ad *OutputPortConfigurationProperties*<sup>12</sup>.

Codice 3.15: OP\_SETOUT

```
// size rappresenta la dimensione
// words rappresenta i parametri passati
words = new byte[size - 4];
for (int i = 0; i < size - 4; i++) {
    // scrittura dei parametri nell'array
    words[i] = lett[pc + i + 4];
}
out = new OutputPortConfigurationProperties(words, table, array,
    size, clumpNow);
out.motor();
```

**OP\_GETOUT** Parametri: Destinazione, Porta, IDProprietà.

Memorizza in destinazione la proprietà identificata da "IDProprietà" del servomotore inserito nella porta "Porta".

Codice 3.16: OP\_GETOUT

```
// dest rappresenta destinazione
```

<sup>11</sup>Per dettagli vedere il documento [4] alla bibliografia ed il codice sorgente di NXT Simulator.

<sup>12</sup>Vedi sezione 3.3.1 pagina 50

```
// source1 rappresenta la porta
// source2 rappresenta l'ID della proprieta'
// Estrazione del valore
num = NXTSView.getMotorController(table[source1].getValDefault()).
    getValue(source2);
// Memorizzazione del valore
table[dest].setValDefault(num);
```

**Istruzioni complesse mancanti** Nella versione precedente di NXTSimulator alcune istruzioni non sono state implementate dato che le funzioni da loro svolte sono collegate a parti del simulatore che non erano ancora previste. La loro assenza nell'interprete ha portato però a una situazione problematica nel momento in cui queste risultano presenti nel file RXE. Dato che la loro dimensione è variabile, l'interprete è stato modificato per spostare il puntatore PC<sup>13</sup> dell'adeguata quantità di Byte, evitando così l'errata interpretazione di tutte le istruzioni successive.

Segue un elenco delle istruzioni mancanti:

- **OP\_FLATTEN** Parametri: Destinazione, Sorgente.  
Estende i dati della sorgente in un array destinazione.
- **OP\_UNFLATTEN** Parametri: Destinazione, Errore, Sorgente, Default.  
Converte l'array sorgente in un dato memorizzato poi in destinazione.
- **OP\_STRINGTONUM** Parametri: Destinazione, IndicePassato Sorgente, Indice, Default.  
Converte il numero intero all'interno della stringa sorgente e lo memorizza in destinazione. La stringa sorgente può contenere più di un numero (o anche altri dati); nel caso in cui nella posizione puntata da indice non vi sono numeri da convertire, verrà memorizzato il valore default.
- **OP\_STRCAT** Parametri: Dimensione, Destinazione, Sorgente1, ..., SorgenteN.  
Concatena una o più stringhe, memorizza il risultato nella stringa destinazione.
- **OP\_STRSUBSET** Parametri: Destinazione, Sorgente, Indice, Contatore.  
Memorizza una sottostringa di sorgente in destinazione partendo da indice e includendo "Contatore" caratteri.
- **OP\_STRTOBYTEARR** Parametri: Destinazione, Sorgente.  
Converte la stringa sorgente in un array di Byte senza segno, memorizza il risultato in destinazione. Il carattere di terminazione viene rimosso.
- **OP\_BYTEARRTOSTR** Parametri: Destinazione, Sorgente.  
Converte l'array sorgente di Byte senza segno nella stringa destinazione. Viene aggiunto il carattere di terminazione.

<sup>13</sup>Program Counter

### OutputPortConfigurationProperties.java

Le modifiche apportate a questa classe comprendono l'aggiustamento delle imprecisioni e delle soluzioni provvisorie prese nella precedente versione del simulatore. Tra queste troviamo: interpretazione del movimento di un solo servomotore, soluzione provvisoria al movimento di due servomotori, non applicazione del fattore di sterzata.

**Movimento di uno o più servomotori** Come analizzato nell'istruzione OP\_SETOUT, è prevista dal firmware la possibilità di applicare gli stessi parametri alle proprietà di un solo o più servomotori (fino al massimo di 3). L'istruzione prevede quindi due casi: nel primo, un solo servomotore viene modificato, e quindi per identificarlo il numero di porta è scritto direttamente nel CodeSpace come secondo parametro di OP\_SETOUT; nel secondo, più servomotori vengono modificati, e per identificarli viene utilizzato un sistema indiretto, cioè i valori non sono inseriti nel CodeSpace ma al loro posto si trova un identificatore ad un array di dati.

Da una iniziale analisi di un file RXE appositamente creato per comandare l'azione di più servomotori, l'array puntato da SETOUT risultava vuoto o mancante. Inizialmente la strada intrapresa per poter scoprire come correggere il simulatore è stata quella di creare due file contenenti lo stesso programma con l'unica differenza delle due porte (ad esempio: A-B e B-C) scelte per il movimento dei motori. Tramite un analizzatore di binario esadecimale risulta che i due file differiscono per un solo valore iniziale della DSTOC ma che non è associato ad un array, quanto a una variabile di tipo Byte. Procedendo nell'analisi dei file decompilati da binario a *NeXT Byte Codes*, si è notato come le istruzioni precedenti manipolino i dati contenuti nella DSTOC per preparare l'array che sarà poi utilizzato per indicizzare le porte dei servomotori.

```

284          and      s10003, ub003B, s1001E
285          tst      NEQ, ub003C, s10003
286          mov      a0086, a007E
287          brtst    EQ, 1b100F8, ub003C
288          mov      a0086, a0080
289      1b100F8:      and      s10003, ub003B, s1001D
290          tst      NEQ, ub003C, s10003
291          mov      a0084, a007E
292          brtst    EQ, 1b10106, ub003C
293          mov      a0084, a007C
294      1b10106:      and      s10003, ub003B, s1001C
295          tst      NEQ, ub003B, s10003
296          mov      a0082, a007E
297          brtst    EQ, 1b10114, ub003B
298          mov      a0082, a007A
299      1b10114:      arrbuild a0088, a0086, a0084, a0082

```

In questo frammento di codice, si nota come l'array "a0088" viene costruito partendo dagli array "a0086", "a0084", "a0082", i quali vengono costruiti con dei valori (le porte) a seconda dei risultati di espressioni booleane di controllo. Ognuno dei tre sotto-array infatti può essere o vuoto o composto di un solo dato

(la porta) a seconda del valore delle variabili usate per il confronto. Ad esempio l'array "a0086" può essere la copia o di "a007E" o di "a0080" a seconda dei valori delle variabili "sl0003" e "ub003C". Proseguendo a ritroso nel file, si evince che il valore che differisce tra i due RXE viene utilizzato proprio in questi confronti per la costruzione dell'array "a0088", poi usato per l'istruzione SETOUT.

L'intuizione di utilizzare l'array puntato dall'istruzione era quindi giusta ma l'array risultava ancora vuoto o danneggiato, la soluzione al problema è stata quindi di procedere alla correzione di tutte le istruzioni che manipolano i dati (ed in particolare gli array)<sup>14</sup>.

**Fattore di sterzata** Quando si imposta una certa potenza a una coppia di motori, è possibile definire un fattore di sterzata per distribuire in maniera non uniforme la potenza ed ottenere così una virata della direzione robot.

Considerando come positiva la rotazione in senso orario delle ruote e supponendo che questa rotazione muova il robot in avanti, se la ruota<sub>1</sub> ha una velocità angolare di  $\omega_1$  e la ruota<sub>2</sub> ha una velocità angolare di  $\omega_2$ , con  $\omega_2 \geq \omega_1$  si verifica una virata e le due ruote disegnano due traiettorie circolari concentriche. La velocità del servomotore ( $\omega$  in radianti al secondo) è direttamente proporzionale alla potenza  $P$  ad esso applicata secondo la relazione  $\omega = k_{\omega p} P$ . Per carichi pesanti la linearità della relazione è limitata dalla potenza dei servomotori ma idealmente, ai fini della simulazione, è possibile trascurare questo fattore.

Per ottenere quindi una sterzata vi sono vari metodi: utilizzando il comando SETOUT per impartire due potenze diverse ai due motori, e, con lo stesso comando, erogare pari potenza a due motori contemporaneamente applicando un fattore di sterzata, oppure modificando il parametro denominato *steering* nel blocco move in NXT-G (che produrrà poi un file RXE con l'unico comando SETOUT per entrambi i motori più la sterzata).



Figura 3.8: Blocco Move di NXT-G con fattore di sterzata modificato.

Il parametro *steering* controlla il rapporto tra le velocità angolari dei due motori scelti. Sfortunatamente non esiste documentazione che descriva l'esatto comportamento del firmware in proposito e sono state necessarie delle misure

<sup>14</sup>Vedi sezione 3.3.1 pagina 34

empiriche. Si assuma che, quando il parametro direzione è impostato per il movimento in avanti, la velocità angolare del motore sia positiva. Il parametro *steering* può variare da -100 a +100, dove 0 corrisponde al movimento parallelo dei motori ( $\omega_1 = \omega_2 > 0$ ), valori negativi fanno svoltare verso destra ( $\omega_1 > \omega_2$ ) e valori positivi fanno svoltare verso sinistra ( $\omega_1 < \omega_2$ ). Con i valori agli estremi il robot ruota attorno al punto medio ( $\omega_1 = -\omega_2$ ,  $\omega_1 > 0$  con -100 e  $\omega_1 < 0$  con +100).

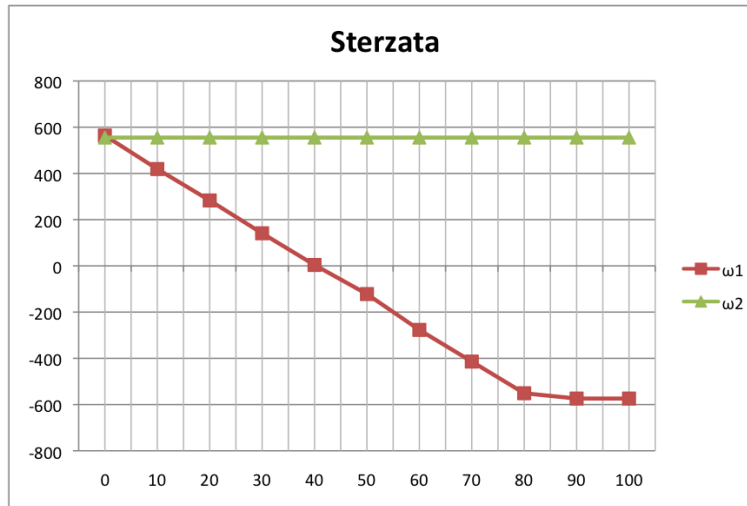


Figura 3.9: Rapporto tra le velocità angolari al variare di Steering.

Come si può vedere nel grafico in figura 3.9, aumentare il valore di *steering* mantiene praticamente costante il valore di  $\omega_2$  e fa decrescere il valore di  $\omega_1$  in maniera quasi lineare. In pratica il comportamento del robot si può riassumere, indicando con  $S$  il valore di *steering*, così:

$$S \in [0, 80] \rightarrow \omega_1 = \frac{40 - S}{40} * \omega_2$$

$$S \in [80, 100] \rightarrow \omega_1 = -\omega_2$$

Per quanto concerne il simulatore, questa approssimazione è molto buona in quanto emula la realtà con un grado di fedeltà più che accettabile.

#### Codice 3.17: OutputPortConfigurationProperties.java

```
ArrayList arrayporte = new ArrayList();
int indexarrayporte = 0;
int porta = -1;
/* Due casi:
 * Porta singola il cui valore e' inizializzato nell'istruzione
 * Porte multiple con valori salvati nell'array
 */
if (entry[conv.getPosInt(coppia)].getType().equals("TC_Array")){
    // Caso Array
    // Ricerca della entry della DSTOC che contiene il descrittore dell'array
    int posizionedstoc = 0;
    for (int s = 0; s < entry.length; s++){
        if (entry[s].getDataDescriptor() == entry[conv.getPosInt(coppia)].
            getDataDescriptor())
            { posizionedstoc = s; break; }
    }
```



```

// Traduzione della posizione nel DSTOC all'indice di "array"
indexarrayporte = calcolaIndexArray(array, posizionedstoc);
// Estrazione numeri di porta dall'array
for (int c = 3; c < array[indexarrayporte].size(); c++){
    try {
        arrayporte.add(((Byte)array[indexarrayporte].get(c)).intValue());
    }
    catch(ClassCastException e) {
        System.out.println("Conversion Error");
        arrayporte.add(((Integer)array[indexarrayporte].get(c)).intValue());
    }
}
} else {
    // Caso porta singola
    porta = entry[conv.getPosInt(coppia)].getValDefault();
}
// Inizializzazione dei motori a seconda delle porte utilizzate
// Il motor data mc (primo motore) e' sempre inizializzato perche' almeno una
// porta e' utilizzata!
MotorData mc = null;
MotorData mc1 = null;
MotorData mc2 = null;
if (porta != -1) mc = NXTSView.getMotorController(porta);
else
{
    if (arrayporte.size() == 2)
    {
        mc = NXTSView.getMotorController(((Integer)arrayporte.get(0)).intValue());
        mc1 = NXTSView.getMotorController(((Integer)arrayporte.get(1)).intValue());
    }
    else
    {
        if (arrayporte.size() == 3)
        {
            mc = NXTSView.getMotorController(((Integer)arrayporte.get(0)).intValue());
            mc1 = NXTSView.getMotorController(((Integer)arrayporte.get(1)).intValue());
            mc2 = NXTSView.getMotorController(((Integer)arrayporte.get(2)).intValue());
        }
    }
}
}

// Qui' viene omissso, per brevit , il codice della scrittura delle proprieta'
// del motore

// Nel caso vi siano 2 o 3 porte e' necessario applicare i valori anche agli
// altri motori
if(mc1 != null){
    mc1.FLAGS = mc.FLAGS;
    mc1.MODE = mc.MODE;
    mc1.SPEED = mc.SPEED;
    // Supporto alla sterzata di due ruote
    int vel = mc.SPEED;
    int ster = mc.TURN_RATIO;
    if (ster>0){
        if (ster>=80){
            mc1.SPEED = -vel;
        }else{
            mc1.SPEED = (int)((40-(float)ster)/40*(float)vel);
        }
    }
    if (ster<0){
        if (ster>=80){
            mc.SPEED = -vel;

```

```

    }else{
        mc.SPEED = (int)((40+(float)ster)/40*(float)vel);
    }
}
mc1.ACTUAL_SPEED = mc.ACTUAL_SPEED;
mc1.TACH_COUNT = mc.TACH_COUNT;
mc1.TACH_LIMIT = mc.TACH_LIMIT;
mc1.RUN_STATE = mc.RUN_STATE;
mc1.TURN_RATIO = mc.TURN_RATIO;
mc1.REG_MODE = mc.REG_MODE;
mc1.OVERLOAD = mc.OVERLOAD;
mc1.REG_P_VALUE = mc.REG_P_VALUE;
mc1.REG_I_VALUE = mc.REG_I_VALUE;
mc1.REG_D_VALUE = mc.REG_D_VALUE;
mc1.BLOCK_TACH_COUNT = mc.BLOCK_TACH_COUNT;
mc1.ROTATION_COUNT = mc.ROTATION_COUNT;
}

if(mc2 != null){
    mc2.FLAGS = mc.FLAGS;
    mc2.MODE = mc.MODE;
    mc2.SPEED = mc.SPEED;
    mc2.ACTUAL_SPEED = mc.ACTUAL_SPEED;
    mc2.TACH_COUNT = mc.TACH_COUNT;
    mc2.TACH_LIMIT = mc.TACH_LIMIT;
    mc2.RUN_STATE = mc.RUN_STATE;
    mc2.TURN_RATIO = mc.TURN_RATIO;
    mc2.REG_MODE = mc.REG_MODE;
    mc2.OVERLOAD = mc.OVERLOAD;
    mc2.REG_P_VALUE = mc.REG_P_VALUE;
    mc2.REG_I_VALUE = mc.REG_I_VALUE;
    mc2.REG_D_VALUE = mc.REG_D_VALUE;
    mc2.BLOCK_TACH_COUNT = mc.BLOCK_TACH_COUNT;
    mc2.ROTATION_COUNT = mc.ROTATION_COUNT;
}

```

## Correzioni a InsDstoc.java

Lo scopo di questa classe è quello di inizializzare la DSTOC nella memoria del simulatore, cioè inserire i dati statici nei record della tabella stessa e creare le strutture per il contenimento dei dati dinamici (array).

Tramite la libreria *java.util.\** viene letta la porzione del file RXE che contiene i suddetti dati ed inserita nell'oggetto *lett*. Utilizzando poi i valori dei flag delle entry DSTOC, si procede all'inserimento dei dati statici. Successivamente a questa fase, è necessario riorganizzare la memoria perché i dati dinamici siano accolti in strutture dedicate, che permettano al meglio la loro gestione<sup>15</sup>.

Anche in questo caso gli errori sono stati scoperti in due momenti, inizialmente testando alcuni file RXE che risultavano non funzionanti, e poi effettuando un debug a ritroso (partendo dai bug noti e cercando poi altri problemi).

Da principio il bug sembrava limitarsi alla parte di codice che gestisce l'inserimento dei valori statici di default nella DSTOC. Non era infatti ipotizzata una situazione particolare, la quale prevede che, se un valore è inferiore ai 4 Byte di lunghezza (quella massima), viene effettuato un *padding*<sup>16</sup>, per arrivare a tale

<sup>15</sup>Le strutture **array** e **arrayCluster** discusse nella sezione precedente.

<sup>16</sup>Inserimento di valori nulli.

portata. Quindi, quando ad esempio doveva essere inizializzato un TC\_UWORD da 16bit o un TC\_UBYTE da 8bit, non era considerato il padding adiacente e la lettura dei successivi valori veniva compromessa. È stato corretto poi il codice che gestisce la stessa funzione, ma per i valori di default dinamici. Questo "allineamento" dei valori nel file è dovuto al fatto che la VM necessita di un accesso veloce ai dati, ma il processore ARM7, utilizzato nel Brick NXT, non può accedere direttamente ai dati non allineati.

Codice 3.18: Spostamento corretto del puntatore dei valori di default

```
// puntatore ai dati
pData = 0;
for(int el = 0; el < valDopVec; el++){
    if (!(arrayDopeVector[el].getElementCount() == 0))
        // pData si deve muovere di (numero elementi)*(
        // dimensione elementi)
        pData += arrayDopeVector[el].getElementSize()*
        arrayDopeVector[el].getElementCount();
    // Se pData non si trova in una posizione multiplo di 4 va'
    // fatto avanzare:
    while ((pData%4) != 0) {
        pData++;
    }
}
```

Altra problematica risolta è la stessa che affliggeva le istruzioni che manipolano gli array. Infatti i casi di array di array o array di cluster che contenevano array, non prevedevano la caratteristica struttura descritta a pagina 38.

Codice 3.19: Esempio del caso array nell'array di cluster

```
// Caso array di cluster
arrayCluster[ac][0].add((Integer)i); // salvataggio indice del DSTOC
arrayCluster[ac][1].add((Integer)z); // memorizzazione posizione del
// dope vector
// Nel caso in cui vi sia un (o piu') array nel cluster e'
// necessario spostarsi in avanti per copiare anche il tipo di array
int ko = 0; // Fattore di correzione offset
for (int j = 0; j < table[i+1].getDataDescriptor(); j++){
    if (!table[i+j+ko+2].getType().equals("TC_Array"))
        arrayCluster[ac][j+ko+2].add(table[i+j+ko+2]);
    else {
        arrayCluster[ac][j+ko+2].add(table[i+j+ko+2]); //
        // Salvataggio array
        ko++;
        arrayCluster[ac][j+ko+2].add(table[i+j+ko+2]); // e
        // salvataggio del tipo
    }
}
```

### Analisi del problema al movimento di tre motori

Durante la scrittura del codice associato ai motori, in una sessione di test, è stato scoperto un bug che si manifesta con l'utilizzo di un particolare file RXE creato con NXT-G. Utilizzando il blocco Move per impostare il movimento simultaneo dei tre servomotori per un certo periodo di tempo, il compilatore di NXT-G produce un codice particolare, il quale nel simulatore si blocca in un punto specifico.

Il comportamento del simulatore è quello di interpretare correttamente le istruzioni per il movimento dei motori, ma poi di *"intrappolarsi"* su una routine, la quale risulta un ciclo che legge la proprietà `RUN_STATE` dei motori e ne controlla il valore: se risulta diverso da `RUN_STATE_IDLE` continua a ciclare, altrimenti prosegue con l'esecuzione del codice che porta all'interruzione dei motori. La suddetta proprietà rappresenta lo stato del motore e può avere 4 valori: `RUN_STATE_IDLE`, `RUN_STATE_RAMPUP`, `RUN_STATE_RUNNING`, `RUN_STATE_RAMPDOWN`, i quali rappresentano gli stati del servomotore: *porta disabilitata*, *potenza incrementata automaticamente "a rampa"*, *porta abilitata*, *potenza decrementata automaticamente "a rampa inversa"*.

Codice 3.20: Frammento di codice NeXT Byte Codes del RXE anormale

```
1b1033C:      add      s10016, s10036, ub00A3
              cmp      GTEQ, ub004E, s10016, s10035
              index    ub005F, a00EA, s10036
              getout    ub005F, ub005F, RunState
              tst       EQ, ub005F, ub005F
              or        ub004E, ub005F, ub004E
              brtst     NEQ, 1b1035C, ub004E
              add       s10036, s10036, ub00A2
              jmp       1b1033C
```

Dal frammento si evince che, finché il parametro letto in `"getout ub005F, ub005F, RunState"` non conterrà lo stesso valore salvato nella variabile `"ub004E"`, l'operazione di test `"brtst NEQ, 1b1035C, ub004E"` non risulterà vera e non verrà effettuato il salto al label `"1b1035C"` esterno al ciclo. L'unica soluzione possibile a questa situazione è la modifica del suddetto parametro da parte del firmware, ma se e come questo avvenga non è documentato.

**Analisi codice sorgente del firmware** Per cercare di risolvere la problematica esposta al paragrafo precedente, si è tentata l'analisi del codice sorgente del firmware NXT; tramite l'accettazione dell'accordo di licenza "LEGO Open Source License Agreement" è possibile scaricare dal sito ufficiale LEGO<sup>17</sup> un archivio contenente i file in linguaggio C [9].

In particolare sono interessati all'analisi `d_output.c` e `c_output.c` cioè i due file che contengono le istruzioni per controllare l'output del Brick NXT, i motori.

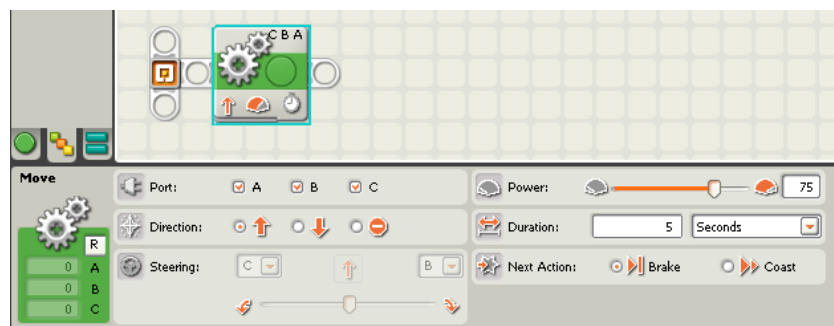
<sup>17</sup><http://mindstorms.lego.com/Overview/nxtreme.aspx>

Le funzioni controllate sono:

- `void dOutputCtrl(void)`: la funzione viene chiamata ogni millisecondo e controlla ogni motore presente. Vengono passati al controller vari parametri, tra i quali: la velocità attuale, il numero di giri effettuati, la differenza di giri effettuati, ecc...
- `void dOutputEnableRegulation(UBYTE MotorNr, UBYTE RegulationMode)`: funzione chiamata da `dOutputCtrl(void)` utilizzata per resettare i parametri di regolazione in base allo stato del motore.
- `void dOutputSetSpeed (UBYTE MotorNr, UBYTE NewMotorRunState, SBYTE Speed, SBYTE NewTurnParameter)`: funzione chiamata da `dOutputCtrl(void)` utilizzata per impostare i parametri di velocità e modalità del motore.
- `void cOutputCtrl(void)`: dal file `c_output.c` richiama varie funzioni del file `d_output.c`.

Non è stata trovata alcuna informazione utile riguardante il possibile cambiamento di stato del parametro `RUN_STATE` dei motori nella situazione rappresentata nel RXE in esame. Saranno necessari quindi maggiori approfondimenti nell'analisi del file per capire se veramente si tratta di un problema di simulazione o di altra natura.

Figura 3.10: Programma NXT-G per il movimento di tre motori.



Da notare il fatto che, scrivendo il programma equivalente in linguaggio NXC, questo viene interpretato correttamente.

Codice 3.21: Codice NXC equivalente al programma in figura 3.10.

```
task main() {
    int po[] = {0,1,2};
    OnFwd(po, 75);
    Wait(5000);
}
```

**Possibile soluzione** Effettuando ulteriori sessioni di debugging, sono emersi problemi nell'esecuzione di file prodotti tramite NXT-G che sono riconducibili ad un possibile errore comune alla suddetta problematica. Tutto ciò è imputabile ad un'errata gestione da parte del simulatore dello scheduling dei Clump nella VM. Infatti è provato, tramite test su file RXE, che i Clump non sono sempre correttamente pianificati nei casi di esecuzione multipla o comunque sufficientemente complessa. Il codice che gestisce queste situazioni risiede nel metodo `public int[] whoSchedule(int s1, int s2, int cN)` della classe **Interpreter.java**. Una correzione (o riscrittura se necessario) potranno certamente aiutare nella soluzione della problematica in questione.

**MotorPanel.java** Durante il debug per la correzione del problema al movimento dei tre motori, sono emersi errori anche nella classe che gestisce i pannelli che li rappresentano. In particolare non era aggiornata la variabile che rappresenta la velocità attuale del motore (potenza applicata), e quindi non era modificabile in fase di esecuzione se non per il primo valore impostato. Si è quindi proceduto all'inserimento delle seguenti righe di codice:

```
// Alcune righe di codice dalla classe MotorPanel.java
speed = NXTSView.getMotorController( portIndex ).SPEED;
if (speed == 0) NXTSView.getMotorController(portIndex).REG_MODE = 0;
```

**Conclusioni** Le correzioni apportate al codice dell'interprete hanno portato ad un simulatore stabile e funzionale. Gli ulteriori passi da fare nello sviluppo saranno quelli di verificare il codice di scheduling della classe **Interpreter.java** ed implementare le istruzioni mancanti e le funzioni di sistema ad esse collegate.

### Esempi caratteristici

Per la costruzioni di file RXE funzionanti è consigliato utilizzare il linguaggio NXC ed impiegare, oltre che ai costrutti base di programmazione, gli array<sup>18</sup> e le funzioni per la loro manipolazione, quali:

```
num = ArrayLen(a);
ArrayInit(a, val, cnt);
ArraySubset(aout, asrc, idx, len);
ArrayBuild(aout, src1, ..., srcN);
str = NumToStr(num);
funzioni per il controllo dei motori, quali:
Coast(ports);
Float(ports);
Off(ports);
OnFwd(ports, power);
OnRev(ports, power);
OnFwdReg(ports, power, regmode);
```

---

<sup>18</sup>Come definito in sezione 2.2.2 pagina 13.

```

OnRevReg(ports, power, regmode);
OnFwdSync(ports, power, turnpct);
OnRevSync(ports, power, turnpct);
CoastEx(ports, reset);
OffEx(ports, reset);
OnFwdEx(ports, power, reset);
OnRevEx(ports, power, reset);
OnFwdRegEx(ports, power, regmode, reset);
OnRevRegEx(ports, power, regmode, reset);
OnFwdSyncEx(ports, power, turnpct, reset);
OnRevSyncEx(ports, power, turnpct, reset);
RotateMotor(ports, power, angle);
RotateMotorEx(ports, power, angle, turnpct, sync, stop);
RotateMotorPID(ports, power, angle, p, i, d);
RotateMotorExPID(ports, power, angle, turnpct, sync, stop, p, i, d);
SetOutput(ports, field1, value1, ..., fieldN, valueN);
val = GetOutput(port, field);
val = MotorMode(p);
val = MotorPower(p);
val = MotorActualSpeed(p);
val = MotorTachoCount(p);
val = MotorTachoLimit(p);
val = MotorRunState(p);
val = MotorTurnRatio(p);
val = MotorRegulation(p);
val = MotorOverload(p);
val = MotorRegPValue(p);
val = MotorRegIValue(p);
val = MotorRegDValue(p);
val = MotorBlockTachoCount(p);
val = MotorRotationCount(p);

```

ed oggetti complessi (che sono tradotti in cluster nel file RXE).

Un semplice esempio:

```

struct prova { // Struttura complessa tradotta come cluster
    byte aa;
    byte bb;
};
// Array di tipo "prova"
prova a[];
prova b[];

task main() {
    prova x, y;
    x.aa = 1;
    x.bb = 2;
}

```

```

y.aa = 3;
y.bb = 4;
a[0] = x;
a[1] = y;
// Costruzione array "b" dagli elementi: "a" array e "x"
ArrayBuild(b,a,x);
// Avvio motore alla porta 0 con potenza 75
OnFwd(0, 75);
// Attesa di 2 sec
Wait(2000);
} // Fine del programma

```

### 3.3.2 Integrazione dei sensori di input

#### Premessa

In NXTSimulator i sensori di input sono già predisposti per la visualizzazione nell'interfaccia grafica come pannelli istanziabili dall'utente<sup>19</sup>. I pannelli sono di due tipi: uno per i sensori di luce, suono e distanza e uno per il sensore di tocco. I primi tre tipi sono accomunati dalla stessa modalità di input (tramite slider), mentre le differenze sono minime, quali l'immagine visualizzata ed il fondo scala dei dati in input. Sono quindi presenti due classi: *SensorGenericPanel.java* e *SensorTouchPanel.java*. Entrambe estendono *GenericPanel* di Java ed implementano i componenti: etichetta per distinguere il numero di porta, etichetta per il tipo di sensore, bottone per la rimozione del pannello, immagine per identificare il sensore, slider o bottoni per l'input (a seconda del tipo). Il controllo delle proprietà dei sensori spetta a queste classi per quanto riguarda l'interazione dell'utente, mentre l'interprete può accedere per leggere le informazioni immesse o impostare dei parametri di controllo.

#### Le classi e strutture

**SensorData.java** Utilizzata per memorizzare le proprietà dei sensori collegati al Brick virtuale, la classe rispecchia in tutto e per tutto i dati utilizzati nella VM del firmware 1.03. Durante l'esecuzione ogni parte del simulatore farà riferimento a questa classe, istanziata negli oggetti *s1*, *s2*, *s3*, *s4* che rappresentano le 4 porte.

Codice 3.22: Classe SensorData

```

public class SensorData {
    // Vengono omessi da questo listato di codice
    // il costruttore ed i metodi della classe,
    // i quali servono solo ad accedere in lettura/scrittura
    // alle seguenti variabili

    private int portN;
    //IDProprieta' definite come in InputPortConfigurationData
    public byte[] IN_TYPE;
    /*
        Legal Values:

```

<sup>19</sup>Vedi figura 4.1 alla sezione 3.2.



```

0x00 NO_SENSOR          No sensor configured.
0x01 SWITCH             NXT or RCX touch sensor
0x02 TEMPERATURE        RCX temperature sensor
0x03 REFLECTION          RCX light sensor
0x04 ANGLE              RCX rotation sensor
0x05 LIGHT_ACTIVE       NXT light sensor with floodlight enabled
0x06 LIGHT_INACTIVE     NXT light sensor with floodlight disabled
0x07 SOUND.DB           NXT sound sensor; dB scaling
0x08 SOUND.DBA          NXT sound sensor; dBA scaling
0x09 CUSTOM             Unused in NXT programs
0x0A LOWSPEED           I2C digital sensor
0x0B LOWSPEED_9V        I2C digital sensor; 9V power
0x0C HIGHSPEED          Unused in NXT programs
*/
public byte[] IN_MODE;
/*
    Legal values:
    0x00 RAWMODE          Report scaled value equal to raw value.
    0x20 BOOLEANMODE      Report scaled value as 1 (TRUE) or 0 (FALSE).
                        The firmware uses inverse Boolean logic to match the physical
                        characteristics of NXT sensors. Readings are FALSE if raw value
                        exceeds 55% of total range; readings are TRUE if raw value is less
                        than 45% of total range.
    0x40 TRANSITIONCNTMODE Report scaled value as number of transitions
                        between TRUE and FALSE.
    0x60 PERIODCOUNTERMODE Report scaled value as number of transitions
                        from FALSE to TRUE, then back to FALSE.
    0x80 PCTFULLSCALEMODE Report scaled value as percentage of full scale
                        reading for configured sensor type.
    0xA0 CELSIUSMODE      Scale TEMPERATURE reading to degrees Celsius.
    0xC0 FAHRENHEITMODE   Scale TEMPERATURE reading to degrees Fahrenheit
                        .
    0xE0 ANGLESTEPMODE     Report scaled value as count of ticks on RCX-
                        style rotation sensor.
*/
public int IN_ADRAW;      // [0, 1023]
public int IN_NORMRAW;    // [0, 1023]
public int IN_SCALED_VAL;
/*
    The legal value range depends on MODE, as listed below:
    RAWMODE                [0, 1023]
    BOOLEANMODE            [0, 1]
    TRANSITIONCNTMODE      [0, 65535]
    PERIODCOUNTERMODE      [0, 65535]
    PCTFULLSCALEMODE       [0, 100]
    CELSIUSMODE            [-200, 700] (readings in 10th of a
                        degree Celsius)
    FAHRENHEITMODE         [-400, 1580] (readings in 10th of a
                        degree Fahrenheit)
    ANGLESTEPMODE          [0, 65535]
*/
public int IN_INVALID_DATA; // [0,1] -> False, True
private Converter converter = new Converter();
}

```

**InputPortConfigurationProperties.java** Questa classe, già presente nella precedente versione di NXT Simulator, è stata modificata al fine di assicurare l'interazione con *SensorData*. La funzione alla quale è adibita è quella di fornire accesso in scrittura all'interprete sui parametri dei sensori; ad esempio è possibile impostare il tipo di sensore istanziato in una certa porta o rendere non validi i dati letti fino a quell'istante.

Codice 3.23: Classe InputPortConfigurationProperties

```

/* Omessi costruttore e metodi di conversione */
// source rappresenta il valore da scrivere
// port rappresenta la porta sulla quale e' installato il sensore
// propID rappresenta l'identificatore della proprieta'
public void device(){
    try{
        // Lettura della classe SensorData relativa alla porta scelta
        SensorData sensore = NXTView.getSensorController(port);
        switch (propID){ // Scelta della proprieta' da modificare
            case 0:
                codeBit = bit.bitConverter(source);
                // Scrittura del valore
                sensore.IN_TYPE = codeBit;
                for (int i=0; i<16; i++)
                    if (codeBit[i] != 0)
                        count++;
                if (count == 0){
                    System.out.println("    NO_SENSOR");
                    outGen.println("    NO_SENSOR");
                    outClump.println("    NO_SENSOR");
                }
                if (codeBit[14]==1){
                    System.out.println("    SWITCH");
                    outGen.println("    SWITCH");
                    outClump.println("    SWITCH");
                }
                if (codeBit[13]==1){
                    System.out.println("    TEMPERATURE");
                    outGen.println("    TEMPERATURE");
                    outClump.println("    TEMPERATURE");
                }
                if (codeBit[12]==1){
                    System.out.println("    REFLECTION");
                    outGen.println("    REFLECTION");
                    outClump.println("    REFLECTION");
                }
                if (codeBit[11]==1){
                    System.out.println("    ANGLE");
                    outGen.println("    ANGLE");
                    outClump.println("    ANGLE");
                }
                if (codeBit[10]==1){
                    System.out.println("    LIGHT_ACTIVE");
                    outGen.println("    LIGHT_ACTIVE");
                    outClump.println("    LIGHT_ACTIVE");
                }
                if (codeBit[9] ==1){
                    System.out.println("    LIGHT_INACTIVE")
                    ;
                    outGen.println("    LIGHT_INACTIVE")
                    ;
                    outClump.println("    LIGHT_INACTIVE")
                    ;
                }
                if (codeBit[8] ==1){
                    System.out.println("    SOUND_DB");
                    outGen.println("    SOUND_DB");
                    outClump.println("    SOUND_DB");
                }
                if (codeBit[7] ==1){
                    System.out.println("    SOUND_DBA");
                    outGen.println("    SOUND_DBA");
                    outClump.println("    SOUND_DBA");
                }
                if (codeBit[6] ==1){
                    System.out.println("    CUSTOM");
                    outGen.println("    CUSTOM");
                    outClump.println("    CUSTOM");
                }
            }
        }
    }
}

```

```

    }
    if (codeBit[5] ==1){
        System.out.println("        LOW_SPEED");
        outGen.println("        LOW_SPEED");
        outClump.println("        LOW_SPEED");
    }
    if (codeBit[4] ==1){
        System.out.println("        LOW_SPEED_9V");
        outGen.println("        LOW_SPEED_9V");
        outClump.println("        LOW_SPEED_9V");
    }
    if (codeBit[3] ==1){
        System.out.println("        HIGH_SPEED");
        outGen.println("        HIGH_SPEED");
        outClump.println("        HIGH_SPEED");
    }
    }
    break;
    case 1:
// Scrittura del valore
sensore.IN_MODE = codeBit;

    System.out.println();
    for (int i=0; i<16; i++)
        if (codeBit[i] != 0)
            count++;

    if (count == 0){
        System.out.println("        RAWMODE");
        outGen.println("        RAWMODE");
        outClump.println("        RAWMODE");
    }
    if (codeBit[10]==1){
        System.out.println("        BOOLEANMODE");
        outGen.println("        BOOLEANMODE");
        outClump.println("        BOOLEANMODE");
    }
    if (codeBit[9] ==1){
        System.out.println("        TRANSITIONCNTMODE");
        outGen.println("        TRANSITIONCNTMODE");
        outClump.println("        TRANSITIONCNTMODE");
    }
    if ((codeBit[10]==1) && (codeBit[9]==1)){
        System.out.println("        PERIODCOUNTMODE");
        outGen.println("        PERIODCOUNTMODE");
        outClump.println("        PERIODCOUNTMODE");
    }
    if (codeBit[8] == 1){
        System.out.println("        PCTFULLSCALEMODE");
        outGen.println("        PCTFULLSCALEMODE");
        outClump.println("        PCTFULLSCALEMODE");
    }
    if ((codeBit[8] ==1) && (codeBit[10]==1)){
        System.out.println("        CELSYNUSMODE");
        outGen.println("        CELSYNUSMODE");
        outClump.println("        CELSYNUSMODE");
    }
    if ((codeBit[8] ==1) && (codeBit[9]==1)){
        System.out.println("        FAHRENHEITMODE");
        ;
    }

```

```

        outGen.println("          FAHRENHEITMODE");
        ;
        outClump.println("          FAHRENHEITMODE");
        ;
    }
    if ((codeBit[8]==1) && (codeBit[9]==1) && (
        codeBit[10] == 1)){
        System.out.println("          ANGLESTEPMODE");
        outGen.println("          ANGLESTEPMODE");
        outClump.println("          ANGLESTEPMODE");
    }

    break;
    case 2:
    // Scrittura del valore
    sensore.IN_ADRAW = source;
    break;
    case 3:
    // Scrittura del valore
    sensore.IN_NORMRAW = source;
    break;
    case 4:
    // Scrittura del valore
    sensore.IN_SCALED_VAL = source;
    break;
    case 5:
    // Scrittura del valore
    sensore.IN_INVALID_DATA = source;
    break;
}
}
}

```

**SensorGenericPanel.java** Per l'integrazione del pannello dei sensori con le classi delle proprietà è stato scelto di procedere applicando la stessa metodologia usata per *MotorPanel*. Grazie all'oggetto *Timer*, della libreria *Swing* di Java, è possibile impostare un'azione che venga ripetuta ciclicamente nel tempo mentre il Timer è attivo, cosa indispensabile per leggere i valori che l'utente può impostare attraverso l'interfaccia. Questa classe dispone dei metodi per l'avvio (metodo *start()*) ed il blocco del timer (metodo *stop()*), necessita della dichiarazione del codice da eseguire e l'intervallo di tempo tra un'esecuzione e l'altra.

L'azione del timer è definita nel costruttore di *SensorGenericPanel*:

```

// Questi valori sono utilizzabili per il sensore luce , ultrasonico
// e suono
timer = new Timer(400 , new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // Il valore di IN_SCALED_VAL dipende dal valore di IN.MODE
        // come da specifiche firmware
        NXTSView.getSensorController(indice).IN_SCALED_VAL = 0;
        // Modo PCTFULLSCALEMODE
        if (NXTSView.getSensorController(indice).IN_MODE[8] == 1)
            NXTSView.getSensorController(indice).IN_SCALED_VAL =
                slider.getValue();
        // Modo TRANSITIONCNIMODE o PERIODCOUNTERMODE o
        // ANGLESTEPMODE
        if (NXTSView.getSensorController(indice).IN_MODE[6] == 1 ||
            NXTSView.getSensorController(indice).IN_MODE[4] == 1 ||

```

```

        NXTSTView.getSensorController(indice).IN_MODE[14] == 1)
        NXTSTView.getSensorController(indice).IN_SCALED_VAL = (int)
        )(slider.getValue()*655.35F);
    // Modo BOOLEANMODE
    if (NXTSTView.getSensorController(indice).IN_MODE[2] == 1) {
        // Usata la logica inversa
        if (slider.getValue() >= 55) NXTSTView.
            getSensorController(indice).IN_SCALED_VAL = 0;
        if (slider.getValue() <= 45) NXTSTView.
            getSensorController(indice).IN_SCALED_VAL = 1;
    }
    NXTSTView.getSensorController(indice).IN_NORMRAW = (int)(
        slider.getValue()*10.24F);
    NXTSTView.getSensorController(indice).IN_INVALID_DATA = 0;
    }
});
timer.setInitialDelay(0);

```

Mentre le azioni si start e stop sono aggiunte alla classe *SensorGenericPanel* come metodi pubblici, quindi accessibili nel codice del simulatore:

```

public void startTimer(){
    timer.start();
}
public void stopTimer(){
    timer.stop();
    // Lo slider viene riportato al valore medio
    slider.setValue(50);
}

```

Questi metodi sono stati chiamati quindi nella classe principale del simulatore: *NXTSTView* e precisamente all'interno del metodo *stopSim()*, il quale viene eseguito quando la simulazione termina (o viene interrotta) assieme ad altre azioni quali: la fermata dei timer dei motori, il ripristino nei menù bloccati e del Thread dell'interprete.

Codice 3.24: Metodo stopSim() della classe NXTSTView

```

public static void stopSim(){
    // Interruzione del Thread dell'interprete
    sim.halt();
    // Disabilitazione dei sensori
    if ((!isEmpty(1)) & (prt1 instanceof SensorGenericPanel)){
        ((SensorGenericPanel)prt1).stopTimer();
    }
    if ((!isEmpty(2)) & (prt2 instanceof SensorGenericPanel)){
        ((SensorGenericPanel)prt2).stopTimer();
    }
    if ((!isEmpty(3)) & (prt3 instanceof SensorGenericPanel)){
        ((SensorGenericPanel)prt3).stopTimer();
    }
    if ((!isEmpty(4)) & (prt4 instanceof SensorGenericPanel)){

```

```

        ((SensorGenericPanel)prt4).stopTimer();
    }
    // Riabilitazione dei menu
    CRController.menuHiderShower();
    showTimerStatus();
    // Arresto dei timer dei motori
    stopMotors();
    // Azioni accessorie
    showTimerStatus();
    new Msg(SimulationEnd,"messageL").start();
    busyIconTimer.stop();
    statusAnimationLabel.setIcon(idleIcon);
    // Creazione nuovo Thread dell'interprete dallo stesso RXE
    sim = new Interpreter(fileR);
}

```

**SensorTouchPanel.java** Similmente al pannello generico, anche per il sensore di tocco sono state apportate delle modifiche che consentano la comunicazione tra l'utente ed i dati del sensore utili alla simulazione. In questo caso però si è scelto di non utilizzare la classe *Timer*, ma piuttosto di sfruttare gli eventi dei bottoni presenti nel pannello.

Il sensore di tatto può essere premuto, rilasciato o premuto velocemente.

Codice 3.25: Metodo setStatus(String str) della classe SensorTouchPanel

```

// Chiamato ad ogni cambiamento di stato del bottone
private void setStatus(String str){
    // Aggiorna la stringa che e' mostrata all'utente
    statusL.setText(str);
    String stato = str.toLowerCase();
    // A seconda dello stato imposta i valori
    if (stato.equals("bump") | stato.equals("pushed") | stato.equals("premuto")) {
        NXTSView.getSensorController(indice).IN_SCALED_VAL = 1;
        NXTSView.getSensorController(indice).IN_NORMRAW = 1;
        NXTSView.getSensorController(indice).IN_INVALID_DATA = 0;
    } else {
        NXTSView.getSensorController(indice).IN_SCALED_VAL = 0;
        NXTSView.getSensorController(indice).IN_NORMRAW = 0;
        NXTSView.getSensorController(indice).IN_INVALID_DATA = 0;
    }
}
}

```

## Esempi caratteristici

Oltre alle strutture NXC elencate a pagina 58, grazie all'integrazione dei sensori di input è possibile utilizzare anche:

```

ClearSensor(port);
ResetSensor(port);

```

```
val = Sensor(port);
val = SensorUS(port);
SetSensorLight(port);
SetSensorSound(port);
SetSensorTouch(port);
SetSensorLowspeed(port);
SetSensorType(port, type);
SetSensorMode(port, mode);
SetInput(port, field, value);
val = GetInput(port, field);
val = SensorType(p);
val = SensorMode(p);
val = SensorRaw(p);
val = SensorNormalized(p);
val = SensorScaled(p);
val = SensorInvalid(p);
```

Un semplice esempio:

```
task main(){
    // Impostazione tipo di sensore alla porta 3
    // (che equivale al valore 2), sens. prossimita'
    SetSensorType(2, IN_TYPE_REFLECTION);
    // Lettura distanza
    int val = SensorScaled(2);
    // Avvio motore porta A, potenza 75
    OnFwd(0, 75);
    while(val > 5){
        val = SensorScaled(2);
        // Da 50 o meno cm la potenza del motore
        // e' ridotta linearmente
        if(val < 50){
            // Calcoli interi per limitazione del firmware
            int a = val*1666/1000;
            SetOutput(0, 2, a);
        }
        else SetOutput(0, 2, 75);
    } // Il ciclo termina quando e' raggiunta una
        // distanza inferiore ai 5cm
} // Fine del programma
```

### 3.3.3 Bug noti

Al momento della stesura di questo elaborato non sono noti Bug per quanto riguarda i sensori e la loro gestione. L'unica nota da segnalare, riguarda il caso del movimento di tre motori azionati simultaneamente, utilizzando il file prodotto dal blocco "Move" in NXT-G (con lo stesso codice scritto in NXC il bug non è presente). Per eliminare l'errore si può suggerire di effettuare opportune e

ulteriori analisi del Byte Code nel RXE, magari visionando il codice passo-passo (accompagnando il test ad una verifica del codice che gestisce la schedulazione dei Clump).



## Capitolo 4

### Conclusioni e lavoro futuro

Lo sviluppo dell'applicazione NXTSimulator è stato eseguito in varie parti: dapprima con la costruzione della struttura dell'interprete, poi con la creazione della basilare interfaccia utente, seguita dall'integrazione al supporto delle localizzazioni<sup>1</sup> ed infine alla soluzione di Bug ed errori ed inserimento del codice per i sensori di input. Il programma si può definire stabile e maturo, ciò nonostante non vi è stato applicato il numero di versione 1 definitivo in quanto persistono alcune lacune dell'interprete, quali le istruzioni complesse mancanti. Pertanto, poiché le modifiche apportate hanno aggiunto nuove funzioni, è risultato più consono utilizzare il numero 0.9.

L'avvio alla programmazione di NXTSimulator ha richiesto, oltre alle conoscenze di programmazione e teoria degli elaboratori, che venisse compiuto uno studio rigoroso della VM<sup>2</sup>, delle specifiche del firmware [4] ed un approfondimento delle conoscenze contattando gli autori delle precedenti versioni [7] [8]. La chiave per conoscere al meglio il simulatore è stata però l'esperienza data dall'uso estensivo degli strumenti utilizzati per lo sviluppo dei file RXE, e dallo studio diretto del codice.

In definitiva questo progetto ha raggiunto lo scopo prefisso, cioè la correzione dei Bug (noti e non), l'ampliamento delle funzioni presenti e l'aggiunta della simulazione dei sensori di input.

Per raggiungere la versione completa, è necessario che venga terminato il lavoro di interpretazione delle istruzioni e implementato un sistema che permetta di utilizzare una modalità di input più elaborata per i sensori NXT, come ad esempio la possibilità di utilizzare un sistema automatizzato per i valori di input che segua determinate leggi o regole, ad andamento continuo o discreto. Facendo ciò si potrebbe quindi scavalcare il limite di utilizzo di un sensore alla volta (dato che il puntatore nei sistemi a mouse è uno solo). Altra possibilità di espansione futura consiste nel supporto ad altri tipi di sensore (disponendo della documentazione degli stessi) e, in un futuro più remoto, l'ampliamento dell'interfaccia

---

<sup>1</sup>Per l'utilizzo con più lingue.

<sup>2</sup>Virtual Machine del Brick NXT.

utente con la possibilità di "costruire" a video il robot, collegando le varie parti direttamente al Brick virtuale (se possibile in modalità 3D).

Il limite per attuare tutto ciò non è costituito dalla mancanza di librerie standard (già disponibili in Java), ma dalle esigenze e priorità future del progetto.



Figura 4.1: Assemblaggio del kit LEGO<sup>®</sup> Mindstorm<sup>®</sup> NXT

# Appendice A

## Istruzioni per il controllo di flusso

Sono riportate di seguito le istruzioni dedicate al controllo di flusso: queste modificano la pianificazione dei Clump e delle istruzioni all'interno degli stessi.

Quando ci si riferisce a "questo Clump", si intende il Clump ove l'istruzione descritta risiede.

Le istruzioni `OP_JMP`, `OP_BRCMP` e `OP_BRTST` modificano il program counter del proprio Clump per specificare quale fra le istruzioni del loro stesso Clump va schedulata successivamente.

`OP_BRCMP` e `OP_BRTST`, modificano condizionatamente il program counter del proprio Clump solo se i loro parametri assieme al codice di comparazione, producono un risultato *TRUE*. Le comparazioni di queste istruzioni utilizzando le stesse regole di `OP_CMP` e `OP_TST`<sup>1</sup>.

Le istruzioni `OP_FINCLUMP`, `OP_FINCLUMPIMMED`, `OP_SUBCALL` e `OP_SUBRET` modificano l'esecuzione del Clump in vari modi. Queste istruzioni riavviano o sospendono il Clump corrente e rilasciano il controllo all'algoritmo di schedulazione della VM.

**OP\_JMP** Parametri: Offset.

Modifica il program counter del Clump del valore immediato "Offset".

**OP\_BRCMP** Parametri: Offset, Sorgente1, Sorgente2.

Compara "Sorgente1" con "Sorgente2" secondo il codice di comparazione; se il risultato è *TRUE*, modifica il program counter del Clump del valore immediato "Offset".

**OP\_BRTST** Parametri: Offset, Sorgente.

Compara "Sorgente1" con zero secondo il codice di comparazione; se il risultato è *TRUE*, modifica il program counter del Clump del valore immediato "Offset".

---

<sup>1</sup>Vedi documento [4]

**OP\_STOP** Parametri: Conferma.

Interrompe il programma corrente se il valore di conferma è *TRUE* (intero non-zero). Nota: questa istruzione interrompe ogni Clump e provoca l'uscita immediata del programma dalla RAM.

**OP\_FINCLUMP** Parametri: Inizio, Fine.

Termina l'esecuzione del Clump. Se i valori "Inizio" e "Fine" sono interi positivi, essi sono utilizzati per indicizzare la lista dei Clump dipendenti di questo Clump, schedulando condizionatamente ognuno tra quelli specificati nel range.

Se i valori sono negativi, essi sono ignorati: non è previsto nessun altro Clump.

**OP\_FINCLUMPIMMED** Parametri: IDClump.

Termina l'esecuzione del Clump. Viene quindi schedulato il Clump identificato dal valore di "IDClump".

**OP\_ACQUIRE** Parametri: IDMutex.

Acquisisce il mutex identificato nel dataspace con "IDMutex". Se il mutex è già riservato piazza questo Clump nella relativa coda di attesa.

**OP\_RELEASE** Parametri: IDMutex.

Rilascia il mutex identificato nel dataspace con "IDMutex". Se la coda di attesa non è vuota, il Clump successivo in attesa acquisisce automaticamente il mutex e riprende l'esecuzione.

**OP\_SUBCALL** Parametri: Subroutine, IDChiamante.

Chiama il Clump identificato da "Subroutine" e sospende il chiamante (questo Clump). Salva l'ID del chiamante nella posizione "IDChiamante".

**OP\_SUBRET** Parametri: IDChiamante.

Ritorna dalla subroutine riattivando il Clump specificato da "IDChiamante" (vedi OP\_SUBCALL).

# Bibliografia

- [1] Project TERECOP, <http://www.terecop.eu/>, (2009)
- [2] NXT-G MINDSTORM NXT SOFTWARE,  
[http://mindstorms.lego.com/overview/NXT\\_Software.aspx](http://mindstorms.lego.com/overview/NXT_Software.aspx)
- [3] John Hansen. Not eXactly C Programmer's Guide,  
[http://bricxcc.sourceforge.net/nbc/nxcdoc/NXC\\_Guide.pdf](http://bricxcc.sourceforge.net/nbc/nxcdoc/NXC_Guide.pdf)
- [4] LEGO. MINDSTORMS NXT Executable File Specification.
- [5] John Hansen. Sourceforge NBC & NXC,  
<http://bricxcc.sourceforge.net/nbc/index.html>
- [6] Microsoft Robotic Studio,  
<http://msdn.microsoft.com/en-us/robotics/default.aspx>
- [7] Mauro Donadeo, Realizzazione di un simulatore elementare del sistema robotico Lego NXT in Java, 2008 Università degli Studi di Padova
- [8] Andrea Donè, Realizzazione di un semplice simulatore per il robot didattico Lego Mindstorms NXT, 2008 Università degli Studi di Padova
- [9] NXT Firmware Open Source, The Open Source files include all the source files needed for the ARM7 ATMEL microcontroller and the 8-bit AVR AT-MEL microcontroller..  
<http://mindstorms.lego.com/Overview/OpenSource.aspx>