# *NXTSimulator*: Sviluppo di Aggiornamenti ed Implementazione di chiamate a sistema per le funzionalità più avanzate

Laureando: Marco Guariso - Matricola n. 603783

Relatore: Prof. Michele Moro

**Corso di laurea Magistrale in Ingegneria Informatica**

**(indirizzo Gestionale)**

Data Laurea: 14/03/2011

ANNO ACCADEMICO 2010/2011

*Alla mia famiglia,*
*per come mi ha cresciuto,*
*a Fahira,*
*per avermi reso l'uomo più felice al mondo.*

# Indice

# Sommario

La seguente tesi illustra il processo di modifica, miglioramento ed estensione che è stato effettuato sul simulatore visuale (denominato *NXTSimulator*) per il robot Lego® Mindstorms® NXT. Lo scopo del lavoro è stato quello di rimediare a tutti i malfunzionamenti riscontrati nel suddetto simulatore e di integrarlo con delle nuove necessarie funzionalità.

*NXTSimulator*, nella sua versione 0.9b, era uno strumento che offriva la possibilità di simulare su un calcolatore il comportamento del robot, semplicemente fornendo in ingresso all'applicazione il programma che andava a determinare le azioni del robot stesso, senza dover disporre per forza di quest'ultimo. Tale simulatore, tuttavia, era lontano dall'essere nella sua versione definitiva, in quanto non era ancora in grado di simulare tutte le funzionalità del robot LEGO® [1s], ed alcune di quelle che già svolgeva non operavano in modo corretto.

Il progetto relativo al simulatore è inserito nell'ambito operativo di TERECoP [8s], un progetto europeo avente come scopo principale la realizzazione di una struttura atta alla formazione di insegnanti, nello sviluppo futuro di attività formative per studenti che comportino l'uso della robotica.

Il simulatore è stato interamente implementato nel linguaggio di programmazione ad oggetti Java [1t], e nella fattispecie utilizzando l'ambiente di sviluppo NetBeans [5s], entrambi di proprietà di Oracle®, e si è deciso di perseguire tale via anche per questo lavoro di tesi.

Il risultato raggiunto è stato la risoluzione di tutti i malfunzionamenti rilevati nel simulatore, e l'estensione dello stesso con nuove funzionalità aggiuntive, quali ad esempio la possibilità di simulare anche il display, il riproduttore di suoni, i bottoni del brick, e il gestore di file del robot. La nuova versione di *NXTSimulator*, ossia la 0.9c, non prevede ancora la possibilità di simulare tutta la gamma completa di funzioni del robot LEGO®, ma costituisce un grande passo in avanti verso una versione completa e definitiva.

# Capitolo 1

# Introduzione

Il progetto relativo ad *NXTSimulator* è iniziato nell'anno 2008 presso il Dipartimento di Ingegneria dell'Informazione dell'Università di Padova, con lo scopo di realizzare un simulatore funzionale del robot Lego® Mindstorms® NXT (per una cui descrizione dettagliata si rimanda al capitolo 2 di questo elaborato). L'obiettivo era quello di ottenere un software che simulasse al meglio il comportamento del robot tramite un ambiente grafico, che mostrasse all'utente le azioni che il robot stesso avrebbe effettuato qualora gli fosse stato caricato un certo programma.

La versione del simulatore precedente a questo elaborato, ossia la 0.9b, aveva già intrapreso la strada giusta verso l'obiettivo che ci si era prefissati all'inizio del progetto, ma vi era ancora molto lontana. Il software, infatti, presentava ancora molti problemi, quali, ad esempio, quello riguardante la simulazione che a volte non terminava mai, facendo entrare il programma in un *loop* infinito, quello riguardante i servomotori del robot, la cui simulazione non rispecchiava esattamente ciò che in realtà il robot avrebbe fatto con quel dato programma in ingresso, o l'errata schedulazione dei clump [1s] di un programma. Oltre ai problemi appena citati, *NXTSimulator* presentava, inoltre, molte lacune funzionali: esso, infatti, non era ancora stato dotato delle funzionalità necessarie per simulare molti componenti del robot LEGO®, quali ad esempio il display, il riproduttore di suoni, i pulsanti del brick, od il gestore di file.

L'obiettivo che ci si è dati in questa attività di tesi è stato quindi quello di rimediare a tutte le problematiche citate sopra, in modo da ottenere una nuova versione del simulatore, ossia la 0.9c, non ancora definitiva (in quanto ancora mancante di alcune funzionalità), ma sicuramente molto più vicina a quello che era l'obiettivo iniziale del progetto *NXTSimulator*, prefissatosi nel 2008: ottenere un simulatore del robot funzionale in tutto e per tutto.

## 1.1 Progetto TERECoP

TERECoP (Teacher Education on Robotics-Enhanced Constructivist Pedagogical Methods) è un progetto didattico internazionale partito nell'ottobre dell'anno 2006, nel quale è inserito anche il Dipartimento di Ingegneria dell'Informazione dell'Università di Padova; esso ha come tema fondante l'utilizzo della robotica, della scienza e della tecnologia nell'ambito dell'educazione. Nella fattispecie 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 ai propri allievi. Questo progetto prende ispirazione dalle *teorie costruttiviste* dello psicologo e pedagogista svizzero Jean Piaget e dalla *filosofia didattica costruzionista* del matematico sudafricano Seymour Papert.

Le teorie di Piaget sostengono che l'apprendimento non sia tanto il risultato di un passaggio di conoscenze, ma un processo attivo di costruzione della conoscenza basato su esperienze empiriche ricavate dal mondo reale e collegate a preconoscenze uniche e personali (Piaget, 1972).

La filosofia di Papert, invece, introduce l'idea che il processo di apprendimento risulti decisamente più efficace qualora vengano introdotti artefatti cognitivi, ovvero oggetti e dispositivi che si basino su concetti familiari allo studente. Il Costruzionismo (Papert, 1992) è quindi una naturale estensione in chiave più moderna del Costruttivismo, la quale enfatizza l'aspetto pratico dell'apprendimento. In un ambiente costruzionista gli studenti vengono messi in grado di realizzare da soli oggetti tangibili e significativi. L'obiettivo del Costruzionismo è quello di fornire agli studenti dei buoni strumenti, in modo tale che possano imparare facendo meglio di quanto potessero fare prima (Papert, 1980).

Sulla base delle teorie appena brevemente esposte, l'intento finale del progetto è quindi quello di implementare il metodo costruttivista e costruzionista non solo appunto nelle classi di studenti, ma anche nell'educazione dei futuri insegnanti, attraverso l'utilizzo di specifici strumenti tecnologici per la creazione di diversi percorsi formativi al passo con l'innovazione scientifica e tecnologica dei nostri tempi. Tenendo, infatti, in considerazione che lo studente ottiene una migliore comprensione se si esprime attraverso inventiva e creatività (Piaget, 1974), gli insegnanti devono essere in grado di fornirgli l'opportunità di progettare, costruire e

programmare i propri modelli cognitivi. 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).

Il robot della LEGO® associa in qualche modo la tecnologia alle idee del Costruzionismo. Il sistema Lego® Mindstorms® NXT è, infatti, uno strumento flessibile per l'apprendimento costruzionista, offrendo l'opportunità di progettare e costruire strutture robotiche con tempo e fondi limitati. Esso è composto da materiale di montaggio (mattoncini, ruote e dispositivi vari) e da un software di programmazione, che offre una comoda interfaccia grafica iconica per controllare il comportamento del robot. Queste strutture programmabili rendono possibili nuovi tipi di esperimenti scientifici, grazie ai quali lo studente può comprendere attraverso l'esperimento pratico i fenomeni fisici della vita quotidiana (sia in classe che fuori) [8s].

E' nel contesto appena delineato che si è resa necessaria la realizzazione di un simulatore (*NXTSimulator*) del robot LEGO®, per la realizzazione del quale si è impegnato il Dipartimento di Ingegneria dell'Informazione dell'Università di Padova. Una volta realizzato un programma per il robot, infatti, per testarlo e rilevare eventuali bug in esso presenti, era necessario disporre del robot medesimo, caricarvi il programma, e farlo eseguire. Con l'avvento di *NXTSimulator*, invece, si può comodamente testare il comportamento di un programma seduti davanti al proprio PC, ed osservandone l'esecuzione al simulatore. In tal modo l'utente può così correggere eventuali bug presenti nel programma sviluppato, prima di caricarlo sul robot. Il simulatore non è considerato come un'alternativa al robot, in quanto esso non è in grado di riprodurre tutte le variabili del mondo reale, con cui il robot ha a che fare, ma semplicemente un utile strumento di supporto nella fase di test dei programmi.


**Figura 1.1:** *Logo TERECoP*

## 1.2 Strumenti utilizzati

Gli strumenti utilizzati per realizzare i miglioramenti apportati ad *NXTSimulator* sono molteplici: per la parte di programmazione sono stati utilizzati il linguaggio *Java* e l'ambiente di sviluppo gratuito *NetBeans*, mentre, per la parte di test ed analisi del codice dei programmi del robot, il linguaggio grafico *NXT-G*, quello simile al C *NXC*, quello simile ad assembly [14s] *NBC*, ed il programma *BricxCC*.

Di seguito viene riportata una breve descrizione dei suddetti strumenti.

### 1.2.1 Java

Java è un linguaggio di programmazione orientato agli oggetti utilizzabile gratuitamente (licenza GNU General Public License) e di proprietà di Oracle®. Esso venne ideato da James Gosling, Patrick Naughton e da altri ingegneri dall'azienda americana Sun Microsystems© (acquisita poi da Oracle® nel gennaio del 2010). La piattaforma di programmazione Java è fondata sul linguaggio stesso, sulla Java Virtual Machine (JVM) e sulle API (Application Programming Interface). La sintassi prende spunto da quella del C++ (e quindi indirettamente dal C), ma a differenza di questi due Java consente di creare programmi eseguibili su molte piattaforme, grazie all'utilizzo della JVM citata sopra: il codice prodotto in seguito alla compilazione di un programma scritto in Java non è, infatti, specifico per la macchina nella quale lo si è compilato, ma è un codice intermedio (detto *bytecode*) che può essere interpretato ed eseguito su qualsiasi macchina nella quale sia installata una JVM [1t]. Quello appena descritto, la semplicità del linguaggio stesso, ed il suo orientamento agli oggetti costituiscono i motivi per cui lo si è scelto come linguaggio per implementare *NXTSimulator*.

Per lo sviluppo del software è stata utilizzata la versione 1.6 di Java, ossia la più recente e presente nella maggior parte delle piattaforme in circolazione. L'eseguibile è stato sviluppato su piattaforma Windows (Vista®), ma è stato testato anche su piattaforme Linux e Macintosh, senza riscontrare alcun genere di problema, confermando così una volta di più le potenzialità e soprattutto la portabilità del codice scritto in Java.

### 1.2.2 NetBeans

NetBeans è un ambiente di sviluppo multi-linguaggio gratuito scritto interamente in Java e nato nel giugno del 2000. E' l'ambiente scelto da Sun Microsystems© come IDE (Integrated Development Environment) ufficiale per lo sviluppo di applicazioni in Java. Sono disponibili numerosi plug-in per arricchirne la sua versione standard, i quali lo rendono molto appetibile al pubblico, e richiede 512 [MB] di RAM per essere eseguito [5s].

Le versioni utilizzate per la realizzazione di *NXTSimulator* sono state molteplici durante il ciclo di vita del programma, in particolare per questo elaborato è stata utilizzata la versione più recente disponibile al momento, ossia la 6.9.1.

NetBeans offre un ambiente completo per lo sviluppo di applicazioni e si è rivelato un ottimo strumento, soprattutto per la sua facilità di creazione e manipolazione di oggetti grafici, per la presenza di utili funzionalità per il debugging, e per la possibilità di costruire con facilità un programma eseguibile completo di tutti i file, e delle librerie da esso utilizzate.

### 1.2.3 NXT-G

NXT-G è un IDE basato sulla tecnologia LabVIEW™, celebre ambiente di sviluppo grafico elaborato da National Instruments [6s], appositamente modificato per rispettare le specifiche del firmware NXT [1s]. L'intuitiva interfaccia grafica permette la creazione di semplici programmi per il robot LEGO® in pochissimi minuti, grazie all'utilizzo di elementi grafici che identificano i diversi componenti del robot stesso e le diverse azioni da esso eseguibili.

Questo strumento è stato utilizzato per la creazione di programmi per il test del simulatore.

Ulteriori dettagli su NXT-G saranno forniti nel capitolo seguente.

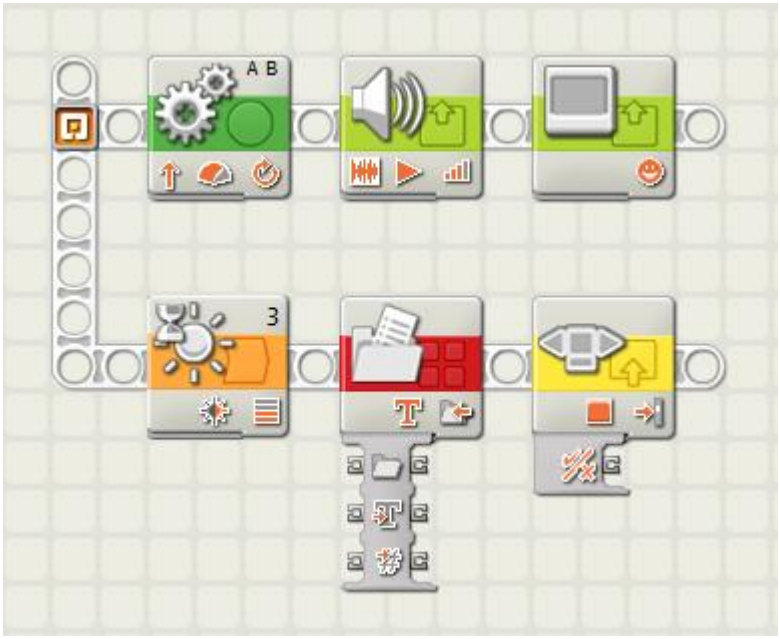


**Figura 1.2:** *Esempio di programma NXT-G*

### 1.2.4 NXC

NXC (Not eXactly C) è un linguaggio di programmazione simile al C come sintassi, ma sviluppato solo per la creazione di programmi per il robot Lego® Mindstorms® NXT [7s]. Esso mette a disposizione dell'utilizzatore numerose funzioni predefinite, che agiscono sui vari dispositivi del robot, quali, ad esempio, i motori ed i sensori. Questo linguaggio ad alto livello sfrutta il compilatore NBC per produrre file eseguibili dal robot LEGO®.

Questo linguaggio è stato utilizzato per la creazione di programmi per il test del simulatore.

### 1.2.5 NBC

Next Byte Codes (NBC) è un semplice linguaggio con una sintassi molto simile ad assembly [14s], che può essere utilizzato per sviluppare programmi eseguibili dal brick Lego® Mindstorms® NXT [7s]. Nel corso dello sviluppo del simulatore non si è tuttavia programmato mediante tale linguaggio, ma ci si è limitati a convertire programmi, sviluppati e compilati mediante NXT-G o NXC, in codice NBC (mediante il programma descritto nella sottosezione successiva): operando questa conversione si ottiene, infatti, un codice a basso livello, facile da analizzare per individuare e studiare i meccanismi con cui lavora il firmware Lego® NXT.

```
; -------------- program code --------------
  thread t000
  not ub004C, ub0048
  mov sw0042, ub004C
  sub ub004C, ub004A, ub004D
  sub ub004F, ub004B, ub004E
  mov uw0043, sw0042
  acquire m0006
  mov sl0003, sl0000
  mov ub0050, ub0049
  mov sl0004, sl0001
  mov sl0005, sl0002
  mov ub0051, ub004C
  mov ub0052, ub004F
  subcall t007, sl0007
```

**Figura 1.3:** *Estratto di codice NBC*

### 1.2.6 Bricx Command Center (BricxCC)

Bricx Command Center (BricxCC) è un programma per piattaforme Windows comunemente noto come un IDE per sviluppare programmi per il brick Lego® Mindstorms® NXT, utilizzando i già citati linguaggi NXC e NBC. Come già detto nella sottosezione precedente, tale programma è stato molto utile per aprire programmi realizzati e compilati mediante NXT-G o NXC e convertirli in automatico nel formato NBC corrispondente, potendoli così poi analizzare. I file compilati (in formato *.RXE*, del quale si discuterà nel capitolo 3 di questo elaborato) possono così essere tradotti in un linguaggio simile ad assembly, in modo da poter visionare le singole istruzioni del firmware LEGO®, invocate all'interno dei programmi per il robot.

## 1.3 Prerequisiti necessari alla comprensione

I requisiti di cui il lettore deve essere in possesso, per avere una perfetta comprensione degli argomenti trattati in questo elaborato, sono essenzialmente i seguenti:

- buona conoscenza della sintassi e del funzionamento di un linguaggio di programmazione orientato agli oggetti (nella fattispecie Java), soprattutto nel contesto di utilizzo di un IDE come NetBeans;
- conoscenza di base di alcune delle librerie maggiormente utilizzate nel linguaggio Java, come ad esempio quelle riguardanti i *Thread* od il *Timer*;
- rappresentazione dei numeri in base binaria ed esadecimale e conseguenti operazioni su di essi, conversione da base binaria a decimale e viceversa, e

conversione da base esadecimale a decimale;

- rappresentazione a mantissa ed esponente dei numeri decimali (*floating-point*) in un calcolatore;
- nozioni di base sulla rappresentazione delle informazioni e sul funzionamento dei calcolatori.

Il lettore in possesso dei requisiti sopra presentati possiede tutti gli strumenti necessari per apprezzare in pieno il lavoro svolto durante l'attività di tesi, nonché i motivi di certe scelte implementative, piuttosto che di altre.

Tuttavia, anche un lettore non molto esperto nei temi sopra riportati potrà cogliere i frutti del lavoro svolto, grazie alle spiegazioni dettagliate che saranno fornite a corredo di ogni scelta implementativa effettuata.

# Capitolo 2

# Lego® Mindstorms® NXT

## 2.1 Il kit Lego® Mindstorms® NXT

Il kit Lego® Mindstorms® NXT contiene il brick NXT, quattro tipi diversi di sensori (luce, suono, tocco, ed ultrasuoni), tre servomotori interattivi, sette cavi per collegare i motori ed i sensori al brick, e più di seicento pezzi Lego Technic da utilizzare per la costruzione del proprio robot NXT [1s]. Di seguito viene proposta una breve panoramica sulle caratteristiche dei componenti di cui poi si discuterà nel corso della tesi.

### 2.1.1 Brick NXT

Il brick è il componente principale del robot, e lo si può considerare come il suo vero e proprio "cervello": vi si possono, infatti, caricare dei programmi, che esso si occuperà poi di interpretare e far eseguire al robot. Ciò è reso possibile grazie alla presenza di un firmware (la cui versione più recente è la 1.29), all'interno del brick stesso, composto da numerosi moduli quali, ad esempio, quelli per i sensori o per i motori, ed una virtual machine (VM) [1s]. Per ulteriori dettagli a riguardo consultare il capitolo 3 di questo elaborato.

Il brick ha quattro porte di ingresso per collegarvi i sensori, tre porte di uscita adibite al collegamento dei servomotori, una porta USB ed un'interfaccia Bluetooth, queste ultime entrambe concepite per il trasferimento dati tra un PC ed il brick stesso. L'interfaccia Bluetooth può, però, essere anche utilizzata per controllare il robot da remoto o per inviare e ricevere messaggi da altri dispositivi.

Vengono di seguito elencate precisamente le specifiche tecniche del brick [2s]:

- processore a 32 bit Atmel AT91SAM7S256 (classe ARM7) a 48 [MHz];
- coprocessore a 8 bit Atmel ATmega48 (classe AVR) a 8 [MHz];
- memoria Flash da 256 [KB];
- memoria RAM da 64 [KB];
- 4 porte di input;
- 3 porte di output;
- interfaccia Bluetooth v2.0+EDR, velocità teorica massima 0,46 [Mbit/sec];
- display LCD in bianco e nero da 100x64 [pixel] (ogni pixel è circa 0,4×0,4 [mm]);
- altoparlante mono a 8 bit fino a 16 [KHz] per la riproduzione di suoni;
- tastiera con 4 tasti in gomma;
- porta USB 2.0;
- alimentazione: 6 batterie AA (1,5 [V]), oppure tramite batteria ricaricabile al litio.

**Figura 2.1:** *Brick NXT*

E' importante, infine, segnalare come sia possibile espandere il numero di sensori collegabili al brick, mediante l'utilizzo di alcuni moduli esterni [3s].

### 2.1.2 Servomotori

I servomotori vengono installati nel robot per permettergli di muoversi. Essi sono collegati alle porte dei sensori di output del brick. Questi dispositivi si distinguono da dei semplici motori, in quanto, a differenza di quest'ultimi, devono possedere bassa inerzia, linearità di coppia e velocità, rotazione uniforme e capacità di sopportare picchi di potenza. Ciascun servomotore possiede, inoltre, al suo interno un sensore di rotazione che permette all'utente di avere un controllo molto accurato sui movimenti

del robot, consentendo di tracciare la posizione dell'asse esterno del motore in gradi o in rotazioni complete (con incertezza di circa un grado). Un giro completo corrisponde ad una rotazione di 360°. Notare come la possibilità di controllare accuratamente i movimenti di un servomotore permetta di sincronizzare più motori, in modo che si muovano alla stessa velocità [3s].

**Figura 2.2:** *Servomotore*

### 2.1.3 Sensori

I sensori consentono al robot di raccogliere, interpretare, ed eventualmente reagire ad informazioni provenienti dall'ambiente ad esso circostante. I sensori presenti all'interno del kit Mindstorms® ed utilizzabili all'interno del simulatore sono di quattro tipi distinti e saranno di seguito brevemente descritti. Si possono, però, acquistare altre tipologie di sensori da integrare nel robot, a seconda delle proprie necessità: questi sono ad esempio l'accelerometro, la bussola, il sensore di temperatura, etc. [3s][4s].

#### Sensore di luce

Il sensore di luce permette al robot di percepire il livello di intensità di luce presente in una stanza, o la luminosità di una superficie rischiarata dalla luce emessa dal led rosso di cui è dotato il sensore stesso. Essendo il sensore monocromatico, esso non può percepire tutti i colori, ma li rileva in una scala di grigi. Pertanto, dopo aver effettuato una lettura, esso restituisce in output un valore che può variare nell'intervallo tra 0 e 100, dove lo 0 corrisponde ad una situazione di buio completo, mentre il 100 alla maggiore intensità di luce che il sensore riesce a rilevare.

**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 con due diverse unità di misura: *dB* e *dBA*, dove *dBA* rappresenta la risposta in frequenza dell'orecchio umano ai suoni, quindi in uno spettro di frequenze ben preciso. Una volta percepito un suono di una data intensità, questo sensore restituisce in output un valore percentuale che va dal 4-5% (rumore all'interno di un soggiorno silenzioso) al 30-100% (persone che gridano, o musica ad alto volume).

**Figura 2.4:** *Sensore di suono*

### Sensore di tocco

Il sensore di tocco è dotato di un singolo bottone, che può assumere soltanto tre stati diversi: premuto, rilasciato, o "*bumped*". Quest'ultimo stato è assunto quando il bottone viene premuto e successivamente rilasciato in rapida successione, e può essere pertanto paragonato ad una sorta di "click" del mouse.

**Figura 2.5:** *Sensore di tocco*

### Sensore ad ultrasuoni

Il sensore ad ultrasuoni è in grado di simulare (assieme a quello di luce) la "vista" da parte del robot. Tale dispositivo permette di misurare la distanza di un oggetto solido posto davanti ad esso, ed è in grado di calcolare distanze che vanno da 0 a 255 centimetri con un'incertezza di +/- 3 [cm]. Per adempiere a questa funzionalità esso utilizza gli ultrasuoni, ossia emula quello che è il comportamento dei pipistrelli: misura la distanza calcolando il tempo impiegato da un'onda sonora a colpire un oggetto e ritornare come un eco.

**Figura 2.6:** *Sensore ad ultrasuoni*

## 2.2 Linguaggi utilizzati per programmare NXT Brick

I linguaggi che sono stati utilizzati nel corso dell'aggiornamento di *NXTSimulator* per sviluppare programmi di test per il brick del robot LEGO®, e di cui si è già brevemente discusso nella sezione 1.2 di questo elaborato, sono stati essenzialmente i seguenti: NXT-G e NXC.

Per far comprendere meglio al lettore l'utilità dei suddetti programmi, viene prima riportato il *modus operandi* che si è praticato durante tutto il processo di aggiornamento e test delle funzionalità effettuato sul simulatore, e riassumibile nei punti seguenti:

- implementazione *ex novo* o modifica mediante NetBeans di una funzionalità fornita dal simulatore (e.g: funzione di simulazione del display del robot);
- realizzazione mediante NXT-G o NXC di un programma di test che vada a richiamare la funzionalità del simulatore di cui al punto precedente;
- conversione del programma prodotto nella forma di file eseguibile ed interpretabile dal simulatore, ossia in formato *.RXE* (consultare il capitolo 3 di questo elaborato per un trattamento approfondito di questo tipo di file);
- caricamento dell'eseguibile ottenuto su *NXTSimulator* e verifica della correttezza del risultato dato in uscita dalla simulazione;
- in caso di esito soddisfacente del test effettuato (e di tutti i molteplici altri ad esso seguenti), la funzionalità sotto esame si considera come definitivamente realizzata; in caso contrario si procede col punto seguente;
- conversione del file eseguibile dal formato *.RXE* al formato *.NBC*, mediante l'apertura dello stesso col software *BricxCC* (di cui si è parlato nella sottosezione 1.2.6 di questo elaborato);

- analisi passo passo delle istruzioni di basso livello contenute nel file *.NBC*, di cui al punto precedente, così da riuscire a capire quale istruzione o blocco di codice specifico della funzionalità in esame (o di qualcun'altra da essa invocata) causa il malfunzionamento ravvisato, ed operare poi le dovute modifiche correttive ai responsabili.

Viene ora presentata una descrizione dettagliata dei programmi citati sopra: NXT-G e NXC.

### 2.2.1 NXT-G (LEGO MINDSTORMS Education NXT Programming 2.0)

Come già accennato nel capitolo precedente, NXT-G è un linguaggio grafico per programmare il robot Lego® Mindstorms® NXT. L'ambiente di sviluppo grafico contempla il paradigma della cosiddetta *Graphic Language*, uno stile di programmazione contraddistinto dall'assenza di codice scritto sotto forma di testo. La definizione di ogni componente di un programma avviene, infatti, tramite icone ed oggetti grafici (detti anche blocchi), che talvolta possono richiedere la specifica di alcuni parametri in forma testuale, mentre l'eventuale scambio di informazioni tra i vari blocchi avviene semplicemente mediante delle linee di collegamento tracciate tra di essi.

Vediamo ora nel dettaglio quali sono le caratteristiche principali di NXT-G.

#### Componenti di un Programma

Come rappresentato nella figura 1.2 a pagina 16, un programma sviluppato con NXT-G non è altro che una semplice sequenza di blocchi, ciascuno dei quali svolge una determinata funzione. All'interno di LEGO MINDSTORMS Education i suddetti blocchi sono suddivisi in sei categorie per favorirne all'occorrenza la reperibilità al programmatore:

- *blocchi comuni*: questa categoria contiene i blocchi più comunemente utilizzati, come ad esempio quello di movimento, del display, del riproduttore di suoni, o quello che permette di ripetere ciclicamente una sequenza di blocchi di un programma;
- *blocchi di azione*: questa classe include dei blocchi che riguardano delle

azioni vere e proprie che si desidera far compiere al robot. Tra questi si annoverano, ad esempio, il blocco che invia un messaggio via Bluetooth, o ancora quelli relativi al movimento o alla visualizzazione di qualcosa sul display;

- *blocchi dei sensori*: questa categoria racchiude i blocchi relativi a tutti i sensori di input di cui si può dotare il robot. Alcuni di questi sensori sono, ad esempio, quello di suono, di luce o di temperatura;
- *blocchi di flusso*: questo insieme contiene, come suggerisce il nome stesso, dei blocchi che permettono di imporre un controllo di flusso sull'esecuzione di un programma; tra i blocchi che appartengono a questa categoria si trova, ad esempio, quello che impone al programma di arrestarsi, o quello di "*switch*", il quale permette di scegliere quale strada intraprendere tra due flussi distinti sulla base di una certa condizione (per ulteriori delucidazioni su questo costrutto consultare la sottosezione successiva);
- *blocchi di dati*: questa classe include dei blocchi che permettono di effettuare delle operazioni tra dati di vario tipo di un programma. Alcuni dei blocchi presenti in questa categoria sono, ad esempio, quello che permette di effettuare varie operazioni matematiche su uno o due operandi, e quello che esegue un'operazione logica tra due valori booleani;
- *blocchi avanzati*: quest'ultimo insieme racchiude alcuni blocchi che eseguono funzionalità avanzate e non racchiuse in alcuna delle categorie precedenti. Di questo gruppo fanno parte, ad esempio, il blocco che permette la gestione di file di testo, o quello per la conversione di un numero in stringa e viceversa.

E' importante citare, infine, come, per ciascuno dei blocchi di tutte le categorie appena presentate, sia possibile imporre, sempre mediante interfaccia grafica, alcuni parametri con cui si desidera venga eseguito (per il blocco display, ad esempio, si può specificare cosa si vuole visualizzare, ed il punto preciso del display in cui visualizzarlo).

**Struttura di un Programma**

Come già ribadito in precedenza, un programma NXT-G è costituito da un insieme di blocchi che eseguono determinate azioni. Un programma può anche presentare più flussi (sequenze di blocchi) paralleli che eseguono in concorrenza (come accade nella

figura 1.2 di pagina 16): ciò può tornare utile quando, ad esempio, si desidera che il robot si muova in una certa direzione e, in concomitanza, si desidera controllare l'azione di un braccio. Un programma può, inoltre, eseguire un flusso piuttosto che un altro sulla base del verificarsi o meno di una certa condizione, che può essere dipendente dalla lettura dei sensori, utilizzando il blocco *switch*, citato nella sottosezione precedente e rappresentato in figura 2.7.

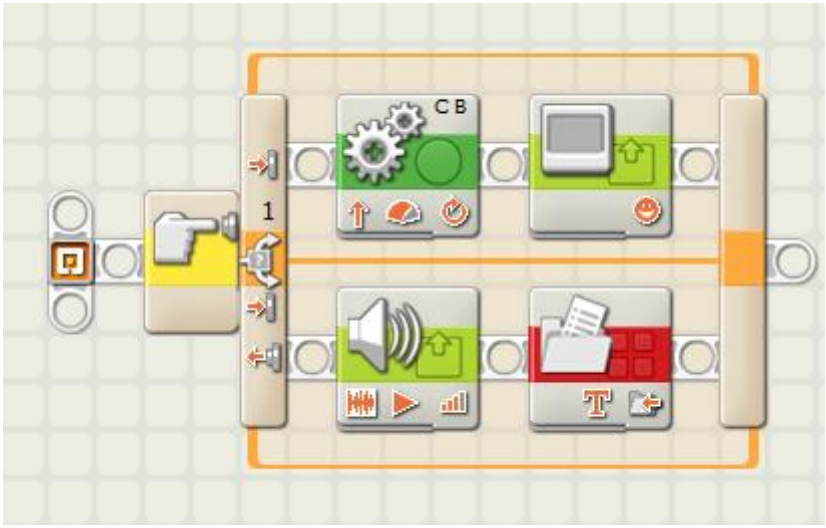


**Figura 2.7:** *Costrutto "switch" all'interno di programma NXT-G*

**Compilazione di un Programma**

Una volta realizzato il programma desiderato mediante NXT-G, di default è possibile salvarlo nel PC solo nel formato proprietario LEGO® con estensione *.RBT*, ossia come file consultabile soltanto mediante lo stesso NXT-G, includente tutte le caratteristiche grafiche impostate durante la "programmazione". Per poter, invece, salvare il programma nella forma di file eseguibile (con estensione *.RXE*, i cui dettagli saranno trattati nel capitolo successivo), interpretabile anche da *NXTSimulator*, si deve ricorrere ad un plug-in di terze parti. Tale plug-in consente di aggiungere una voce, denominata "*Download To File...*", al menù *Strumenti* di NXT-G, premendo la quale si origina il file *.RXE* compilato senza, per ottenerlo, dover prima caricare e poi recuperare il programma compilato direttamente dal robot tramite cavo USB o connessione Bluetooth.

Per installare il plug-in, di cui sopra, è sufficiente copiare il file *DownloadToFile.llb* (presente all'interno della distribuzione di *NXTSimulator*) nel percorso seguente della cartella di installazione di NXT-G del proprio sistema: *..\LEGO MINDSTORMS Edu NXT\engine\project*.

### 2.2.2 NXC

NXC (acronimo di Not eXactly C) è un linguaggio di programmazione ad alto livello per il robot Lego® Mindstorms® NXT e, come suggerisce il nome, presenta una sintassi simile al C. Esso è comodamente utilizzabile all'interno dell'ambiente di sviluppo Bricx Command Center (di cui si è già parlato nella sottosezione 1.2.6 di questo elaborato) e, per chi possiede un'esperienza di programmazione di base, rappresenta la scelta ideale per programmare eseguibili per il brick NXT. Esso, infatti, offre diverse comode funzioni predefinite che operano, ad esempio, sui vari sensori o motori del robot.

Questo linguaggio sfrutta il compilatore NBC (di cui si discuterà nella sezione successiva) per produrre file eseguibili dal robot LEGO® [7s].

```
t = CurrentTick() + loopTime;
while (t > CurrentTick())
{
  // read the light sensor value
  GetRawValue(IN_3, SV);

  // set speed for motor 1
  SetMotorSpeed(OUT_A, LT, Threshold2, SpeedFast, SpeedSlow);

  // set speed for motor 2
  SetMotorSpeed(OUT_B, GT, Threshold1, SpeedFast, SpeedSlow);

  // display sensor value
  DisplayNum(SV);

  // increment loop count
  Inc(LoopCount);
}
```

**Figura 2.8:** *Estratto di codice NXC*

## 2.3 Altri Linguaggi disponibili

Esistono anche altri linguaggi che possono essere utilizzati per programmare il robot Lego® Mindstorms® NXT. Di seguito vengono brevemente presentati quelli maggiormente utilizzati.

### 2.3.1 NBC

Come già anticipato nel capitolo precedente, NBC (il cui acronimo sta per *NeXT Byte Codes*) è un linguaggio a basso livello stile assembly [14s] (come si può notare anche dall'estratto di codice nella figura 1.3 di pagina 17), creato appositamente per

sviluppare programmi per il brick NXT. Esso possiede i numerosi vantaggi di un linguaggio di programmazione, come ad esempio la possibilità di definire variabili, salti condizionati e cicli; tuttavia è necessario essere familiari con un linguaggio sullo stile di assembly, per essere in grado di utilizzarlo, o quantomeno di comprenderlo. I programmi per il robot Lego® Mindstorms® NXT scritti con questo linguaggio risultano essere molto lunghi e densi di istruzioni e, per questa ragione, non è molto agevole utilizzarlo come linguaggio per programmare.

### 2.3.2 Microsoft® Robotics Developer Studio

Microsoft® Robotic Developer Studio (Microsoft RDS) è un ambiente di sviluppo per piattaforme Windows, che permette la creazione di programmi per svariate tipologie di robot. Questo software può essere utilizzato anche per sviluppare programmi per il robot LEGO®, ma, non essendo specifico per questo tipo di hardware, il lavoro del programmatore non risulta propriamente agevole. Per la creazione di un programma è, infatti, necessario accollarsi l'onere di fornire una descrizione accurata di ogni singolo sensore collegato al brick NXT, rendendo di fatto la programmazione un lavoro soltanto per utenti esperti. Microsoft RDS comunica col brick tramite Bluetooth, rendendo così possibile il caricamento dei programmi nella memoria del robot, nonché il controllo remoto dello stesso tramite un'interfaccia web creata ad-hoc.

Questo ambiente di sviluppo permette sia la programmazione testuale, tramite i linguaggi *C#* e *VisualBasic.NET*, che visuale (in maniera analoga a NXT-G), tramite *Microsoft Visual Programming Language* [9s].

### 2.3.3 leJOS

leJOS (nome nato dalla fusione delle due lettere iniziali della parola LEGO® con l'acronimo di *Java Operating System*) è un firmware sostitutivo per il brick del robot Lego® Mindstorms® NXT. Esso include una Java Virtual Machine, che permette di programmare il robot LEGO® mediante l'utilizzo del linguaggio di programmazione Java. leJOS risulta quindi in grado di eseguire programmi scritti in Java, tramite i quali si possono comandare i motori ed i sensori del robot [10s].

Per utilizzare leJOS, com'è facilmente intuibile, è tuttavia necessario rimuovere dal brick del robot il firmware NXT originario, perdendone così le funzionalità e le

proprietà peculiari. Questa caratteristica fa sì che leJOS non sia il contesto più adatto nel quale operare nel nostro caso, in quanto *NXTSimulator* si basa esclusivamente sul firmware originale del robot Lego® Mindstorms® NXT.

### 2.3.4 RobotC

RobotC è un IDE (Integrated Development Environment) concepito per l'utilizzo da parte degli studenti, usato per programmare e controllare anche il robot Lego® Mindstorms® NXT, nonché altri tipi di robot (come ad esempio il VEX [11s]), utilizzando un linguaggio di programmazione basato sul C. Questo ambiente di sviluppo è stato concepito con lo scopo di permettere la portabilità del codice da una piattaforma robotica all'altra, necessitando soltanto di pochissimi cambiamenti.

Tra gli aspetti positivi di RobotC si possono annoverare la disponibilità di molte più funzioni rispetto a quelle presenti in NXT-G, la possibilità di creare agevolmente programmi molto complessi, e la presenza di un'efficiente strumento di debugging dei programmi sviluppati [12s]. Nonostante tutte le caratteristiche positive appena descritte, si è deciso di preferire NXT-G e NXC a RobotC, in quanto specifici per il robot LEGO® e decisamente più semplici da utilizzare per la creazione di semplici programmi da utilizzare come test durante lo sviluppo di *NXTSimulator*.

# Capitolo 3

# Specifiche File Eseguibile Lego® Mindstorms® NXT

In questo capitolo verranno presentate le specifiche del file eseguibile *.RXE*, ossia il file compilato (a partire da un programma di partenza) contenente il *bytecode* delle istruzioni che il robot LEGO® deve eseguire, cioè quelle che *NXTSimulator* interpreta per poi simulare il comportamento del robot stesso con quel determinato codice in input.

Per ulteriori approfondimenti sull'argomento si rimanda al documento *LEGO® MINDSTORMS® NXT Executable File Specification*, consultabile dal sito specificato al punto [1s] della bibliografia.

## 3.1 Introduzione all'argomento

Come già anticipato nel capitolo precedente, il brick del robot LEGO® è dotato di una virtual machine (VM), un modulo software in grado di interpretare il programma compilato che viene dato in ingresso al robot, e di eseguire le azioni in esso specificate. Quando la VM esegue un programma, essa legge il file *.RXE* caricato nella memoria flash del brick, ed inizializza un *pool* di memoria RAM pari a 32 [KB], la quale viene riservata per l'utilizzo da parte dei programmi caricati dall'utente. Il file *.RXE* si incarica di specificare il "layout" ed il contenuto iniziale di questo *pool* di memoria. Una volta che quest'ultimo è stato inizializzato, il programma è considerato *attivo*, ossia pronto per l'esecuzione. Com'è facile intuire, inoltre, molte delle istruzioni del programma in questione potranno comportare una modifica dei dati contenuti nel *pool* di memoria RAM in questione.

### 3.1.1 Istruzioni del *Bytecode*

Le istruzioni presenti nel *bytecode* rappresentano la parte principale di un programma. La VM, infatti, interpreta tali istruzioni per poi agire direttamente sui dati salvati in RAM ed effettuare le eventuali operazioni di I/O (input/output) sul brick del robot. Queste istruzioni si possono classificare in sei categorie distinte:

- *Matematiche*: include operazioni matematiche di base, come l'addizione, la sottrazione od il calcolo della radice quadrata di un numero;
- *Logiche*: racchiude operazioni logiche di base tra operatori Booleani, come quelle di *AND* od *OR*, ad esempio;
- *Confronto*: comprende istruzioni per il confronto tra valori presenti in RAM, le quali producono come risultato un valore Booleano, determinato dall'esito del confronto;
- *Manipolazione dati*: presenta istruzioni per la copia, conversione o manipolazione di dati presenti in RAM;
- *Controllo di flusso*: contiene istruzioni che modificano il flusso di esecuzione di un programma, mediante, ad esempio, salti condizionati od invocazioni di subroutine;
- *System I/O*: raggruppa istruzioni che permettono al brick di interfacciarsi con i dispositivi di I/O ad esso collegati;

### 3.1.2 Esecuzione del *Bytecode*

Il *bytecode* delle istruzioni dei programmi è suddiviso in uno o più pezzi di codice. Questi pezzi di codice sono chiamati *clump*, e sono usati come subroutine per il multi-tasking. Ogni clump è costituito da una o più istruzioni facenti parte del *bytecode*.

La VM stabilisce *run-time* con che ordine schedulare i vari clump interdipendenti, e questa decisione è basata soltanto su informazioni contenute all'interno del file eseguibile. Esso, infatti, contiene dei cosiddetti *clump record*, uno per ciascun clump definito nel programma. Un clump record non fa altro che specificare quali istruzioni appartengono al clump cui esso si riferisce, lo "stato" del clump stesso, e la lista dei clump da esso "dipendenti", i quali devono essere eseguiti soltanto dopo che il

clump, cui il clump record si riferisce, è terminato. E' utile specificare che un clump è definito come *dipendente* da altri clump, qualora esso necessiti di dati manipolati da altri clump prima di poter essere eseguito. Un clump dipendente può quindi andare in esecuzione soltanto dopo che tutti i clump da cui esso dipende sono terminati. Notare, infine, come il firmware LEGO® consenta ad un clump di invocare un altro clump durante la sua esecuzione, ma non consenta ad un clump di effettuare invocazioni a sé stesso: non è pertanto ammessa la *ricorsione*.

### 3.1.3 Stato di esecuzione di un Programma

Quando un utente avvia sul brick del robot l'esecuzione di un programma, la virtual machine si accolla l'onere di compiere sequenzialmente le quattro fasi seguenti:

- *Validazione*: lettura del file eseguibile, validazione della versione e di altri contenuti dell'*header* del file (per approfondimenti sull'header consultare la sottosezione 3.2.1);
- *Attivazione*: allocazione ed inizializzazione delle strutture dati in RAM;
- *Esecuzione*: interpretazione delle istruzioni del *bytecode* contenute nel file *.RXE*, utilizzando le informazioni sull'ordine di schedulazione per decidere il clump da mandare di volta in volta in esecuzione. Questa fase prosegue sino a che non sono stati eseguiti tutti i clump, o sino a che l'utente non impone un arresto del programma;
- *Disattivazione*: reinizializzazione di tutti i dispositivi di I/O e delle strutture dati in RAM, e rilascio dell'accesso al programma appena eseguito.

E' importante notare che, anche a *run-time*, la VM non carica mai le istruzioni del *bytecode* in RAM, ma le esegue direttamente dalla memoria flash dove è contenuto il file *.RXE* da processare. Questo *modus operandi* garantisce così l'utilizzo soltanto di una porzione di RAM relativamente esigua, da utilizzare solamente per i dati che possono subire dei cambiamenti *run-time*. Le istruzioni del *bytecode*, infatti, non possono subire questo genere di cambiamenti, giacché non variano mai.

### 3.1.4 Dati *Run-time*

Durante l'esecuzione di un programma la virtual machine utilizza, come già accennato in precedenza, un *pool* di memoria RAM per salvarvi tutti i dati che il programma stesso utilizza. Questo *pool* contiene un segmento riservato per i dati dell'utente; tale segmento è chiamato *dataspace*, ed è strutturato come una collezione di record di un certo "tipo" (nella sottosezione successiva verrà discusso il significato del termine). Ogni record possiede un *entry* corrispondente all'interno della cosiddetta *dataspace table of contents* (DSTOC), la quale tiene traccia di tutti i tipi di dati contenuti nel dataspace. Tutte le istruzioni del *bytecode* fanno riferimento ai record del dataspace, indicizzandoli mediante gli entry della DSTOC. Questi indici sono denominati *dataspace ID*, e sono utilizzati dalla VM per trovare ed operare sui dati in RAM. Come per il *bytecode* delle istruzioni, anche la DSTOC giace in memoria flash, dal momento che essa non cambia *run-time*.

### 3.1.5 Tipi di dati

La versione 1.28 del firmware NXT supporta i seguenti tipi di dati:

- *Interi*: scalari con o senza segno, aventi lunghezza possibile pari a 8, 16 o 32 bit e definiti rispettivamente come *byte*, *word* e *long*;
- *Numeri decimali*: numeri in virgola mobile a 32 bit, con precisione singola;
- *Array*: lista di zero o più elementi tutti del medesimo tipo;
- *Cluster*: collezione di tipi di dati diversi, paragonabile alle "strutture" del linguaggio C;
- *Mutex record*: struttura dati a 32 bit utilizzata per la gestione sicura in parallelo di risorse da parte di più clump.

I valori *Booleani* sono memorizzati sotto forma di byte senza segno, dove il valore *false* è rappresentato dallo '0', mentre il valore *true* da tutti gli altri valori diversi da '0'.

Le *stringhe* testuali sono, invece, un caso speciale di array. Una stringa è, infatti, considerata come un array di byte senza segno (un byte, ossia il rispettivo codice ASCII [15s], per ogni carattere della stringa), con un byte extra aggiunto alla fine. Tale byte terminatore è il valore nullo.

### 3.1.6 Dati Statici e Dati Dinamici

Come già detto in precedenza nel corso di questo capitolo, la DSTOC non cambia durante l'esecuzione di un programma. Ciò significa che tutti i tipi di dati, comprese le dimensioni iniziali degli array, sono già completamente specificati al momento della compilazione, ossia alla nascita del file *.RXE*. Appena si procede con la fase di *attivazione* di un programma, la VM inizializza tutti i dati utente ai loro valori di default, i quali sono specificati nel file *.RXE*. Questi dati utente sono suddivisi in due categorie: *statici* e *dinamici*.

I dati *statici* sono quelli che la virtual machine non può spostare a *run-time*. Tutti i dati sono considerati statici, eccezion fatta per gli array, i quali sono definiti, invece, come dati *dinamici*. A *run-time*, infatti, la VM può ridimensionare o spostare gli array in locazioni differenti della RAM, in seguito ad operazioni del *bytecode* che lo impongono, o per altri fattori interni.

Dati statici e dinamici sono memorizzati in due sotto-bacini separati di memoria all'interno del dataspace. Quelli statici sono sempre posizionati ad un indirizzo di memoria più basso rispetto a quelli dinamici, e le loro rispettive porzioni di memoria non si sovrappongono mai.

### 3.1.7 Gestione dei Dati Dinamici

La memorizzazione e la gestione dei dati dinamici (array) differisce da quella dei dati statici. Nel corso della compilazione di un programma per il robot LEGO® vengono definiti tutti gli entry della DSTOC ed i valori di default per i dati sia statici che dinamici. Il compilatore porta a termine questo compito semplicemente estrapolando tali dati dal file eseguibile. Quando il programma entra nella fase di *attivazione* il firmware utilizza un cosiddetto *memory manager* per gestire i dati dinamici.

Il memory manager utilizza uno schema di allocazione per tener traccia degli array, ed eventualmente ridimensionarli, all'interno del *pool* di memoria riservato ai dati dinamici. Dopo che i dati statici sono stati piazzati nella RAM, la VM riserva il rimanente spazio (dei 32 [KB] iniziali) per i dati dinamici. Il memory manager gestisce in automatico tutti i piazzamenti e i ridimensionamenti degli array, che avvengono durante l'esecuzione del programma, avvalendosi di strutture dati di supporto chiamate *dope vector.*

Un dope vector (DV) è una struttura dati che “descrive” un array presente in RAM. Ciascun array nello spazio di dati dinamici possiede un dope vector ad esso associato. I dope vector sono descrittori di dimensione fissa, composti ognuno da cinque campi. La seguente tabella illustra i suddetti campi .

| Campo | Descrizione |
|---|---|
| Offset | Offset dell'inizio dei dati dell'array in RAM, rispetto all'inizio dei dati utente. |
| Dimensione Elemento | Dimensione, in byte, di ciascun elemento dell'array. |
| Numero Elementi | Numero di elementi correntemente contenuti nell'array. |
| Back Pointer | Campo non utilizzato nel firmware 1.28. |
| Link Index | Indice del prossimo dope vector nella lista del memory manager. |

**Tabella 3.1:** *Struttura di un record di tipo Dope Vector*

Giacché la dimensione e la posizione degli array può variare durante l'esecuzione di un programma, il DV associato ad ogni array deve essere anch'esso passibile di cambiamenti. Per questo motivo l'insieme dei dope vector è posizionato in una porzione specifica della RAM, detta *dope vector array* (DVA). Tale struttura è salvata nello stesso *pool* di memoria degli array di dati dell'utente, e vanta le seguenti proprietà:

- il DVA è un oggetto singolo, il che significa che la memoria può contenerne uno ed un solo esemplare alla volta. Tale DVA contiene informazioni riguardanti tutti i dope vector in memoria;
- il primo entry del DVA (posizionato all'indice 0) è un dope vector che descrive il DVA in sé, ed è chiamato *root dope vector*;
- dal momento che il DVA è utilizzato unicamente per uso interno, ai fini della gestione della memoria, le istruzioni del *bytecode* dei programmi non fanno mai riferimento ad esso, né possono modificarlo;
- il memory manager tratta il DVA alla stregua di una lista concatenata, sfruttando per fare ciò il campo *Link Index* di ogni dope vector.

Il metodo di gestione della memoria appena presentato fornisce così un metodo per risolvere l'indirizzo in RAM di ogni dato array, necessitando soltanto di un indice nella DSTOC. Per risolvere l'indirizzo di un array in memoria, infatti, la VM consulta

la DSTOC per trovare poi un indice secondario, chiamato *DV index*. Essa utilizza poi tale DV index per cercare nel DVA il vero e proprio offset dell'array in RAM. I seguenti due passi descrivono il processo appena illustrato.

*indirizzo DV index = inizio dataspace + DSTOC[dataspace entry ID].offset*

*indirizzo array = inizio dataspace + DVA[DV index].offset*

## 3.2 Formato del file eseguibile (.*RXE*)

In questa sezione vengono trattati nel dettaglio il formato e le specifiche dei file eseguibili (con estensione .*RXE*) che possono essere interpretati ed eseguiti mediante il firmware 1.28 del brick.

I file .*RXE* sono essenzialmente suddivisi in quattro segmenti principali, come riportato nella tabella rappresentata qui di seguito.

| Segmento | Dimensione [B] | Descrizione |
|---|---|---|
| File Header | 38 | Specifica il contenuto del file. |
| Dataspace | Variabile | Descrive i tipi di dati ed i valori di default per ogni dato definito nel programma. |
| Clump Records | Variabile | Descrive come devono essere schedulati i clump a *run-time*. |
| Codespace | Variabile | Contiene tutte le istruzioni del *bytecode*. |

**Tabelle 3.2:** *Struttura di un file eseguibile*

Verrà ora illustrato con cura ciascuno dei quattro segmenti brevemente descritti nella tabella soprastante, e dai quali è composto un file eseguibile.

### 3.2.1 Header

I primi 38 byte di tutti i file .*RXE* sono occupati dal segmento denominato *header*, il quale descrive la composizione di tutto il resto del file. Tale segmento è composto, a sua volta, da quattro campi separati. La tabella a pagina seguente descrive proprio i campi appena citati.

| Dimensione [B] | Campo | Descrizione |
|---|---|---|
| 16 | Stringa di formato | Stringa di formato e numero di versione.<br>Per il firmware 1.28 questa stringa deve contenere la parola 'MindstormsNXT', seguita da un byte nullo di padding, e poi 0x0005, ossia il numero di versione del file supportato con i byte disposti secondo l'ordine big-endian [13s].<br>In altre parole, tutti i file *.RXE* supportati dal firmware 1.28 iniziano con i seguenti 16 byte (in codice esadecimale):<br>4D 69 6E 64 73 74 6F 72 6D 73 4E 58 54 00 00 05 |
| 18 | Dataspace Header | Sotto-segmento che descrive la dimensione e la disposizione dei dati statici e dinamici nel file eseguibile. |
| 2 | Clump Count | *Word* senza segno da 16 bit che specifica il numero di clump nel file.<br>Il firmware 1.28 permette un numero massimo di 255 clump per programma. |
| 2 | Code Word Count | *Word* senza segno da 16 bit che specifica il numero di *word* di istruzioni totali da cui è composto il *bytecode*. |

**Tabella 3.3:** *Struttura dell'Header del file eseguibile*

Tra i campi presentati qui sopra ve ne è uno che merita un'analisi più approfondita, in modo da poter così illustrare nel dettaglio tutti i sotto-campi da cui esso è a sua volta composto; tale campo è il *Dataspace Header*.

### Dataspace Header

Il *Dataspace Header* è un sotto-segmento dell'header che tiene conto del numero, della dimensione e della sistemazione degli oggetti del dataspace di un programma. Questo segmento inizia sempre al sedicesimo byte dell'header ed è composto da nove word (da 16 bit) senza segno. La tabella a pagina seguente descrive questi word nell'ordine in cui essi compaiono nel file *.RXE*.

| Campo | Descrizione |
|---|---|
| Count | Numero di record nella DSTOC.<br>Il firmware 1.28 limita il numero di record nella DSTOC a 16.383. |
| Initial Size | Dimensione iniziale (in byte) del dataspace in RAM, comprendente sia i dati statici che quelli dinamici. |
| Static Size | Dimensione (in byte) del segmento di dati statici del dataspace in RAM. |
| Default Data Size | Dimensione (in byte) del segmento del dataspace contenente i valori di default per i dati sia statici che dinamici. |
| Dynamic Default Offset | Offset (in byte) dell'inizio dei dati di default dinamici, rispetto all'inizio di tutti i valori di default del file. |
| Dynamic Default Size | Dimensione (in byte) dei valori di default dinamici. Questo valore è sempre pari alla differenza tra il campo *Default Data Size* e *Dynamic Default Offset*. |
| Memory Manager Head | Indice nel DVA che punta al primo elemento della lista concatenata dei dope vector utilizzata dal memory manager. |
| Memory Manager Tail | Indice nel DVA che punta all'ultimo elemento della lista concatenata dei dope vector utilizzata dal memory manager. |
| Dope Vector Offset | Offset (in byte) della locazione di memoria iniziale dei DV in RAM, rispetto all'inizio del *pool* di memoria del dataspace in RAM. |

**Tabella 3.4:** *Struttura del Dataspace Header*

### 3.2.2 Dataspace

Il segmento *Dataspace* del file eseguibile include tutte le informazioni necessarie per inizializzare il dataspace durante la fase di *attivazione* di un programma, nonché le specifiche di tutti i tipi di dati e della loro disposizione, le quali sono poi necessarie durante l'*esecuzione* del programma stesso. Tale segmento è suddiviso, a sua volta, in tre sotto-segmenti, i quali sono presentati nella tabella seguente.

| Sotto-segmento | Descrizione |
|---|---|
| DSTOC | Specifica dei tipi di dati e della loro posizione nel dataspace. |
| Dati di default statici | Valori iniziali per i dati di tipo statico. |
| Dati di default dinamici | Valori iniziali per i dati di tipo dinamico. |

**Tabella 3.5:** *Struttura del Dataspace*

**Dataspace Table of Contents (DSTOC)**

La DSTOC descrive i tipi di dati utilizzati durante l'esecuzione del programma e la loro collocazione all'interno del dataspace del programma stesso. Tale struttura può essere definita come la “mappa” che viene utilizzata dalla virtual machine del firmware per allocare i dati in RAM, ed i suoi campi sono utilizzati come argomenti delle istruzioni che operano sui dati. Per ulteriori dettagli sulle istruzioni consultare la sottosezione 3.2.4 di questo elaborato.

E' utile ricordare ancora una volta che la DSTOC è costruita al momento della compilazione di un programma e che essa non cambia durante l'esecuzione dello stesso.

La DSTOC è organizzata come un array i cui elementi sono dei record di lunghezza fissa pari a 4 byte ed esemplificati nella tabella seguente.

***DSTOC Record***

| **Campo** | Tipo | Flags | Data Descriptor |
|---|---|---|---|
| **Bit** | 0..7 | 8..15 | 16..31 |

**Tabella 3.6:** *Struttura di un record della DSTOC*

Nella tabella soprastante il campo relativo al *Tipo* contiene un semplice codice numerico intero; i valori assumibili da tale campo sono i seguenti:

- **0**: *TC_VOID* - usato per gli elementi non utilizzati nel codice;
- **1**: *TC_UBYTE* - numero intero di 8 bit senza segno;
- **2**: *TC_SBYTE* - numero intero di 8 bit con segno;
- **3**: *TC_UWORD* - numero intero di 16 bit senza segno;
- **4**: *TC_SWORD* - numero intero di 16 bit con segno;
- **5**: *TC_ULONG* - numero intero di 32 bit senza segno;
- **6**: *TC_SLONG* - numero intero di 32 bit con segno;
- **7**: *TC_ARRAY* - array di elementi di un qualsiasi tipo;
- **8**: *TC_CLUSTER* - struttura dati composta da sottotipi diversi;
- **9**: *TC_MUTEX* - dato mutex per la schedulazione dei clump;
- **10**: *TC_FLOAT* - numero decimale (in virgola mobile) di 32 bit con precisione singola.

Il campo *Flags* è utilizzato soltanto al momento dell'inizializzazione di un programma, ed il suo scopo è quello di segnalare se il dato a cui il record si riferisce possiede o meno un valore di default da assegnarvi. Se il campo assume il valore '1' il dato in questione è inizializzato al valore nullo; se, invece, esso assume valore '0' significa che è presente tra i dati di default un valore da assegnare a tale dato.

Il campo *Data Descriptor* può, invece, assumere diversi significati a seconda del tipo di dato descritto dal record della DSTOC.

**Grammatica della DSTOC**

La DSTOC adotta una specifica "grammatica" con regole ben precise per descrivere tutti i possibili tipi di dati. Tale grammatica segue un approccio *top-down*, il che significa che i tipi di dati più complessi sono definiti per mezzo di una lista ordinata di diversi record della DSTOC. Di questa categoria di dati fanno parte gli array ed i cluster; per definire ciascuno di essi, infatti, la DSTOC si serve di un primo record per indicare il tipo di dato (*TC_ARRAY* o *TC_CLUSTER*) e di altri record per specificare i tipi di dati in esso contenuti.

Le regole che vengono esposte di seguito aiutano a comprendere meglio l'approccio appena citato.

- I dati di tipo *numerico* necessitano di un solo record nella DSTOC. Nel loro caso il campo Data Descriptor contiene un offset utile al fine di recuperare il valore del dato in RAM;

  *e.g.*: record per dato numerico: *TC_SWORD 0x00 0x0000*

- I dati di tipo *array* hanno bisogno di due (caso di array di numeri) o più record (caso di array di cluster) nella DSTOC. Il campo Data Descriptor del primo record rappresenta un offset utile per reperire il dope vector che descrive l'array medesimo. I record successivi specificano, invece, la tipologia dei dati contenuti nell'array;

  *e.g.*: record per array di byte: *TC_ARRAY 0x00 0x0200*

  *TC_BYTE 0x00 0x0000*

- I dati di tipo *cluster* necessitano di due o più record nella DSTOC (dipende dal numero di oggetti da cui essi sono composti). Il campo Data Descriptor del primo record comunica il numero di oggetti contenuti nel cluster. I record

successivi illustrano il tipo di ogni oggetto da cui il cluster è composto.

*e.g.*: record per cluster contenente due numeri ed un array di numeri:

*TC_CLUSTER 0x00 0x0300*

*TC_ULONG 0x00 0x0C00*

*TC_ULONG 0x00 0x1000*

*TC_ARRAY 0x00 0x1400*

*TC_SWORD 0x00 0x0000*

Come un lettore perspicace avrà già intuito, le regole appena presentate possono essere impiegate ricorsivamente, così da poter definire, ad esempio, array di cluster, o cluster contenenti array; di quest'ultimo caso ve ne è un'occorrenza nell'esempio allegato all'ultima regola esposta.

E' importante precisare che i Data Descriptor, tanto dei dati numerici quanto degli array, contengono come offset un numero senza segno da 2 byte relativo ai dati presenti in RAM, ma esso viene interpretato in maniera differente nei due casi. Gli offset delle variabili e degli array sono relativi all'inizio del dataspace in RAM, e per questo motivo sono chiamati *dataspace offset*. Gli offset che, invece, sono successivi al record che definisce un array sono relativi all'inizio dell'array in RAM, e per questa ragione sono detti *array data offset*. Questa distinzione viene fatta per rendere possibile lo spostamento od il ridimensionamento degli array in RAM.

**Dati di default statici**

I valori di default per i dati statici sono posizionati subito dopo la DSTOC, e la composizione di questo sotto-segmento dipende interamente dai record contenuti nella DSTOC stessa. Tali valori sono memorizzati nel medesimo ordine in cui i loro record corrispondenti sono elencati nella DSTOC.

E' importante ricordare che, se il valore del campo *Flags* di un record della DSTOC è pari ad '1', la porzione di RAM riservata all'oggetto descritto da quel record è automaticamente inizializzata al valore '0'. Conseguenza diretta di questo *modus operandi* è il fatto che la dimensione del sotto-segmento dei dati di default statici risulta essere minore, od al massimo uguale, a quella specificata dal campo *Static Size* del *Dataspace Header* (di cui si è già discusso nella sottosezione 3.2.1).

**Dati di default dinamici**

I valori di default per i dati dinamici sono gestiti in maniera differente rispetto a quelli per i dati statici. E' utile innanzitutto ricordare che i dati dinamici sono costituiti unicamente da array. Un'altra differenza importante è che i dati di default dinamici sono una copia perfetta di tutti i valori iniziali del *pool* di dati dinamici del dataspace. In altre parole, i valori di default dinamici devono essere “formattati” in modo tale da poterli copiare direttamente nel segmento di RAM riservato ai dati dinamici, senza dovervi apportare alcuna modifica.

E' importante ricordare che anche il DVA è esso stesso un array, ed è quindi salvato tra i dati dinamici. Ogni dope vector nel DVA ha una valore di default, ed anche questi valori risiedono nello spazio riservato ai dati di default dinamici. Per semplicità i cosiddetti “default dope vector” sono posizionati all'inizio dei dati di default dinamici.

### 3.2.3 Clump Record

Il segmento *Clump Record* del file eseguibile ha il compito di specificare come sono suddivise le istruzioni presenti all'interno del codespace, ossia descrive i clump (insiemi di istruzioni correlate) che compongono un programma ed in che modo essi debbono essere schedulati a *run-time*. Il numero totale di clump è precisato dal campo *Clump Count* dell'header (di cui si è discusso nella sottosezione 3.2.1).

La tabella allegata qui di seguito descrive i campi da cui è composto ogni record presente nel segmento che si sta illustrando.

| Campo | Dimensione [B] | Descrizione |
|---|---|---|
| Fire Count | 1 | Byte senza segno che specifica quando il clump è pronto per essere eseguito. |
| Dependent Count | 1 | Byte senza segno che specifica il numero di clump del file eseguibile che sono dipendenti da questo clump. |
| Code Start Offset | 2 | Word da 16 bit senza segno che specifica l'offset nel codespace al quale iniziano le istruzioni di questo clump. |
| Dependent List | Variabile | Array di byte senza segno che specifica gli indici di tutti i clump che dipendono da questo clump. |

**Tabella 3.7:** *Struttura di un Clump Record*

Come si evince anche dalla tabella riportata a pagina precedente, ogni clump record è formato da una parte iniziale di lunghezza complessiva pari a 4 byte, e da una parte finale di lunghezza variabile; quest'ultima coincide col campo *Dependent List* del record. Durante la fase di compilazione di un programma i campi *Dependent List* di ciascun clump record (qualora la rispettiva lista delle dipendenze sia non vuota ovviamente) sono raggruppati in un sotto-segmento separato del file eseguibile, posizionato subito dopo la parte di lunghezza fissa (di cui sopra) di tutti i clump record. Questo *modus operandi* è stato adottato per mantenere allineati in memoria gli spezzoni di record di lunghezza fissa, riducendo così lo spreco di spazio in memoria.

E' importante far notare che in un certo istante possono essere eseguiti più clump di uno stesso programma in parallelo, ma è necessario che non vi siano dipendenze tra di loro, e che tutti i clump, da cui essi eventualmente dipendono, abbiano terminato la loro esecuzione.

### 3.2.4 Codespace

Il segmento *Codespace* del file eseguibile è composto da parole di codice, o code word (word da 16 bit ciascuna), che sono interpretate come istruzioni di lunghezza variabile (un'istruzione può essere, infatti, composta da una o più parole di codice). Esso raggruppa quindi tutte le istruzioni da cui è composto un programma. Per tali istruzioni sono disponibili due diverse codifiche: quella *long* e quella *short*. Verrà ora fornita una descrizione di entrambe le codifiche. E' importante precisare che la rappresentazione che verrà utilizzata per i byte delle parole di codice sarà di tipo little-endian, in accordo con la modalità con cui tali byte sono salvati nel file *.RXE*. Con questa rappresentazione i singoli bit sono identificati da sinistra verso destra, ossia il bit 0 è il bit più significativo del byte meno significativo [13s].

Il bit 12 del primo code word di ogni istruzione ha il compito di identificare il tipo di codifica utilizzato per l'istruzione stessa: se tale bit vale '0' la codifica è di tipo long, altrimenti è di tipo short.

I campi delle istruzioni che specificano gli operandi, sui quali esse stesse agiscono, per la maggior parte dei casi sono degli indici che puntano a record della DSTOC, consultando la quale è poi possibile individuare la locazione di memoria precisa dove l'operando risiede. Solo in alcuni casi il valore di un operando è specificato direttamente nel campo dell'istruzione che lo utilizza (in questo caso si parla di

operando dal valore *immediato*).

**Codifica Long delle istruzioni**

Questa tipologia di codifica è quella più semplice ed, infatti, la maggior parte delle istruzioni supporta soltanto tale codifica. Il primo code word di un'istruzione contiene nell'ordine i campi *opcode*, *size* e *flags*. Il secondo e gli eventuali code word seguenti costituiscono, invece, gli operandi dell'istruzione stessa. La tabella riportata di seguito descrive la struttura di un'istruzione appartenente alla categoria in esame.

| | | ***Word 1*** | | ***Word 2*** | ***Word 3*** |
|---|---|---|---|---|---|
| **Campo** | Opcode | Size | Flags | Argument1 | Argument2 |
| **Bit** | 0..7 | 8..11 | 12..15 | 0..15 | 0..15 |

**Tabella 3.8:** *Struttura di un'istruzione con codifica Long*

Il campo *opcode* del *Word 1* è un byte senza segno che identifica univocamente l'operazione che deve essere eseguita.

Il campo *size* per la maggior parte delle istruzioni specifica la dimensione totale delle stesse in numero di byte, comprensiva del primo code word e di tutti quelli contenenti gli operandi. Tuttavia alcune istruzioni (di dimensione più ampia) presentano un campo size pari al valore '14' (decimale), e ciò significa che la dimensione totale dell'istruzione è specificata dal primo operando successivo al code word iniziale dell'istruzione stessa.

Il campo *flags* (nella fattispecie i suoi bit 13, 14 e 15) è utilizzato soltanto dalle istruzioni che effettuano un confronto tra due valori: in questo caso tale campo specifica il tipo di confronto che si desidera venga effettuato.

*Syscall (Chiamate a Sistema)*

Esiste una particolare famiglia di istruzioni che utilizzano la codifica long appena esposta: quella delle cosiddette *syscall*, o chiamate a sistema. Il firmware 1.28 raggruppa ben quarantasette esemplari di tali chiamate. Esse non sono altro che istruzioni con codifica long e la seguente struttura: campo opcode con codice decimale pari a '40' (identificante appunto tale famiglia di istruzioni, ossia *OP_SYSCALL*), e due argomenti. Il primo è un valore cosiddetto “immediato” (il cui significato lo si è visto in precedenza), che contiene il codice numerico (*SysCallID*)

che identifica quale chiamata a sistema della famiglia effettuare. Il secondo contiene, invece, il puntatore ad un record della DSTOC, il quale permette poi di risalire in memoria al cluster contenente la lista di parametri di cui la syscall necessita per essere eseguita.

All'interno di tale famiglia di istruzioni si possono individuare ulteriori importanti sotto-famiglie, quali, ad esempio, quella delle syscall relative al display, quelle relative al riproduttore di suoni, al gestore di file, o ai bottoni del brick. Ciascuna di queste sotto-famiglie fornisce le funzionalità di cui il firmware deve disporre per far eseguire al robot programmi che agiscono sui rispettivi componenti.

**Codifica Short delle istruzioni**

Alcune tra le istruzioni più comuni utilizzano una codifica alternativa a quella long, per risparmiare spazio di memoria: quella short. L'utilizzo o meno di questa tipologia di codifica dipende dalla capacità del compilatore di allestire il dataspace in modo tale che una delle seguenti due condizioni sia verificata:

- l'unico operando di cui necessita un'istruzione può essere contenuto in un singolo byte;
- il primo operando di un'istruzione a due operandi può essere identificato tramite un offset (espresso come un byte con segno) relativo all'indirizzo del secondo operando dell'istruzione stessa.

Per ciascuna istruzione che soddisfa una delle condizioni appena illustrate, il primo code word di tale istruzione può essere riorganizzato nel modo seguente: il campo *size* assume lo stesso significato del caso di codifica long, ma i campi *flags* e *opcode* sono utilizzati in maniera differente. Il bit 12 è ovviamente impostato al valore '1' (codifica short), mentre i rimanenti tre bit del campo flags sono adoperati per specificare il codice dell'istruzione da eseguire (da ciò ne consegue che le operazioni di confronto non possono essere ottimizzate con questa codifica). Il campo opcode è rimpiazzato da un byte che specifica un argomento dell'istruzione, il quale viene interpretato in maniera diversa, a seconda della variante di codifica short adottata.

Per istruzioni con un solo argomento la codifica short si esaurisce in un unico code word. La tabella a pagina seguente descrive la struttura di questo code word.

***Word 1***

| **Campo** | Argument | Size | 1 | Op |
|---|---|---|---|---|
| **Bit** | 0..7 | 8..11 | 12 | 13..15 |

**Tabella 3.9:** *Struttura di un'istruzione con codifica Short ed un solo argomento*

Nella variante riportata sopra il campo *argument* rappresenta l'unico operando di cui l'istruzione necessita.

Per istruzioni con due argomenti, invece, la codifica short assume la forma schematizzata nella tabella che segue.

***Word 1*** ***Word 2***

| **Campo** | Offset | Size | 1 | Op | Argument 2 |
|---|---|---|---|---|---|
| **Bit** | 0..7 | 8..11 | 12 | 13..15 | 0..15 |

**Tabella 3.10:** *Struttura di un'istruzione con codifica Short e due argomenti*

In questo caso il secondo operando dell'istruzione è specificato direttamente dal campo *argument 2* del secondo code word, mentre il primo è ottenuto grazie alla semplice equazione seguente:

*argument 1 (primo operando) = argument 2 (secondo operando) + offset*

E' utile ricordare che il campo offset è un byte con segno, il che significa che il valore di *argument 1* deve essere compreso in un intervallo di indirizzi compreso tra +127 e -128 rispetto a quello di *argument 2*.

# Capitolo 4

# Aggiornamenti al Simulatore

In questo capitolo verranno discussi nel dettaglio tutti gli aggiornamenti che sono stati effettuati durante il lavoro di tesi sul simulatore per il robot Lego® Mindstorms® NXT. Tali aggiornamenti riguardano l'implementazione di funzionalità aggiuntive di cui *NXTSimulator* era ancora sprovvisto.

Nel capitolo successivo si illustreranno, invece, tutti i bug rilevati nella versione 0.9b del simulatore ed il modo in cui essi sono stati corretti.

Viene ora proposta una breve descrizione di *NXTSimulator*, così da fornire al lettore un'infarinatura sulla struttura di questo software, in modo da rendere più agevole la comprensione delle modifiche apportatevi, e che saranno descritte nel corso del capitolo. Per ulteriori informazioni sulle funzionalità e sull'utilizzo del simulatore, si rimanda al *Manuale Utente* dello stesso, riportato alla fine di questo elaborato.

## 4.1 *NXTSimulator*

Come già detto più volte nel corso di questa tesi, *NXTSimulator* è un software sviluppato dal Dipartimento di Ingegneria dell'Informazione dell'Università di Padova per simulare su di un calcolatore programmi realizzati per il robot Lego® Mindstorms® NXT.

La versione del simulatore precedente all'attività di tesi descritta in questo elaborato, ossia la 0.9b, all'avvio del software presentava all'utente la schermata raffigurata nell'immagine [2t] riportata a pagina seguente.

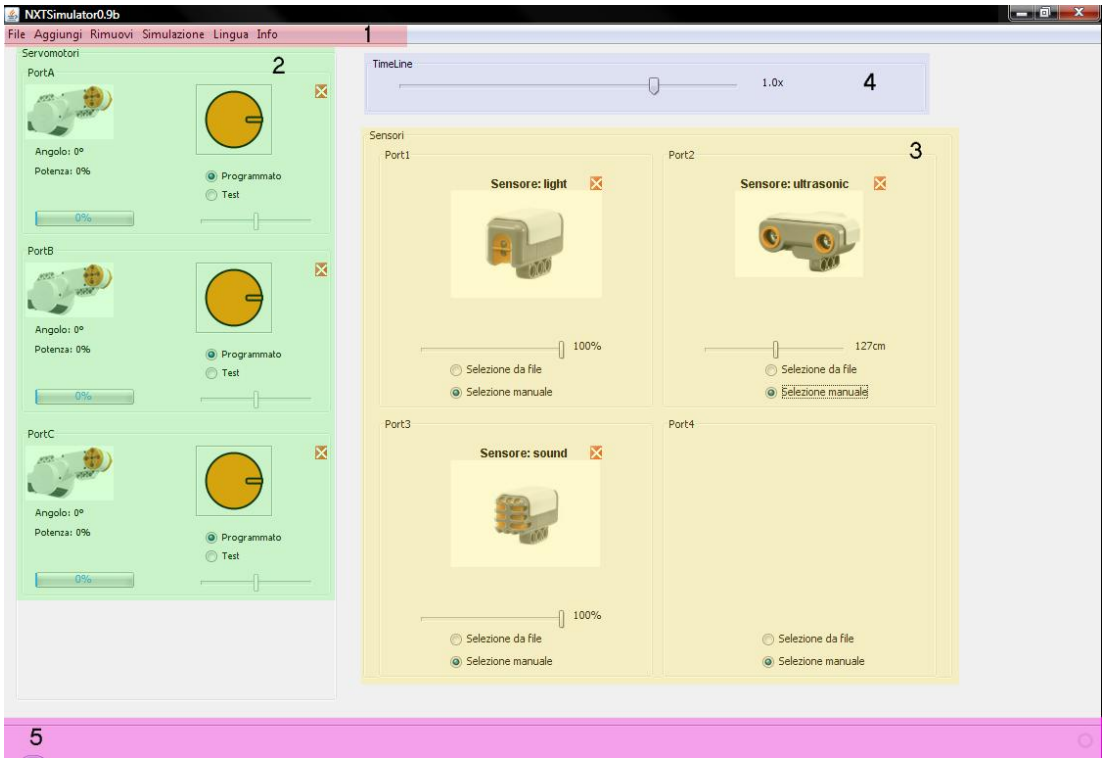


**Figura 4.1:** *NXTSimulator 0.9b*

Come si può notare, la schermata iniziale del simulatore sopra rappresentata è suddivisa essenzialmente in cinque parti, i cui ruoli sono i seguenti:

- **1**: *menù contestuali* tramite i quali l'utente può accedere a tutte le funzionalità del simulatore;
- **2**: *pannello dei motori* nel quale l'utente può allestire la propria configurazione delle porte di output del robot da simulare;
- **3**: *pannello dei sensori* nel quale l'utente può allestire la propria configurazione delle porte di input del robot da simulare;
- **4**: *TimeLine* per specificare la scala temporale con cui eseguire la simulazione;
- **5**: *barra di stato* dove vengono visualizzati messaggi di servizio.

Tramite l'interfaccia grafica di figura 4.1, l'utente può quindi “costruire” la configurazione desiderata per il proprio robot mediante qualche semplice click del mouse nelle opportune voci di menù, caricare su *NXTSimulator* il programma (in formato *.RXE*) che si desidera simulare, ed avviare, infine, la simulazione stessa.

## 4.2 Modifica layout del Simulatore

L'aggiornamento del simulatore alla versione 0.9c ha previsto innanzitutto la modifica dell'interfaccia grafica dello stesso, per aggiungervi dei componenti di cui era ancora sprovvista. Il risultato ottenuto è esemplificato nella figura che segue.

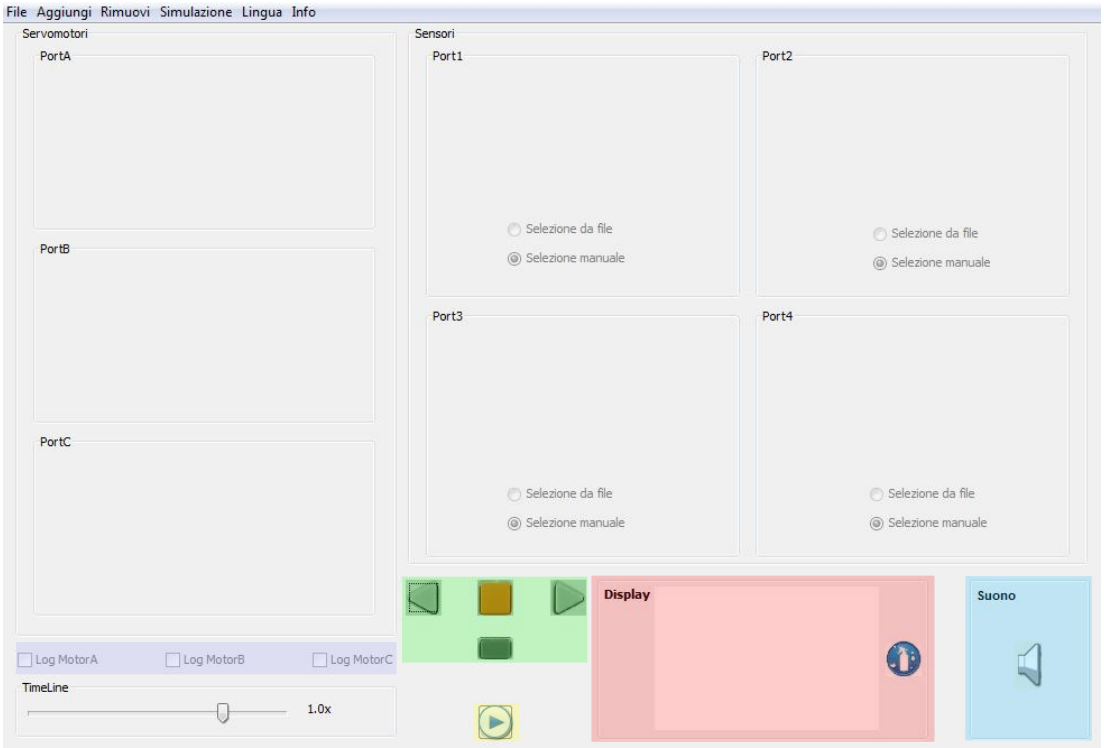


**Figura 4.2:** *NXTSimulator 0.9c*

Come si intuisce dalla figura sopra riportata, il processo di modifica del layout di *NXTSimulator* ha previsto essenzialmente una risistemazione più attenta degli oggetti già presenti nel pannello principale dello stesso (pannello dei motori, pannello dei sensori e TimeLine), in modo da ricavare lo spazio necessario per aggiungervi i cinque nuovi componenti richiesti: display, riproduttore di suoni, bottoni del brick, pulsante di avvio/arresto rapido della simulazione e caselle per scegliere se produrre o meno dei file di log del movimento dei servomotori. Per fare ciò si è intervenuti sulla classe, istanza di *javax.swing.JPanel* [16s], *NXTSView.java* del progetto del simulatore mediante NetBeans. Tramite la funzionalità dell'IDE, che permette di accedere a classi di quel tipo in modalità *Design* [5s], così da poterne modificare agevolmente l'interfaccia dell'oggetto di tipo JPanel da esse creato, con qualche semplice click del mouse si è ottenuto quanto rappresentato in figura 4.2. Nelle sezioni che seguono verranno presentati i dettagli implementativi dei nuovi componenti di cui sopra.

## 4.3 Risoluzione del Simulatore

La versione 0.9b del simulatore presentava l'interfaccia grafica esemplificata nella figura 4.1 di pagina 50, la cui risoluzione era di 800x670 [pixel]. Nella versione 0.9c sono state apportate al layout di *NXTSimulator* le modifiche di cui si è discusso nella sezione precedente di questo elaborato, e la risoluzione è stata aumentata a 1024x768 [pixel]. In seguito a questa modifica si è deciso che era opportuno visualizzare, all'avvio del simulatore, un messaggio che comunicasse la risoluzione idonea alla quale l'utente deve impostare il proprio calcolatore, per poter visualizzare correttamente l'interfaccia di *NXTSimulator*. Nella fattispecie si è optato per far apparire il messaggio rappresentato nella figura sottostante: esso compare appena si lancia il simulatore e scompare in automatico dopo tre secondi, se non lo si chiude prima.

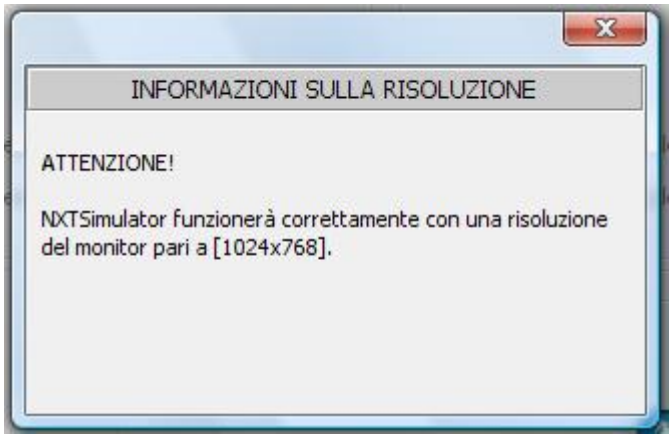


**Figura 4.3:** *Comunicazione risoluzione simulatore*

Per ottenere quanto appena descritto, si sono dovute modificare la classe *NXTSView.java*, che, come già ribadito nel corso di questo capitolo, definisce l'interfaccia grafica del simulatore e la classe *NXTSApp.java*, ossia la classe principale del software, che è responsabile del lancio dell'interfaccia medesima di *NXTSimulator*. E' stata, inoltre, definita una nuova classe denominata *ResolutionBox.java*. Quest'ultima è un'istanza della classe Java *javax.swing.JDialog* [16s] e non fa altro che definire il box con il messaggio riguardante la risoluzione, visualizzato in figura 4.3. All'interno della classe NXTSView è stata definita un'occorrenza di tale classe, denominata *resolutionBox*, ed un metodo chiamato *showResolutionBox()*, il quale ha semplicemente il compito di visualizzare il box di figura 4.3, referenziato da *resolutionBox*, ed il cui codice sorgente è riportato a pagina seguente.

```
protected static void showResolutionBox() {
    JFrame mainFrame = NXTSApp.getApplication().getMainFrame();
    resolutionBox = new ResolutionBox(mainFrame,true);
    resolutionBox.setLocationRelativeTo(mainFrame);
    NXTSApp.getApplication().show(resolutionBox);
}
```

Il metodo sopra riportato viene invocato dall'interno della classe NXTSApp. Quest'ultima lancia dapprima l'interfaccia grafica del simulatore, esemplificata nella figura 4.2 di pagina 51, ed invoca poi il suddetto metodo, attivando, però, prima un timer temporale, istanza di classe *javax.swing.Timer* [16s], con durata di tre secondi, scaduti i quali il box con le informazioni sulla risoluzione viene fatto scomparire.

Segue il codice sorgente relativo alla parte di NXTSApp che svolge le operazioni appena illustrate.

```
show(new NXTSView(this)); //Visualizza interfaccia grafica del simulatore
int elapsedTime = 3000;
javax.swing.Timer t = new javax.swing.Timer(elapsedTime, new java.awt.event.ActionListener(){
    public void actionPerformed(java.awt.event.ActionEvent e){
        NXTSView.resolutionBox.setVisible(false); //Nasconde box con info sulla risoluzione
    }
});
t.setRepeats(false); //Timer viene eseguito una volta soltanto
t.start();
NXTSView.showResolutionBox(); //Visualizza box con info sulla risoluzione
```

## 4.4 Pulsante di Avvio/Arresto rapido della Simulazione

L'avvio e l'arresto di una simulazione nella versione 0.9b di *NXTSimulator* poteva avvenire attraverso la pressione degli appositi pulsanti presenti sotto la voce *Simulazione* del menù rappresentato in alto nella figura 4.1 di pagina 50. Nella versione 0.9c del simulatore si è deciso, invece, di rendere disponibile un ulteriore strumento per adempiere a tale compito. E' stato, infatti, aggiunto all'interfaccia principale di *NXTSimulator* un pulsante per l'avvio e l'arresto rapido di una simulazione (lo si può notare evidenziato in giallo nella figura 4.2 di pagina 51). L'aggiunta del suddetto pulsante si è resa necessaria per evitare all'utente di dover navigare nel menù del simulatore ogni volta che doveva avviare od interrompere una simulazione; con la soluzione adottata tale “rallentamento” è stato superato.

Quando il simulatore è stato avviato, ma non sta eseguendo alcuna simulazione, il pulsante in questione assume le sembianze rappresentate nella prima immagine della figura riportata qui sotto; qualora, invece, vi sia una simulazione in corso, esso assume quelle della seconda immagine della medesima figura.

**Figura 4.4:** *Pulsanti rispettivamente di avvio e arresto rapido di una simulazione*

Qualora sia stato caricato nel simulatore un file eseguibile in formato *.RXE*, e si desideri avviare la simulazione dello stesso, è sufficiente premere il pulsante in questione per ottenere quanto voluto. Durante l'esecuzione della stessa è, invece, possibile arrestarla in qualsiasi istante sempre premendo il medesimo pulsante.

L'implementazione del pulsante di avvio/arresto rapido di una simulazione è stata portata a termine modificando la classe *NXTSView.java* del progetto NetBeans di *NXTSimulator*.

Per quanto riguarda l'aspetto squisitamente grafico della realizzazione del pulsante, esso è stato ottenuto aggiungendo alla classe NXTSView un bottone, istanza di *javax.swing.JButton* [16s], denominato *playStopButton*, avente appunto come icone intercambiabili le immagini di figura 4.4.

Per quanto riguarda, invece, le funzionalità vere e proprie svolte dal suddetto bottone, esse sono state implementate aggiungendo il metodo *playStopButtonMouseClicked()* sempre alla classe NXTSView. Questo metodo è invocato ad ogni pressione del pulsante in analisi e svolge delle operazioni molto semplici. Se viene premuto in una circostanza nella quale è possibile avviare una simulazione, esso esegue le medesime istruzioni svolte dal metodo *startMIMouseReleased()* di classe NXTSView, ed avvia appunto la simulazione. Qualora, invece, esso venga premuto mentre vi è una simulazione in corso, viene invocato il metodo *stopSim()*, anch'esso di classe NXTSView, il quale arresta la simulazione in atto. Notare che sia il metodo *startMIMouseReleased()* che *stopSim()* erano già presenti nella versione 0.9b di *NXTSimulator*.

L'implementazione di questo pulsante di avvio/arresto rapido di una simulazione non ha quindi richiesto particolari sforzi per essere effettuata, ma il risultato ottenuto ha permesso una notevole velocizzazione delle operazioni di avvio e di arresto di una simulazione da parte dell'utente.

## 4.5 Implementazione Display

Il *display* è il dispositivo, integrato nel brick del robot, che permette a quest'ultimo di visualizzare diversi tipi di oggetti, ossia figure geometriche (cerchio, linea retta, punto e rettangolo), stringhe testuali, ed immagini in formato proprietario LEGO® (con estensione *.RIC* e di cui si discuterà in seguito). Come descritto nella sottosezione 2.1.1 di questo elaborato, il display del robot è un LCD in bianco e nero ed ha dimensioni pari a 100x64 [pixel]. Per permettere ad *NXTSimulator* di simulare tale dispositivo, come già accennato in precedenza, si è deciso di aggiungere alla classe *NXTSView.java* del progetto un pannello di tipo *javax.swing.JPanel* denominato *port8*. Quest'ultimo contiene al proprio interno un oggetto *javax.swing.JLabel* [16s] col nome 'Display', un'altra istanza di JPanel (con sfondo bianco), chiamata *monitor*, ed un pulsante, istanza di *javax.swing.JButton* [16s], il cui nome è *cleanDisplayButton*. Il display così ottenuto nel simulatore è rappresentato nel riquadro rosso della figura 4.2 di pagina 51.

Il componente fondamentale di *port8* è l'oggetto *monitor*: esso rappresenta il vero e proprio display del robot, ed è al suo interno, infatti, che verranno visualizzati gli oggetti che un programma dato in ingresso ad *NXTSimulator* desidera tracciare. Si è scelto di creare tale pannello con dimensioni pari a 200x128 [pixel], ossia il doppio rispetto a quelle del display originale del robot, così da renderne più agevole la visione del contenuto da parte dell'utente. Il pulsante *cleanDisplayButton* è utilizzabile, invece, come suggerisce già il nome, per “pulire” *monitor* da eventuali oggetti in esso presenti. Si è scelto di inserire tale pulsante per consentire all'utente di ripulire all'occorrenza il display al termine di una simulazione, prima di iniziarne un'altra.

Ora che si è descritto l'aspetto squisitamente grafico della realizzazione del display, si può passare alla discussione riguardante lo sviluppo delle funzioni che hanno reso possibile l'implementazione vera e propria delle funzionalità del display su *NXTSimulator*.

Il processo che ha permesso di dotare il simulatore delle funzionalità citate sopra ha comportato modifiche alla classe del progetto *Execution.java* e alla già citata *NXTSView.java*; si sono dovute, inoltre, introdurre due nuove classi aggiuntive: *DisplayCondition.java* e *DisplayDrawings.java*.

La classe Execution è utilizzata da *NXTSimulator* per interpretare ed eseguire le

istruzioni presenti nel codespace del file eseguibile fornitogli in ingresso. Un'istanza di tale classe processa ed esegue ad una ad una tutte le istruzioni di un clump in esecuzione in un dato istante [2t]. Quando essa incontra un'istruzione con codifica long e campo opcode pari al numero decimale '40', si trova di fronte ad un'istruzione di *chiamata a sistema* (vedi 3.2.4) e, in questo caso, mediante un costrutto 'switch - case' [1t], che utilizza come parametro di scelta il primo argomento dell'istruzione in questione (SysCallID), estrapola il codice della syscall richiesta dall'istruzione stessa, e la invoca. Si può quindi poi passare all'interpretazione dell'istruzione successiva.

Per poter simulare il display del robot, si è dovuta arricchire Execution con le seguenti sette nuove syscall del firmware:

- **NXTDrawText**: stampa una stringa testuale sul display. Nel cluster di parametri in ingresso necessita nell'ordine delle coordinate *x* ed *y* (relative all'angolo in basso a sinistra del display) in cui piazzare la base della prima lettera della stringa da visualizzare, del testo della medesima, e di un campo opzioni in cui specificare se “pulire” o meno il display prima di stamparvi il testo desiderato;
- **NXTDrawPoint**: disegna un punto nel display. Nel cluster di parametri in ingresso necessita delle coordinate *x* ed *y* in cui piazzare il punto, e di un campo opzioni con ruolo analogo a quello descritto per la syscall precedente;
- **NXTDrawLine**: traccia una linea retta nel display. Nel cluster di parametri in ingresso necessita nell'ordine delle coordinate *x* ed *y* in cui far iniziare la linea, di quelle in cui farla terminare, e dell'usuale campo opzioni;
- **NXTDrawCircle**: disegna un cerchio nel display. Nel cluster di parametri in ingresso necessita delle coordinate cartesiane *x* ed *y* del centro del cerchio, della lunghezza in pixel del raggio, e del campo opzioni;
- **NXTDrawRect**: traccia un rettangolo nel display. Nel cluster di parametri in ingresso necessita nell'ordine delle coordinate *x* ed *y* in cui posizionare l'angolo inferiore sinistro del rettangolo, della larghezza e dell'altezza del medesimo in pixel, e del campo opzioni;
- **NXTDrawPicture**: visualizza nel display un'immagine in formato proprietario LEGO® *.RIC*. Nel cluster di parametri in ingresso necessita nell'ordine delle coordinate *x* ed *y* in cui piazzare il margine più in basso a sinistra dell'immagine, del nome dell'immagine medesima, di un campo di

vari parametri opzionali della stessa, e del campo opzioni;

- **NXTSetScreenMode**: imposta una nuova modalità di esecuzione per il display. Il cluster di parametri in ingresso necessita di un solo campo, il quale specifica la nuova modalità da applicare al display; il firmware 1.28 prevede la sola modalità di "pulizia", ossia quella che permette, se invocata, di eliminare tutti gli oggetti eventualmente visualizzati sul display.

La scelta che si è fatta è stata quella di non sovraccaricare la parte di Execution relativa alle syscall appena presentate di tutte le istruzioni che esse effettivamente richiedevano per essere simulate, ma di "spalmare" le stesse su più classi, seguendo quindi un approccio *modulare*. Per quanto riguarda, ad esempio, la chiamata *NXTDrawText*, essa è stata ivi realizzata mediante l'estratto di codice seguente.

```
case 40: //OP_SYSCALL
   coppia[0] = NXTSView.sim.lett[pc + 2];
   coppia[1] = NXTSView.sim.lett[pc + 3];
   source1 = conv.getPosInt(coppia); //SysCallID
   coppia[0] = NXTSView.sim.lett[pc + 4];
   coppia[1] = NXTSView.sim.lett[pc + 5];
   source2 = conv.getPosInt(coppia); //Cluster dei Parametri di ingresso
   switch((int)source1){
            . . .
      case 13:
         int priority_13 = NXTSView.incrementDisplayThreadPriority();
         NXTSView.dCond.modifyDisplay(priority_13,13,NXTSView.sim.table,
         (int)source2,NXTSView.sim.array);
         NXTSView.dCond.updateDisplayPanel(priority_13);
         break;
            . . .
   }
            . . .
   break;
```

A partire dal codice sopra riportato, si può descrivere com'è stata implementata nella sua totalità la syscall in esame. Tutte le altre sono state implementate in maniera assolutamente analoga e la loro descrizione risulta quindi pleonastica.

L'istruzione '*case 13*' fa riferimento al SysCallID relativo alla chiamata *NXTDrawText*.

La variabile intera *priority_13* contiene la priorità che viene assegnata alla syscall in questione. Tale valore viene fornito dal metodo *incrementDisplayThreadPriority()*, implementato all'interno della classe NXTSView. All'interno di quest'ultima è definita una variabile, chiamata *displayThreadPriority*, la quale assume inizialmente valore '1' (ed il cui valore torna tale ad ogni avvio di una simulazione), ed è aumentata ogni volta di una unità ad ogni invocazione del metodo *incrementDisplayThreadPriority()*. Così facendo, ad ogni syscall del display che va in esecuzione viene assegnata una distinta priorità; in seguito si scoprirà l'utilità di tale scelta (vedi paragrafo *La Priorità delle Syscall del Display*).

All'interno della classe NXTSView è definito un oggetto denominato *dCond* (reinizializzato all'inizio di ogni simulazione), istanza di classe DisplayCondition. Tale classe è stata creata con lo scopo di salvare durante una simulazione lo "stato" del display del simulatore e di lanciare le funzioni che vanno a modificarlo. Ogni volta che viene lanciata una syscall del display, è, infatti, invocato il metodo *modifyDisplay()* della classe DisplayCondition, al quale vengono forniti in ingresso i parametri necessari per invocare successivamente la funzione appropriata di modifica vera e propria del display. Una volta lanciato, tale metodo costruisce semplicemente un'istanza di classe DisplayDrawings, la quale si occupa di attuare le modifiche da applicare al display del simulatore, nel modo che vedremo tra poco. Una volta eseguito il metodo *modifyDisplay()*, viene invocato *updateDisplayPanel()*, il quale si occupa di rendere effettivamente visibili i cambiamenti effettuati sul display nell'interfaccia grafica del simulatore. Il display 200x128 del simulatore viene, infatti, gestito come immagine (nella fattispecie come istanza di *java.awt.image.BufferedImage* [16s]), istanziata all'interno della classe DisplayCondition, e visualizzata dall'oggetto *monitor*. Per ogni syscall che si invoca, *dCond* istanzia, come detto, un oggetto di classe DisplayDrawings, fornendogli in ingresso i parametri richiesti, sulla base del tipo di syscall. Tale oggetto si occupa poi di modificare l'immagine, di cui sopra, da utilizzare per il display. Una volta terminata la modifica, il metodo *updateDisplayPanel()* si incarica quindi di aggiornare *monitor* con la nuova immagine modificata.

La classe DisplayDrawings estende la classe Java *javax.swing.JPanel*, sovrascrivendone il metodo *paintComponent()*. Così facendo, ogni volta che si crea un oggetto di classe DisplayDrawings, viene subito invocato di default il metodo appena citato [16s]. Nel nostro caso si è scelto di dotare tale classe di un costruttore distinto per ciascuna syscall del display, così da inizializzare, di volta in volta,

soltanto le variabili necessarie per invocare quella particolare syscall. Come detto, all'interno del metodo *paintComponent()* avviene poi la modifica vera e propria dell'immagine relativa al display del simulatore. Tale metodo contiene un costrutto 'switch' mediante il quale sceglie, sulla base della syscall da simulare, quale funzione di modifica del display applicare. Di seguito viene proposto un estratto di codice sorgente del metodo in questione, relativo alla syscall per la stampa di una stringa di testo sul display.

```
public void paintComponent(Graphics g){
  synchronized(NXTSView.dCond){
    super.paintComponent(g);
                . . .
    Graphics2D gr = (NXTSView.dCond.getImmagineDisplay()).createGraphics();
                . . .
    switch(operationCode){ //SysCallID
      case 13: //TESTO
        if(options == 1) //'Pulisce' il display prima di ritoccarlo
        { gr.clearRect(0, 0, 2 * DISPLAY_WIDTH, 2 * DISPLAY_HEIGHT);  }
        float xString = 2 * xIn + 1; //coordinata 'x'
        float yString = (DISPLAY_HEIGHT - yIn)* 2 - 3; //coordinata 'y'
        Font f = new Font("Arial",Font.PLAIN,15);
        gr.setFont(f); //Imposta Font corretto del display del robot
        String text = textIn; //Testo da stampare
        gr.drawString(text, xString, yString);
        break;
                . . .
    }
                . . .
    NXTSView.dCond.setActualPriority(); //aggiorna valore 'actualPriority'
    NXTSView.dCond.notifyAll(); //notifica dell'avvenuto aggiornamento
  }
}
```

Analizzando il codice sorgente riportato qui sopra, si nota che il metodo *paintComponent()* inizialmente acquisisce il *lock* [3t] per agire in mutua esclusione sull'oggetto *dCond* che andrà a modificare, e poi istanzia un oggetto di classe Graphics2D [17s], necessario come "involucro" per agire sull'immagine visualizzata nel display. Successivamente, infatti, mediante le funzioni di disegno messe a disposizione da tale oggetto, esso apporta le modifiche desiderate all'immagine che verrà in seguito messa a disposizione di DisplayCondition, per essere visualizzata su

*monitor*, mediante l'invocazione al metodo *updateDisplayPanel()*.

Come detto in precedenza, tutte le syscall che agiscono sul display del simulatore sono state implementate in maniera del tutto analoga a quella di visualizzazione di una stringa appena descritta; tuttavia un'altra di esse merita una precisazione riguardante la propria implementazione: NXTDrawPicture. Quest'ultima si occupa di visualizzare sul display un'immagine in formato proprietario LEGO® (ossia con estensione *.RIC*). Immagini di questo tipo sono visualizzabili soltanto mediante il robot Lego® Mindstorms® NXT od il software di programmazione grafica NXT-G (descritto nella sottosezione 2.2.1 di questo elaborato). Per poter visualizzare questo tipo di immagini anche mediante il display di *NXTSimulator*, si è quindi dovuto effettuare un processo di conversione delle medesime dal formato nativo ad un altro più facilmente accessibile, nella fattispecie il *JPEG*. Al momento attuale non esiste un software che permetta di operare in modo automatico la suddetta conversione. Per metterla in pratica si è quindi dovuto utilizzare un metodo piuttosto "spartano": si sono visualizzate ad una ad una, mediante NXT-G, tutte le immagini da esso messe a disposizione per essere visualizzate sul display, e, per ognuna di esse, si è utilizzata la funzione di Windows (*fn* + *stamp*) che permette di scattare un'*istantanea* dello schermo, andando poi ad estrapolare da essa la parte relativa all'immagine a cui si era interessati, per salvarla in formato *.JPEG*. Una volta ottenuto in questo modo tutto il *pool* di immagini offerto da NXT-G, lo si è salvato nel percorso seguente del file di distribuzione di *NXTSimulator*: *..\src\nxtsimulator\resources\ric*. Quando un programma eseguito dal simulatore richiede di visualizzare un'immagine, esso va quindi a reperirla nel suddetto *pool*. E' importante far notare che, qualora in futuro NXT-G arricchisse il proprio repertorio con nuove immagini, queste non sarebbero ovviamente visualizzabili mediante il display del simulatore. Per rendere ciò possibile, bisognerebbe quindi attuare la procedura di conversione descritta sopra anche per le nuove immagini.

**La Priorità delle Syscall del Display**

Ora che si è compreso il modo in cui è stato implementato il display di *NXTSimulator*, resta soltanto una questione ancora da chiarire: quella relativa alla priorità delle syscall di modifica del display medesimo.

Come già anticipato in precedenza, appena viene invocata, ogni istruzione di questa famiglia riceve un determinato e distinto valore di priorità. La classe

DisplayCondition gestisce, a sua volta, al suo interno un altro valore di priorità connesso a quello appena citato: tale valore numerico si chiama *actualPriority* (il cui valore effettivo è resettato ad '1' all'inizio di ogni simulazione). Quest'ultimo specifica la priorità della successiva syscall che ha diritto di agire sul display. Quando una syscall con una data priorità, dall'interno della classe Execution, invoca il metodo *modifyDisplay()* di DisplayCondition, quest'ultimo (e con esso anche le funzioni da esso successivamente invocate ovviamente) viene eseguito soltanto se la priorità della syscall invocante combacia col valore di *actualPriority*; qualora ciò non accada, la syscall si mette in attesa del proprio "turno". Solo quando esso arriverà, la syscall potrà effettivamente operare le proprie azioni sul display. Ogni volta che una syscall conclude il proprio operato, infatti, essa incrementa il valore di *actualPriority*, aggiornandolo alla priorità dell'eventuale successiva syscall da mandare in esecuzione, e notifica le eventuali syscall in attesa dell'evento appena verificatosi. Le ultime due istruzioni dell'estratto di codice riportato a pagina 59 sono eseguite al termine del metodo *paintComponent()* di un'istanza di DisplayDrawings, per effettuare le due azioni descritte sopra.

La scelta implementativa appena illustrata si è resa necessaria per evitare che, ad esempio, due syscall di modifica del display, presenti l'una di seguito all'altra in un programma da simulare, venissero eseguite in ordine inverso, qualora la seconda di esse acquisisse il *lock* sull'oggetto *dCond* prima dell'altra. Senza le due righe di codice di controllo posizionate al principio sia del metodo *modifyDisplay()* che di *updateDisplayPanel()* e riportate di seguito, a volte si verificava, infatti, il malfunzionamento appena citato. Con il *modus operandi* adottato, invece, le syscall vanno in esecuzione nell'ordine esatto in cui esse sono invocate all'interno della classe Execution.

```
while(threadPriority != actualPriority)
    try{ wait(); } catch(InterruptedException e){}
```

## 4.6 Implementazione Riproduttore di Suoni

Il *riproduttore di suoni* è il dispositivo, integrato nel brick del robot LEGO®, che permette a quest'ultimo di riprodurre contenuti audio essenzialmente di due tipologie. La prima è quella comprendente file audio veri e propri di una certa durata ed in formato proprietario LEGO® (con estensione *.RSO* e di cui si discuterà in seguito). La seconda, invece, è quella relativa a semplici note musicali, di diversa frequenza e durata.

Come descritto nella sottosezione 2.1.1 di questo elaborato, il riproduttore di suoni del robot è un semplice altoparlante mono a 8 bit, capace di riprodurre suoni fino a 16 [KHz]. Per dotare *NXTSimulator* di tale dispositivo, come già accennato in precedenza, si è deciso di aggiungere alla classe *NXTSView.java* del progetto un pannello di tipo JPanel denominato *port9*. Quest'ultimo contiene al proprio interno due oggetti JLabel, uno contenente il solo nome 'Suono', e l'altro riportante l'immagine di un altoparlante e denominato *imgSound*. Il riproduttore così ottenuto nel simulatore è rappresentato nel riquadro azzurro della figura 4.2 di pagina 51. Tra i due di cui è dotato, il componente più importante di *port9* è sicuramente *imgSound*: quando, infatti, all'interno di un programma fatto eseguire al simulatore, è presente un'istruzione che richiede la riproduzione di un suono, l'aspetto di *imgSound* può assumere due sembianze distinte. Qualora si richieda la riproduzione di un file audio, durante l'emissione del suono da parte del calcolatore, l'altoparlante visualizzato da *imgSound* viene temporaneamente arricchito con delle onde sonore in uscita da esso; qualora, invece, sia richiesta la riproduzione di una nota musicale, durante l'emissione della medesima, l'altoparlante di *imgSound* viene temporaneamente rimpiazzato dall'immagine di una nota musicale. Grazie alla scelta implementativa appena descritta, l'utente che utilizza il simulatore può così apprendere sia con l'udito che con la vista della riproduzione di suoni in corso. Di seguito vengono proposte tre immagini relative all'aspetto di *imgSound*, quando si trova rispettivamente a non riprodurre nulla ('idle'), a riprodurre un file audio e a riprodurre una nota musicale.

**Figura 4.5:** *Riproduttore di Suoni - 'Idle', File Audio, Nota Musicale*

Ora che si è descritto l'aspetto strettamente grafico della realizzazione del riproduttore di suoni, si può passare a trattare lo sviluppo delle funzioni che hanno

reso possibile l'implementazione vera e propria delle funzionalità dello stesso su *NXTSimulator.*

Il processo che ha permesso di dotare il simulatore delle funzionalità appena citate ha richiesto modifiche alla classe del progetto *Execution.java* e alla già citata *NXTSView.java*; si sono dovute, inoltre, aggiungere due nuove classi: *SoundCondition.java* e *SoundReproducer.java*.

Per poter simulare il riproduttore di suoni del robot, si è dovuta arricchire la parte di Execution (classe che, come appreso nella sezione 4.5, interpreta ed esegue le istruzioni di un programma dato in ingresso al simulatore) incaricata di eseguire le syscall del firmware con le seguenti quattro nuove chiamate a sistema:

- **NXTSoundPlayFile**: riproduce un file audio in formato proprietario LEGO® *.RSO*. Nel cluster di parametri in ingresso necessita nell'ordine del nome del file da riprodurre, di un valore booleano che specifichi se riprodurre o meno il file in un *loop* continuo, e del volume al quale riprodurlo;
- **NXTSoundPlayTone**: riproduce una specifica nota musicale. Nel cluster di parametri in ingresso necessita della frequenza della nota da riprodurre, della durata della stessa, di un valore booleano che specifichi se riprodurre o meno la nota in un *loop* continuo, e del volume al quale riprodurla;
- **NXTSoundGetState**: legge lo stato ed i flag interni (della cui utilità si discuterà in seguito) del riproduttore di suoni. Nel cluster di parametri vanno salvati i valori di stato e flag che si sono letti;
- **NXTSoundSetState**: imposta lo stato ed i flag del riproduttore di suoni. E' utilizzato soltanto per interrompere la riproduzione audio corrente. Nel cluster di parametri in ingresso necessita dei valori di stato e di flag da assegnare al riproduttore di suoni stesso.

Come accaduto per le syscall relative al display, anche per quelle sopra illustrate la scelta per la quale si è propenso è stata quella di non sovraccaricare la parte di Execution ad esse relativa di tutte le istruzioni effettivamente necessarie per simularle, ma di “spalmare” le stesse su più classi, adottando quindi un approccio più *modulare*. Esse sono state, infatti, ivi realizzate col solo codice sorgente riportato a pagina seguente.

```
case 40: //OP_SYSCALL
   coppia[0] = NXTSView.sim.lett[pc + 2];
   coppia[1] = NXTSView.sim.lett[pc + 3];
   source1 = conv.getPosInt(coppia); //SysCallID
   coppia[0] = NXTSView.sim.lett[pc + 4];
   coppia[1] = NXTSView.sim.lett[pc + 5];
   source2 = conv.getPosInt(coppia); //Cluster dei Parametri di ingresso e uscita
   switch((int)source1){
            . . .
      case 9: //NXTSoundPlayFile
         if(NXTSView.sCond.getLoopSet() == false){
                        . . .
         int priority_9 = NXTSView.incrementSoundThreadPriority();
         NXTSView.sCond.soundPlayFile(priority_9,NXTSView.sim.table,
         (int)source2,NXTSView.sim.array);
         }
         break;
      case 10: //NXTSoundPlayTone
         if(NXTSView.sCond.getLoopSet() == false){
                        . . .
         int priority_10 = NXTSView.incrementSoundThreadPriority();
         NXTSView.sCond.soundPlayTone(priority_10,NXTSView.sim.table,(int)source2);
         }
         break;
      case 11: //NXTSoundGetState
         NXTSView.sCond.soundGetState(NXTSView.sim.table,(int)source2);
         break;
      case 12: //NXTSoundSetState
         NXTSView.sCond.soundSetState(NXTSView.sim.table,(int)source2);
         break;
            . . .
   }
            . . .
   break;
```

Partendo dall'analisi del codice sorgente qui sopra proposto, si può descrivere nel dettaglio come sono state implementate le syscall in questione. Come si può notare a vista d'occhio è presente una certa similitudine tra le istruzioni rispettivamente di NXTSoundPlayFile e NXTSoundPlayTone, e di NXTSoundGetState e NXTSoundSetState. Tale somiglianza “visiva” ha anche un effettivo riscontro a livello implementativo. La descrizione delle chiamate a sistema in analisi verrà perciò effettuata tenendo conto di questo fatto.

All'interno della classe NXTSView è istanziato un oggetto di nome *sCond* (reinizializzato all'inizio di ogni simulazione), istanza di classe SoundCondition. Tale classe è stata creata con lo scopo di salvare durante una simulazione lo "stato" del riproduttore di suoni del simulatore e di lanciare le funzioni che vanno a modificarlo. Ogni volta che viene eseguita una syscall del riproduttore audio, è, infatti, invocato un metodo della classe SoundCondition. Tale metodo varia in base alla chiamata effettuata, e gli vengono forniti in ingresso i parametri necessari per eseguirla.

### 4.6.1 NXTSoundPlayFile - NXTSoundPlayTone

Appena si invoca una syscall di riproduzione audio (NXTSoundPlayFile o NXTSoundPlayTone), il primo evento che avviene è la verifica, mediante il metodo *getLoopSet()* di SoundCondition, che il riproduttore di suoni non stia eseguendo una riproduzione audio ripetuta continuamente (ossia in *loop*). Qualora la verifica dia esito negativo, la syscall invocata non viene eseguita, come vogliono le specifiche del firmware del robot LEGO® [1s], ed *NXTSimulator* procede con l'interpretazione dell'istruzione successiva del programma *.RXE* in esecuzione. Qualora, invece, la verifica dia esito positivo, avviene quanto segue. All'interno della variabile intera *priority_9/priority_10* viene salvata la priorità assegnata alla syscall in questione, mediante un meccanismo assolutamente analogo a quello descritto nella sezione precedente di questo elaborato per le syscall del display. Così facendo, ad ogni chiamata a sistema di tipo NXTSoundPlayFile o NXTSoundPlayTone viene assegnata una distinta priorità; in seguito si scoprirà l'utilità di tale scelta (vedi paragrafo *La Priorità delle Syscall del Riproduttore di Suoni*). Terminata questa prima fase di controllo ed inizializzazione, viene invocato il metodo di *sCond* che andrà poi effettivamente ad agire sul riproduttore di suoni. Nel caso di NXTSoundPlayFile viene chiamato il metodo *soundPlayFile()*, mentre nel caso di NXTSoundPlayTone è lanciato il metodo *soundPlayTone()*. Entrambi i metodi svolgono delle operazioni assolutamente analoghe: essi istanziano semplicemente un oggetto/thread [3t] di classe SoundReproducer, fornendogli in ingresso i parametri di cui necessita, e lanciano poi questo thread, col tipico comando *start()*. Si è scelto di dotare SoundReproducer di due distinti costruttori: uno per l'invocazione da parte di NXTSoundPlayFile, ed uno per NXTSoundPlayTone. Così facendo si inizializzano, di volta in volta, soltanto le variabili necessarie per invocare una delle due syscall. Una volta lanciato, un thread di classe SoundReproducer esegue, come da specifiche

Java [16s], il proprio metodo *run()*, il cui codice sorgente viene riportato di seguito.

```
public void run(){
  if(NXTSView.sCond.getSoundStopped() == false){
     if(loop == 1){ //Loop
        NXTSView.sCond.loopSet(); //Imposta condizione di riproduzione in 'loop'
                  . . .
        while((NXTSView.playIconOn == false) && (NXTSView.sCond.getLoopSet() == true)){
           if(fileName.equals("")) //Riproduce nota musicale
           {
              if(firstTime || !player.isPlaying()) //Verifica se è terminata la riproduzione della nota
              {
                 . . .
                 playNote();
               } else { continue; }
            } else { playSound(); } //Riproduce file musicale
            ImageIcon imgLoop = new ImageIcon("src/nxtsimulator/resources/NoSound.png");
            NXTSView.imgSound.setIcon(imgLoop);
        }
     } else { //No loop
        if(fileName.equals("")) //Riproduce nota musicale
        { playNote(); }
        else //Riproduce file musicale
        { playSound(); }
                  . . .
        ImageIcon img = new ImageIcon("src/nxtsimulator/resources/NoSound.png");
        NXTSView.imgSound.setIcon(img);
     }
  }
}
```

La prima istruzione del metodo è di fatto una “guardia”, antistante a tutto il resto delle istruzioni, le quali vengono eseguite soltanto se la condizione imposta dalla guardia stessa è verificata. Il thread, infatti, riproduce l'audio richiesto soltanto se non è stato dato nel frattempo dall'utente il comando di interrompere la simulazione in corso. Qualora ciò non sia avvenuto, il metodo prosegue facendo un'ulteriore verifica, ossia se si debba riprodurre l'audio in questione in un *loop* continuo, od una volta soltanto. Nel primo caso, il metodo ripete continuamente la riproduzione del file o della nota musicale, fino a che l'utente non interrompe la simulazione, o nel programma che si sta simulando non si incontra un'istruzione che impone

l'interruzione di tale *loop* (mediante la syscall NXTSoundSetState, trattata nella sottosezione successiva); nel secondo caso la riproduzione avviene, invece, una volta soltanto.

Sia in situazione di *loop* che non, la riproduzione effettiva di una nota musicale avviene mediante l'invocazione del metodo *playNote()*, mentre quella di un file audio mediante il metodo *playSound()*. Entrambi i metodi sono implementati all'interno della medesima classe SoundReproducer.

```
public void playNote(){
   synchronized(NXTSView.sCond){
      NXTSView.sCond.flags[1] = 0;     NXTSView.sCond.flags[0] = 1;
      NXTSView.sCond.state = 3;
      NXTSView.sCond.notifyAll();
   }
   ImageIcon img = new ImageIcon("src/nxtsimulator/resources/Tone.png");
   NXTSView.imgSound.setIcon(img);
   int vol = 0;
   switch(volume){ //Imposta volume nota da riprodurre al valore richiesto
      case 0: //disabilitato
         vol = (MAX_TONE_VOL*0)/100;     break;
      case 1: //25%
         vol = (MAX_TONE_VOL*25)/100;     break;
                  . . .
   }
   switch(frequency){ //Riproduce la frequenza corrispondente alla nota richiesta
      case 262:     player.play("X[Volume]="+vol+" I4 C4/"+duration);     break;
      case 277:     player.play("X[Volume]="+vol+" I4 C#4/"+duration);     break;
                  . . .
   }
   synchronized(NXTSView.sCond){
      if(loop == 0){
         NXTSView.sCond.flags[0] = 0;
         NXTSView.sCond.state = 0;
         NXTSView.sCond.setActualPriority();
         NXTSView.sCond.notifyAll();
      }
   }
}
```

L'estratto di codice sorgente riportato a pagina precedente è quello che implementa il metodo *playNote()*.

La riproduzione di note musicali nell'ambiente Java del simulatore è stata resa possibile grazie all'integrazione nel progetto NetBeans di *NXTSimulator* di una libreria Java aggiuntiva, denominata *jfugue* e scaricabile gratuitamente all'indirizzo internet specificato alla voce [18s] della bibliografia. Tale libreria è concepita per la composizione di musica al calcolatore e presenta un gran numero di classi dalle più svariate caratteristiche. Nel nostro caso è stato necessario utilizzare soltanto una delle suddette classi: *Player.java*. Essa permette, mediante un'unica ed intuitiva istruzione, di riprodurre una determinata nota musicale, dovendo specificare solamente il volume al quale eseguirla, la nota stessa (con la notazione inglese - e.g: C sta per Do), e la sua durata in secondi.

Verrà ora delineato il contesto di utilizzo nel quale è stata utilizzata la libreria *jfugue*, ossia quello del metodo *playNote()*.

Il metodo *playNote()*, appena viene invocato, acquisisce il *lock* [3t] sull'oggetto *sCond*, imposta i corretti valori di stato e flag (il cui significato lo si vedrà in seguito), e successivamente lo rilascia, dopo aver notificato eventuali “ascoltatori” dell'avvenuto aggiornamento dell'oggetto *sCond* medesimo. Successivamente il metodo cambia l'immagine di *imgSound*, sostituendo temporaneamente l'altoparlante 'idle' con la nota musicale. Sarà poi il metodo *run()* ad occuparsi di ripristinare l'immagine dell'altoparlante 'idle', una volta terminata l'esecuzione di *playNote()*. Una volta visualizzata la nota musicale, *playNote()* imposta, mediante un costrutto 'switch', il volume (il cui valore può essere una percentuale pari a 0, 25, 50, 75, o 100% del volume attuale del calcolatore su cui è eseguito il simulatore) al quale riprodurre la nota musicale, sulla base dell'apposito parametro fornito in ingresso nel cluster della syscall NXTSoundPlayTone. Tale cluster contiene, inoltre, la frequenza della nota musicale da riprodurre (in Hertz): grazie ad un altro costrutto 'switch', che utilizza come parametro di scelta il valore di frequenza appena citato, si avvia la riproduzione della nota desiderata, mediante l'invocazione del metodo *play()* (con specificati i parametri discussi in precedenza) sull'oggetto *player*, istanza della classe Player, inclusa, come detto, nella libreria *jfugue*.

Una volta terminata la riproduzione della nota musicale desiderata, il metodo *playNote()* acquisisce nuovamente il *lock* sull'oggetto *sCond*, per impostare i nuovi valori di stato e flag, rilasciandolo successivamente, dopo aver notificato eventuali

“ascoltatori” dell'avvenuto aggiornamento dell'oggetto *sCond* medesimo.

```
public void playSound(){
   synchronized(NXTSView.sCond){
      NXTSView.sCond.flags[1] = 0;
      NXTSView.sCond.flags[0] = 1;
      NXTSView.sCond.state = 2;
      NXTSView.sCond.notifyAll();
   }
   ImageIcon img = new ImageIcon("src/nxtsimulator/resources/Sound.png");
   NXTSView.imgSound.setIcon(img);
   File soundFile = new File(fileName); //File da riprodurre
                  . . .
   FloatControl gainControl = auline.getControl(FloatControl.Type.MASTER_GAIN);
   float maxVol = gainControl.getMaximum() + 80.0f;
   float minVol = gainControl.getMinimum() + 80.0f;
   float vol = maxVol - minVol;
   switch(volume){ //Imposta volume file da riprodurre al valore richiesto
      case 0: //disabilitato
         BooleanControl muteControl = auline.getControl(BooleanControl.Type.MUTE);
         muteControl.setValue(true);
         break;
      case 1: //25%
         vol = ((vol * 25)/100) - 80.0f;
         gainControl.setValue(vol);
         break;
                  . . .
   }
   auline.start();
                  . . .
   synchronized(NXTSView.sCond){
      if(loop == 0){
         NXTSView.sCond.flags[0] = 0;
         NXTSView.sCond.state = 0;
         NXTSView.sCond.setActualPriority();
         NXTSView.sCond.notifyAll();
      }
   }
}
```

Il listato di codice Java riportato a pagina precedente contiene le istruzioni più significative del metodo *playSound()*, ossia quelle la cui analisi risulta fondamentale per la comprensione del funzionamento di tale metodo. Molte istruzioni di *playSound()* sono state ivi omesse, in quanto pleonastiche in questo contesto esplicativo. Per una descrizione dettagliata delle stesse si rimanda al sito specificato al punto [19s] della bibliografia, in quanto tali istruzioni sono state tratte da tale fonte.

Come il metodo *playNote()* visto in precedenza, anche il metodo *playSound()*, appena viene lanciato, imposta con procedimento analogo i corretti valori di stato e flag dell'oggetto *sCond*, e cambia l'immagine di *imgSound*, sostituendo temporaneamente l'altoparlante 'idle' con quello che emette onde sonore. Anche in questo caso sarà poi il metodo *run()* ad occuparsi di ripristinare l'immagine dell'altoparlante 'idle', una volta terminata l'esecuzione di *playSound()*.

Successivamente il metodo istanzia un oggetto di classe *java.io.File* [16s], così da avere un riferimento al file audio che si desidera riprodurre. Viene quindi impostato, mediante un costrutto 'switch', il volume al quale riprodurre (il cui valore, come nel caso di playNote, può essere una percentuale pari a 0, 25, 50, 75, o 100% del volume attuale del calcolatore su cui è eseguito il simulatore) il file.

L'istanza di classe File, contenente il file audio da riprodurre, viene incapsulata [1t] in diversi oggetti, mediante vari passaggi successivi, che non sono riportati nel codice a pagina precedente. L'unico passaggio interessante per la nostra discussione è quello che riguarda l'incapsulamento del file all'interno dell'ultimo di questi oggetti, ossia quello chiamato *auline*, istanza di classe *javax.sound.sampled.SourceDataLine* [16s]. Mediante un comando *start()* su tale oggetto, viene, infatti, avviata la riproduzione del file audio desiderata. Una volta terminata quest'ultima, in modo analogo a quanto fatto dal metodo *playNote()*, *playSound()* imposta i nuovi valori di stato e flag dell'oggetto *sCond*.

Resta ancora da fare una precisazione riguardante il formato dei file audio riprodotti dal simulatore. Il riproduttore di suoni del robot Lego® Mindstorms® NXT è in grado di riprodurre, come già asserito in precedenza, file audio in formato proprietario con estensione *.RSO*. File in questo formato sono, però, riproducibili soltanto mediante il robot LEGO®, o mediante il software di programmazione per lo stesso: NXT-G. Per permettere anche al simulatore di riprodurre tale tipologia di file, si è dovuto ricorrere ad uno strumento di conversione degli stessi ad un formato

adatto per la riproduzione in ambiente Java (e.g: file *.WAV*). Per fare ciò si è utilizzato un programma eseguibile (che non necessita di alcuna installazione), scaricato dal sito riportato alla voce [20s] della bibliografia, chiamato *wav2rso*. Come suggerisce il nome, questo software permette di convertire file audio con estensione *.WAV* in file con estensione *.RSO* e viceversa. Appena lo si lancia, il software fornisce all'utente l'interfaccia rappresentata nella figura che segue.

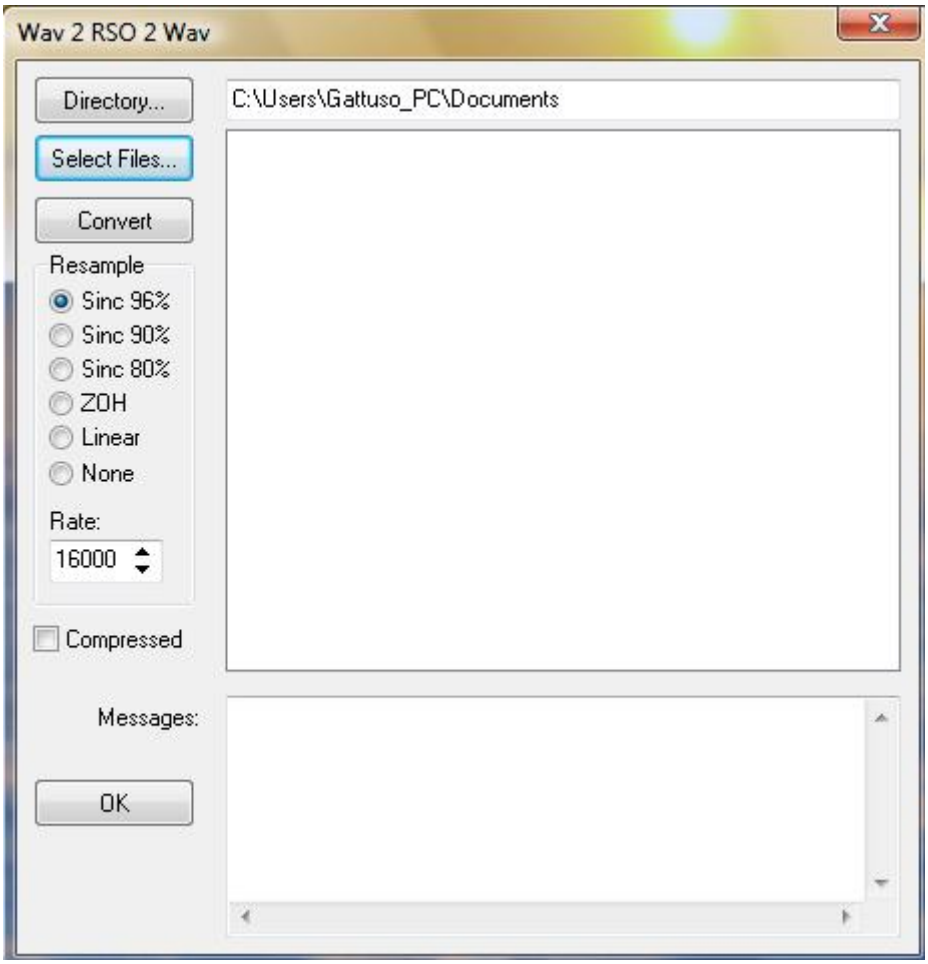


**Figura 4.6:** *wav2rso*

Come si può facilmente intuire dall'immagine sopra riportata, per ottenere la conversione di un file audio nel formato destinazione desiderato, è sufficiente specificare la directory di uscita in cui salvare il file convertito, il file da convertire (il quale deve essere in uno dei due formati di cui sopra), e la frequenza alla quale campionare quest'ultimo.

Nel nostro contesto operativo, si è utilizzato *wav2rso* per convertire tutti i file audio messi a disposizione da NXT-G dal formato proprietario LEGO® *.RSO* al formato *.WAV*. Per fare ciò, si sono presi, ad uno ad uno, tutti i file *.RSO* presenti nella cartella di installazione di NXT-G del calcolatore utilizzato (*..\LEGO MINDSTORMS Edu NXT\engine\Sounds*), e si sono convertiti nel formato *.WAV*,

salvandoli nel percorso seguente del file di distribuzione di *NXTSimulator*: *..\src\nxtsimulator\resources\rso*. Quando un programma eseguito dal simulatore richiede di riprodurre un file audio in formato *.RSO*, esso va quindi a reperirlo nel *pool* creato col metodo appena discusso. E' importante far notare che, qualora in futuro NXT-G arricchisse il proprio repertorio con nuovi file audio, questi non sarebbero ovviamente riproducibili mediante il riproduttore di suoni del simulatore. Per rendere ciò possibile, bisognerebbe quindi attuare la procedura di conversione descritta sopra anche per i nuovi file audio.

**La Priorità delle Syscall del Riproduttore di Suoni**

Ora che si è delineato il modo in cui sono state implementate le syscall di riproduzione audio su *NXTSimulator*, resta da chiarire solamente la questione relativa alla priorità che viene assegnata a ciascuna di esse appena viene invocata.

Il meccanismo che verrà descritto è del tutto simile a quello adottato nel caso della priorità delle syscall del display (vedi sezione 4.5).

In precedenza si è già detto che, appena viene invocata, ogni chiamata di tipo NXTSoundPlayFile o NXTSoundPlayTone riceve un determinato e distinto valore di priorità. La classe SoundCondition gestisce, a sua volta, un altro valore di priorità connesso a quello appena citato: tale valore numerico si chiama *actualPriority* (il cui valore è impostato ad '1' all'inizio di ogni simulazione). Quest'ultimo specifica la priorità della successiva syscall di riproduzione audio che ha diritto di agire sul riproduttore di suoni, per eseguire un determinato contributo sonoro. Quando una syscall, che sia NXTSoundPlayFile o NXTSoundPlayTone (con una data priorità assegnata), dall'interno della classe Execution invoca rispettivamente il metodo *soundPlayFile()* o *soundPlayTone()* di SoundCondition, quest'ultimo viene eseguito soltanto se la priorità della syscall invocante combacia col valore di *actualPriority*; qualora ciò non accada, la syscall si mette in attesa del proprio “turno”. Solo quando esso giungerà, la syscall potrà effettivamente eseguire le proprie azioni di riproduzione audio. Ogni volta che una syscall del tipo in analisi conclude il proprio operato, infatti, essa incrementa il valore di *actualPriority*, aggiornandolo alla priorità dell'eventuale successiva syscall di quel tipo da mandare in esecuzione, e notifica le eventuali syscall in attesa dell'evento appena verificatosi. Le ultime due istruzioni dell'estratto di codice riportato a pagina 67 e di quello di pagina 69 sono eseguite al termine rispettivamente del metodo *playNote()* e *playSound()* di un'istanza di SoundReproducer, per effettuare le due azioni sopra illustrate.

La scelta implementativa appena esposta si è resa necessaria per evitare che due syscall di riproduzione audio, presenti l'una di seguito all'altra o in due flussi paralleli di un programma dato in pasto al simulatore, venissero eseguite in concomitanza, causando così una sgradevole sovrapposizione sonora. Con la soluzione adottata, invece, le syscall vengono eseguite una alla volta, nella sequenza stilata in base alla priorità assegnata a ciascuna di esse.

Per controllare se una syscall di riproduzione audio ha la priorità adatta per essere mandata in esecuzione, sono state inserite le seguenti due istruzioni di controllo al principio sia del metodo *soundPlayFile()* che di *soundPlayTone()*.

```
while(threadPriority != actualPriority)
    try{ wait(); } catch(InterruptedException e){}
```

Con il *modus operandi* adottato le syscall vanno così eseguite nell'ordine in cui sono invocate all'interno della classe Execution.

### 4.6.2 NXTSoundGetState - NXTSoundSetState

Le due chiamate a sistema di cui si tratterà in questa sottosezione non provocano, come facevano, invece, le due viste in precedenza, la riproduzione di un contributo sonoro, ma agiscono sulle variabili interne dell'oggetto *sCond*, il quale, come detto in precedenza, è un'istanza della classe SoundCondition e contiene informazioni sullo "stato" del riproduttore di suoni durante una simulazione. Tale "stato" viene specificato principalmente mediante due valori: i campi *state* e *flags*.

Il campo *state* (stato) è un valore numerico che specifica, appunto, in che 'stato' si trova il riproduttore di suoni in un dato istante; in totale vi sono quattro distinti stati in cui esso può trovarsi, ossia i seguenti:

- **0**: *SOUND_IDLE* - il riproduttore di suoni è 'idle', cioè non sta riproducendo alcun suono;
- **2**: *SOUND_FILE* - il riproduttore di suoni sta riproducendo un file audio (nel formato *.RSO*);
- **3**: *SOUND_TONE* - il riproduttore di suoni sta riproducendo una nota musicale;
- **4**: *SOUND_STOP* - al riproduttore di suoni è stata appena inoltrata una richiesta di interrompere la riproduzione audio corrente.

Il campo *flags* (flag) è un campo di due bit utile per specificare quali delle seguenti condizioni sussistono per il riproduttore di suoni: è 'idle', ha una richiesta di riproduzione audio pendente, sta riproducendo qualcosa. I valori che i suddetti bit (il cui bit più significativo è quello più a sinistra) possono assumere sono i seguenti:

- **00**: *SOUND_FLAGS_IDLE*: il riproduttore di suoni è 'idle', cioè non sta riproducendo alcun suono e non ha alcuna richiesta di riproduzione pendente;
- **01**: *SOUND_FLAGS_UPDATE*: il riproduttore di suoni ha una richiesta di riproduzione pendente;
- **10**: *SOUND_FLAGS_RUNNING*: il riproduttore di suoni sta riproducendo un file audio od una nota musicale;
- **11**: il riproduttore di suoni sta riproducendo un file audio od una nota musicale, ed ha anche una richiesta di riproduzione pendente.

Vediamo ora l'effettivo utilizzo dei campi appena presentati, nel contesto del riproduttore di suoni.

Quando in un programma per il robot LEGO® si desidera riprodurre un file audio od una nota musicale, vengono invocate, come specificato in precedenza, rispettivamente le syscall NXTSoundPlayFile o NXTSoundPlayTone. L'invocazione di queste ultime, però, come da specifiche firmware, è talvolta seguita da una (o più) chiamata a *NXTSoundGetState* (per chiarimenti a riguardo consultare la sottosezione 4.6.4). L'esecuzione di questa chiamata serve per controllare lo “stato” del riproduttore di suoni.

Appena si invoca all'interno di un programma *.RXE* la syscall NXTSoundGetState, la classe Execution del simulatore, come si evince dal listato di codice riportato a pagina 64, non fa altro che lanciare il metodo *soundGetState()* dell'oggetto *sCond*, fornendogli in ingresso i parametri di cui necessita. Questo metodo implementa poi effettivamente le funzionalità della syscall NXTSoundGetState.

Viene proposto a pagina seguente il codice sorgente di *soundGetState()*.

```
public synchronized int soundGetState(DSTOC [] table, int source2){
                . . .
   int valFlags = 1 * flags[1] + 2 * flags[0]; //Converte campo di bit 'flags' in decimale
   table[source2+2].setValDefault(valFlags); //Ritorna in uscita il valore di 'flags'
                . . .
   table[source2+1].setValDefault(state); //Ritorna in uscita il valore di 'state'
   return state;
}
```

Come si nota dall'analisi del codice sopra riportato, il metodo in questione non fa altro che salvare in uscita, negli appositi spazi riservati nel cluster di parametri relativo alla syscall NXTSoundGetState, i valori di *flags e state* richiesti.

Quando in un programma per il robot LEGO® si desidera modificare lo "stato" del riproduttore di suoni, ossia si vogliono assegnare nuovi valori ai campi *state* e *flags*, viene invocata la syscall *NXTSoundSetState*. Quest'ultima tuttavia, nella versione 1.28 del firmware, è utilizzata soltanto per imporre l'interruzione dell'eventuale riproduzione audio in corso. Da ciò ne deriva il fatto che è possibile assegnare soltanto il valore '4' al campo *state* (*SOUND_STOP*) ed il valore '0' (entrambi i bit a '0') al campo *flags* (*SOUND_FLAGS_IDLE*).

Appena si invoca all'interno di un programma *.RXE* la chiamata a sistema di cui sopra, la classe Execution di *NXTSimulator*, come si nota nell'estratto di codice sorgente riportato a pagina 64, lancia il metodo *soundSetState()* dell'oggetto *sCond*, fornendogli in ingresso i parametri di cui necessita. Tale metodo esegue poi effettivamente le funzionalità della syscall NXTSoundSetState.

Segue il codice sorgente di *soundSetState()*.

```
public synchronized int soundSetState(DSTOC [] table, int source2){
   state = (int)table[source2+2].getValDefault(); //Assegnazione valore al campo 'state'
   switch((int)table[source2+3].getValDefault()){ //Assegnazione valore al campo di bit 'flags'
      case 0:   flags[0] = 0;   flags[1] = 0;   break;
      case 1:   flags[0] = 0;   flags[1] = 1;   break;
      case 2:   flags[0] = 1;   flags[1] = 0;   break;
      case 3:   flags[0] = 1;   flags[1] = 1;   break;
   }
   table[source2+1].setValDefault(state);  //Ritorna in uscita il valore di 'state'
   setStop(); //Invoca metodo per fermare l'eventuale riproduzione audio corrente
   return state;
}
```

Il metodo in questione, come prima cosa, legge i valori da assegnare a *state* e *flags*, i quali sono forniti in ingresso nel cluster di parametri della chiamata a sistema NXTSoundSetState, e li assegna poi ai suddetti campi di *sCond*. Fatto ciò, viene salvato nell'apposito spazio del cluster della syscall il valore attuale del campo *state*, e viene invocato il metodo *setStop()*: quest'ultimo è incaricato di comandare l'interruzione dell'eventuale riproduzione audio in corso da parte del riproduttore di suoni. Tale metodo, infatti, chiama il metodo *stopAll()*, di classe SoundReproducer, sull'oggetto denominato *lastLaunched*, il quale è un riferimento che *sCond* mantiene all'ultimo thread di classe SoundReproducer che è stato lanciato. Di seguito viene proposto il codice sorgente relativo a *stopAll()*.

```
public void stopAll(){
   if(player.isPlaying()){ //Riproduzione nota musicale attualmente in corso
      player.stop();
      synchronized(NXTSView.sCond){     NXTSView.sCond.state = 0;
         NXTSView.sCond.setActualPriority(); //Aggiorna priorità al valore successivo
         NXTSView.sCond.notifyAll();
      }
   }else if((auline != null) && (auline.isActive())){ //Riproduzione file musicale in corso
      auline.stop();
      synchronized(NXTSView.sCond){
         NXTSView.sCond.state = 0;
         NXTSView.sCond.setActualPriority();     NXTSView.sCond.notifyAll();
      }
   }
}
```

Dal listato di codice sopra riportato si nota che viene dapprima verificato se sia in corso o meno la riproduzione di una nota musicale o di un file audio e, in caso affermativo, la si sospende con l'apposito comando di *stop()* dato rispettivamente all'oggetto *player* o ad *auline*. Fatto ciò, viene acquisito il *lock* [3t] sull'oggetto *sCond*, in modo da impostare il nuovo valore del campo *state*, il quale viene aggiornato a '0' (*SOUND_IDLE*), giacché l'eventuale riproduzione audio in corso è stata abortita. Viene, infine, aggiornato il valore di priorità della successiva syscall audio da mandare eventualmente in esecuzione, e sono notificati eventuali "ascoltatori" dell'avvenuto aggiornamento dell'oggetto *sCond* medesimo.

### 4.6.3 Ulteriori aggiornamenti allo “stato” del riproduttore di suoni

Nel corso della sottosezione precedente si è discusso di come vengono letti e scritti i valori dei campi *state* e *flags*, che descrivono lo “stato” del riproduttore di suoni, mediante rispettivamente le syscall NXTSoundGetState e NXTSoundSetState. Tuttavia, come un lettore particolarmente arguto avrà intuito, anche le chiamate descritte nella sottosezione 4.6.1 devono agire sui suddetti campi.

NXTSoundPlayFile, appena viene invocata dalla classe Execution, come già visto, chiama il metodo *soundPlayFile()* di classe SoundCondition e, per prima cosa, imposta ad '1' il bit meno significativo del campo *flags*, ossia segnala la presenza di una richiesta di riproduzione audio pendente. Appena viene invocato poi il metodo *playSound()* di classe SoundReproducer, il quale effettivamente riproduce il file audio desiderato, quest'ultimo imposta ad '1' il bit più significativo del campo *flags*, ossia segnala una riproduzione in corso, e ovviamente riporta a '0' quello meno significativo, giacché non vi è più la richiesta pendente. Sempre in questo contesto, il campo *state* viene, invece, posto uguale a '2', ossia è segnalata la presenza di una riproduzione di un file in corso. Una volta terminata la riproduzione del file audio, il metodo *playSound()* riporta a '0' il bit più significativo del campo *flags* ed anche il campo *state*, ossia segnala lo stato nuovamente 'idle' del riproduttore di suoni.

NXTSoundPlayTone procede in modo assolutamente analogo. Essa, infatti, effettua la stessa sequenza di aggiornamenti operata da NXTSoundPlayFile, prevedendo ovviamente l'invocazione del metodo *soundPlayTone()* al posto di *soundPlayFile()*, e poi di *playNote()* al posto di *playSound()*. L'unica differenza sta nel valore assegnato a *state* appena viene invocato *playNote()*: in questo caso, infatti, esso viene impostato al valore '3' e non '2', ossia segnala la riproduzione in corso di una nota musicale e non di un file audio.

### 4.6.4 Puntualizzazioni sull'implementazione

Il modo in cui si è simulato il riproduttore di suoni su *NXTSimulator* non rispecchia fedelmente il comportamento che tale dispositivo ha nel robot LEGO®. L'inserimento della priorità, che si è deciso di assegnare alle syscall di riproduzione sonora, ha, infatti, comportato l'omissione di un particolare nella gestione delle stesse. Programmando mediante NXT-G, ed inserendo un blocco di riproduzione di un file audio o di una nota musicale, tra le opzioni che è possibile specificare ve ne è una

denominata 'Wait for Completion', inclusa nell'ellissi rossa della figura seguente.


**Figura 4.7:** *'Wait for Completion'*

Mettendo la spunta su tale opzione, il firmware non passa ad analizzare l'eventuale blocco successivo del programma NXT-G, sino a quando la riproduzione in corso non è terminata. A livello di file compilato (in formato *.RXE*) questa prassi fa sì che, una volta invocata la syscall NXTSoundPlayFile o NXTSoundPlayTone, siano effettuate continue chiamate a NXTSoundGetState, per verificare lo "stato" del riproduttore di suoni, così da capire quando poter procedere con l'istruzione corrispondente all'eventuale blocco NXT-G successivo. La suddetta verifica controlla se i bit del campo flag sono entrambi pari a '0': in caso di risposta affermativa, si può procedere con l'istruzione successiva. Qualora, invece, ciò non accada, si continua ad interrogare lo "stato" del riproduttore di suoni, mediante chiamate ripetute nel tempo a NXTSoundGetState, fino a quando, appunto, i bit del campo flag non sono entrambi uguali a '0', ossia il riproduttore è tornato ad essere 'idle'.

Nella situazione appena descritta il simulatore si comporta esattamente come il robot LEGO®.

Qualora, invece, non si metta la spunta su 'Wait for Completion', appena lanciata la syscall NXTSoundPlayFile o NXTSoundPlayTone, il firmware passa subito a processare il blocco successivo del programma NXT-G in questione, senza stavolta invocare NXTSoundGetState. Qualora venga processato un altro blocco relativo al riproduttore di suoni, mentre è ancora in corso la riproduzione comandata dal blocco audio precedente, il firmware del robot interrompe quest'ultima e manda in esecuzione la riproduzione sonora appena richiesta. Nel simulatore, però, ciò non accade. Assegnando, infatti, una distinta priorità ad ogni syscall audio, la seconda delle due chiamate descritte sopra viene eseguita soltanto quando la sua priorità è pari al valore di *actualPriority* specificato in *sCond*, ossia quando la prima syscall sonora ha terminato di eseguire ed ha aggiornato *actualPriority*. Tale comportamento sembra quindi, a prima vista, essere un malfunzionamento del modulo di riproduzione audio di *NXTSimulator*, ma in realtà è una semplice anomalia necessaria per evitare lo sgradevole inconveniente che sarà descritto qui di seguito.

Consideriamo la situazione in cui un programma NXT-G sia costituito da due flussi paralleli, contenenti ciascuno un blocco di riproduzione audio come primo blocco, come esemplificato nella figura che segue.

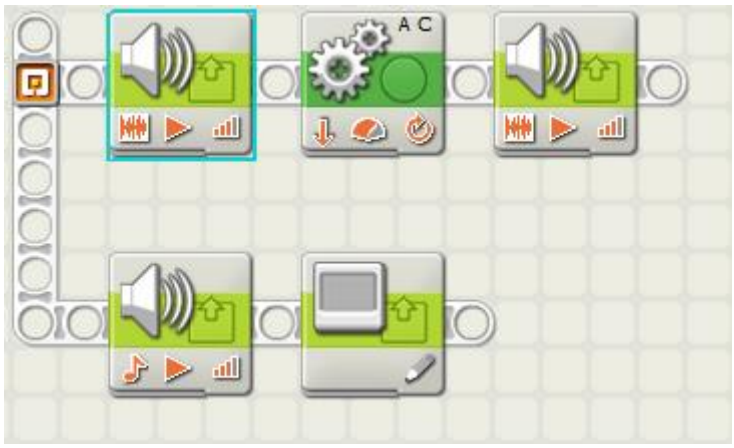


**Figura 4.8:** *Prg. NXT-G con due flussi paralleli e tre blocchi audio*

La struttura del programma sopra riportato impone l'esecuzione concomitante dei due flussi paralleli. Qualora le syscall di riproduzione audio non fossero state dotate ciascuna di un distinto valore di priorità, sarebbero stati lanciati due thread distinti di classe SoundReproducer, i quali avrebbero riprodotto quanto ad essi richiesto nello stesso istante, causando così una sovrapposizione sonora. Questa eventualità, oltre che sgradevole per l'udito dell'utente, avrebbe dato vita ad un'incongruenza con il *modus operandi* del riproduttore audio del robot LEGO®. Quest'ultimo, infatti, può riprodurre un unico suono per volta, essendo di tipo mono e non stereo. Con la soluzione adottata viene, invece, eseguita una syscall alla volta, in base all'ordine di priorità, simulando così in modo adeguato il comportamento del robot in una simile circostanza.

Si è deciso quindi di tollerare l'anomalia di cui si è discusso in precedenza, a beneficio di una condotta meno “caotica” del simulatore nel caso di programmi simili a quello di figura 4.8.

## 4.7 Implementazione Gestore di File

Tra le varie funzioni messe a disposizione dell'utente, il brick del robot LEGO® presenta anche quella di *gestione dei file*. Il firmware del brick è, infatti, dotato di alcuni moduli che permettono la creazione, la lettura, la modifica, la rinomina, o la cancellazione di semplici file di testo (in formato *.TXT*). Tali file possono essere salvati nella memoria flash del brick, il quale li gestisce mediante un semplice file system di tipo *flat*, ossia non contemplante l'organizzazione dei file stessi in gerarchie di cartelle [3t]. Essi vengono referenziati per nome, e possono avere dimensione pari allo spazio disponibile nella memoria flash.

Il meccanismo che il modulo di gestione dei file utilizza per agire sugli stessi è molto semplice: prima di poter leggere o scrivere un file, è necessario aprire un cosiddetto *handle* (letteralmente "aggancio") al file stesso, il quale è un semplice riferimento al file sul quale si desidera operare. Il firmware registra automaticamente all'interno di una comune tabella tutti gli handle aperti nel corso di un programma, così da poter avere sott'occhio una lista di tutti i file *.TXT* sui quali si sta lavorando. E' importante notare come, prima di utilizzare un determinato file per uno scopo diverso da quello per il quale lo si era inizialmente aperto (e.g: dopo aver scritto un file, lo si vuole leggere), sia necessario chiudere l'handle del file ed aprirne uno di nuovo. Al termine di ogni programma il firmware si occupa di chiudere in automatico tutti gli eventuali handle rimasti aperti.

Per realizzare le funzionalità appena presentate anche su *NXTSimulator*, si sono dovute implementare le syscall del firmware che adempiono a tale compito. Per fare ciò, si sono modificate le classi *Execution.java* e *NXTSView.java*, ed è stata introdotta una nuova classe, ossia *FileAccessMethods.java*.

Vediamo ora nel dettaglio il lavoro che si è svolto all'interno delle suddette classi.

Per poter simulare il gestore di file del robot, come appena accennato, si è dovuta arricchire Execution (classe che, come appreso nella sezione 4.5, interpreta ed esegue le istruzioni di un programma dato in pasto al simulatore) con le seguenti ulteriori nove syscall del firmware:

- **NXTFileOpenRead**: prova ad aprire un file di testo (ossia apre un handle), il cui nome è fornito all'interno del cluster di parametri della syscall, per operazioni di sola lettura. Nel suddetto cluster di parametri, oltre al già citato nome del file, sono presenti anche un campo nel quale salvare un identificatore numerico univoco per l'eventuale handle creato, ed uno nel

quale scrivere la lunghezza del file in byte, qualora esso esista. L'handle così eventualmente creato viene registrato nella tabella degli handle aperti e di cui poter usufruire in seguito;

- **NXTFileOpenWrite**: prova a creare un file testuale (creandone anche l'handle), con rispettivamente nome e lunghezza (in byte) specificati all'interno del cluster di parametri della syscall, per operazioni di scrittura. Nel suddetto cluster, oltre ai due già citati, è presente anche un campo nel quale salvare un identificatore numerico per l'handle, creato nel caso in cui un file col nome specificato non sia già esistente. Anche in questo caso, l'handle così eventualmente creato viene registrato nella tabella degli handle aperti e di cui poter disporre in seguito;
- **NXTFileOpenAppend**: prova ad aprire un file di testo già esistente (originandone anche l'handle), il cui nome è fornito all'interno del cluster di parametri della syscall, per operazioni di scrittura in calce al file stesso. Nel cluster di parametri, oltre al nome del file, sono presenti anche un campo nel quale salvare l'identificatore numerico per l'eventuale handle creato, ed uno nel quale scrivere il numero di byte che il file può ancora contenere, qualora esso esista. L'handle così eventualmente originato viene registrato nella tabella degli handle aperti e di cui poter usufruire in seguito;
- **NXTFileRead**: prova a leggere un certo numero di byte, uguale a quello specificato dall'apposito campo del cluster della syscall, dal file aperto, il cui handle è identificato dall'apposito parametro fornito nel cluster. Quest'ultimo, oltre ai campi appena visti, contiene una stringa all'interno della quale salvare i byte eventualmente letti nel file. Il campo, di cui sopra, specificante il numero di byte da leggere nel file, viene poi sovrascritto col conteggio del numero di byte che sono stati effettivamente letti;
- **NXTFileWrite**: prova a scrivere un certo numero di byte, pari a quello precisato nell'apposito campo del cluster di parametri della syscall, nel file aperto, il cui handle è identificato dall'apposito parametro fornito nel cluster. Quest'ultimo, oltre ai campi già citati, comprende anche una stringa col testo da scrivere nel file. Il campo, di cui sopra, specificante il numero di byte da scrivere nel file, viene poi sovrascritto col numero di byte che sono stati effettivamente scritti;
- **NXTFileClose**: prova a chiudere l'handle di un file, il cui identificatore

numerico è specificato all'interno del cluster di parametri della syscall;

- **NXTFileResolveHandle**: cerca il file, il cui nome è precisato nel cluster di parametri della syscall, nella lista degli handle aperti, e, qualora lo reperisca, ne restituisce l'identificatore univoco dell'handle salvandolo nell'apposito campo del suddetto cluster. Quest'ultimo comprende, oltre a quelli già citati, anche un campo booleano nel quale segnalare se l'handle eventualmente trovato è stato aperto per operazioni di lettura o scrittura;
- **NXTFileRename**: prova a cambiare il nome di un file rinominandolo. Sia il vecchio che il nuovo nome sono precisati nel cluster di parametri della syscall. L'operazione ha successo soltanto se il file col vecchio nome esiste, non ci sono handle a lui riferiti ancora aperti, e non esiste già un file denominato col nuovo nome da assegnare al file in questione;
- **NXTFileDelete**: prova a cancellare il file, il cui nome è precisato nel cluster di parametri della syscall. Tale operazione ha successo soltanto se il suddetto file esiste e non ci sono handle a lui associati ancora aperti.

Ciascuna delle chiamate a sistema appena presentate contiene, inoltre, in testa al cluster di parametri ad essa relativa, un campo per il cosiddetto *status code*. All'interno di tale campo viene salvato un codice numerico, una volta terminata l'esecuzione della syscall. Uno status code pari a '0' segnala che l'operazione in questione è andata a buon fine, mentre altri specifici valori positivi segnalano che è avvenuta una qualche anomalia durante l'esecuzione della syscall stessa (e.g: '34560' - File non trovato). Questi status code servono al firmware per evincere se una chiamata a sistema è andata a buon fine o meno, e prendere così eventualmente le dovute contromisure.

Come occorso per le syscall relative al display ed al riproduttore di suoni, anche per quelle sopra esposte la scelta per la quale si è optato è stata quella di non sovraccaricare la parte di Execution ad esse relativa di tutte le istruzioni effettivamente necessarie per simularle, ma di “spalmare” le stesse su più classi, adottando quindi un approccio più *modulare*. Per quanto riguarda, ad esempio, la chiamata *NXTFileOpenRead*, essa è stata ivi realizzata mediante l'estratto di codice a pagina seguente.

```
case 40: //OP_SYSCALL
   coppia[0] = NXTSView.sim.lett[pc + 2];
   coppia[1] = NXTSView.sim.lett[pc + 3];
   source1 = conv.getPosInt(coppia); //SysCallID
   coppia[0] = NXTSView.sim.lett[pc + 4];
   coppia[1] = NXTSView.sim.lett[pc + 5];
   source2 = conv.getPosInt(coppia); //Cluster dei Parametri di ingresso e uscita
   switch((int)source1){
      case 0:
         int priority_0 = NXTSView.incrementFileAccessThreadPriority();
         NXTSView.fAccess.fileOpenRead(priority_0, NXTSView.sim.table, (int)source2,
         NXTSView.sim.array);
         break;
            . . .
   }
            . . .
   break;
```

A partire dal codice sopra riportato, si può illustrare come sono state implementate nella loro totalità le syscall in esame. Il principio dell'esecuzione delle istruzioni ad esse relative avviene all'interno della classe Execution ed, essendo realizzato in maniera assolutamente analoga per ognuna di esse, è sufficiente descriverne l'operato di uno soltanto; nel nostro caso si è scelto appunto quello di *NXTFileOpenRead*.

Nella variabile intera *priority_0* viene salvata la priorità assegnata alla syscall in questione, mediante un meccanismo assolutamente analogo a quello presentato nelle sezioni precedenti di questo elaborato per le syscall del display e del riproduttore di suoni. Così facendo, ad ogni chiamata a sistema della famiglia in esame viene assegnata una distinta priorità; in seguito si comprenderà l'utilità di tale scelta (vedi sottosezione 4.7.10: *La Priorità delle Syscall del Gestore di File*).

All'interno della classe NXTSView è definito un oggetto denominato *fAccess* (reinizializzato all'inizio di ogni simulazione), istanza di classe FileAccessMethods. Tale classe è stata creata con lo scopo di tener traccia durante una simulazione degli handle dei file aperti sui quali si sta operando, e di eseguire sugli stessi le funzioni vere e proprie che le varie syscall del gestore di file richiedono. Ogni volta che viene lanciata una chiamata a sistema di questa famiglia è, infatti, subito invocato sull'oggetto *fAccess* il metodo relativo alla syscall in questione (*fileOpenRead()* nel caso del codice sopra proposto). Di questi metodi ne è stato implementato uno per ciascuna syscall del gestore di file e verranno ora illustrati nel dettaglio.

### 4.7.1 NXTFileOpenRead

Le funzionalità vere e proprie di questa syscall sono implementate dal metodo *fileOpenRead()* di classe FileAccessMethods, le cui istruzioni più considerevoli sono riportate qui di seguito.

```
public synchronized int fileOpenRead(int threadPriority, DSTOC[] table, int source2, ArrayList[]
array){
        . . .
    int fileHandle = -1;
    long length = -1;
        . . .
    boolean exists = (new File(fileName)).exists();
    if(exists){
        fileList.add(new File(fileName));
        fileHandle = fileList.size()-1;
        length = ((File)fileList.get(fileHandle)).length();
        try{
            FileReader fR = new FileReader((File)fileList.get(fileHandle));
            fileOpenMode.add(fR);
            statusCode = 0; //Success
        }catch(IOException e){ statusCode = 35328; } //Generic Error
    }else{ statusCode = 34560; } //File Not Found
    table[source2+1].setValDefault(statusCode);
    table[source2+2].setValDefault(fileHandle);
    table[source2+5].setLongValDefault(length);
        . . .
    return statusCode;
}
```

Vediamo ora di descrivere i passaggi salienti eseguiti dal metodo qui sopra proposto.

Come prima cosa viene verificato mediante il metodo *exists()* di classe *java.io.File* [16s] se effettivamente esiste il file col nome desiderato, che si vuole aprire in lettura: qualora tale verifica dia esito positivo, si crea un'istanza di classe File per incapsulare il file da aprire, e la si aggiunge in coda alla lista *fileList*, istanza di classe *java.util.ArrayList* [16s], la quale contiene tutte le istanze di File relative a file ancora aperti; si ricavano quindi l'identificatore dell'handle del file medesimo e la dimensione dello stesso in byte. Fatto ciò, viene poi effettuata l'apertura vera e propria del file in sola lettura, incapsulando a sua volta l'istanza di classe File, nella

quale era stato precedentemente già incapsulato, all'interno di un'istanza di classe *java.io.FileReader* [16s]. Quest'ultima viene inserita nella lista *fileOpenMode*, istanza di ArrayList, contenente tutte le istanze di FileReader e FileWriter relative a tutti i file aperti. Così facendo la syscall *NXTFileRead* potrà, una volta invocata, usufruire dei metodi offerti da FileReader per agire in lettura sul file in questione.

Una volta terminate le operazioni descritte, il metodo *fileOpenRead()* salva nel cluster relativo alla syscall NXTFileOpenRead i parametri di uscita richiesti, concludendo così la sua esecuzione.

### 4.7.2 NXTFileOpenWrite

Le attività svolte da questa chiamata a sistema sono implementate dal metodo *fileOpenWrite()* della classe FileAccessMethods, le cui istruzioni più importanti sono allegate qui di seguito.

```
public synchronized int fileOpenWrite(int threadPriority, DSTOC[] table, int source2, ArrayList[]
array){
        . . .
    int fileHandle = -1;     long length = -1;
        . . .
    boolean exists = (new File(fileName)).exists();
    if(exists){ statusCode = 36608; } //File Exists
    else{ File myFile = new File(fileName);
       try{ myFile.createNewFile(); } catch(IOException e){ statusCode = 35328; } //Generic Error
       fileList.add(myFile);     fileHandle = fileList.size()-1;
       ((File)fileList.get(fileHandle)).setWritable(true);
       length = ((File)fileList.get(fileHandle)).length();
       try{ FileWriter fW = new FileWriter((File)fileList.get(fileHandle));
          fileOpenMode.add(fW);     statusCode = 0; //Success
       }catch(IOException e){ statusCode = 35328; } //Generic Error
    }
    table[source2+1].setValDefault(statusCode);
    table[source2+2].setValDefault(fileHandle);
    table[source2+5].setLongValDefault(length);
        . . .
    return statusCode;
}
```

Descriviamo ora le operazioni più significative compiute dal metodo presentato a pagina precedente.

Inizialmente si verifica mediante il metodo *exists()* di classe File se esiste già un file col nome di quello che si desidera creare per la scrittura: qualora tale verifica dia esito positivo, il metodo interrompe subito la creazione del file, e segnala in uscita l'anomalia verificatasi, mediante l'apposito status code. Qualora, invece, il file non esista già, viene creata un'istanza di classe File per incapsulare il file da originare, e la si aggiunge in coda alla lista *fileList*, istanza di classe ArrayList, e di cui si è già discusso nella sottosezione precedente. Viene quindi impostato il file come disponibile per la scrittura, e si ricavano l'identificatore dell'handle del file medesimo e la dimensione dello stesso in byte. Fatto ciò, viene poi effettuata l'apertura del file per la scrittura, incapsulando a sua volta l'istanza di classe File, nella quale era stato precedentemente già incapsulato, all'interno di un'istanza di classe *java.io.FileWriter* [16s]. Quest'ultima è accodata nella lista *fileOpenMode*, istanza di ArrayList, di cui si è già trattato nella sottosezione precedente. Così facendo la syscall *NXTFileWrite* potrà, una volta lanciata, utilizzare i metodi messi a disposizione da FileWriter per scrivere sul file in questione.

Una volta concluse le operazioni illustrate, il metodo *fileOpenWrite()* salva nel cluster relativo alla syscall NXTFileOpenWrite i parametri di uscita richiesti, portando così a termine la sua esecuzione.

### 4.7.3 NXTFileOpenAppend

Le operazioni svolte da questa syscall sono implementate dal metodo *fileOpenAppend()* della classe FileAccessMethods, le cui istruzioni più considerevoli sono proposte a pagina seguente.

```
public synchronized int fileOpenAppend(int threadPriority, DSTOC[] table, int source2,
ArrayList[] array){
        . . .
    int fileHandle = -1;
    long length = -1;
    int pos = -1;
        . . .
    boolean exists = (new File(fileName)).exists();
    if(exists){
       for(int i = 0; i < fileList.size(); i++){
          if(fileName.equals(((File)fileList.get(i)).getName())){ pos = i;     break; }
       }
       if(pos == -1){ //File da aprire in scrittura non è già aperto
          fileList.add(new File(fileName));
          fileHandle = fileList.size()-1;
          ((File)fileList.get(fileHandle)).setWritable(true);
          length = ((File)fileList.get(fileHandle)).getFreeSpace();
          try{
             FileWriter fW = new FileWriter((File)fileList.get(fileHandle), true); //Append data
             fileOpenMode.add(fW);
             statusCode = 0; //Success
          }catch(IOException e){ statusCode = 35328; } //Generic Error
       }else{ statusCode = 35584; } //File Busy
    }else{ statusCode = 34560; } //File Not Found
    table[source2+1].setValDefault(statusCode);
    table[source2+2].setValDefault(fileHandle);
    table[source2+5].setLongValDefault(length);
        . . .
    return statusCode;
}
```

Vengono ora definite le principali attività effettuate dal metodo presentato qui sopra.

Al principio si verifica grazie al metodo *exists()* di classe File se effettivamente esiste il file col nome specificato, ed in calce al quale si desidera effettuare un'operazione di scrittura: qualora tale verifica dia esito positivo, il metodo prosegue con le operazioni successive; in caso contrario, invece, esso interrompe subito l'apertura del file, e segnala in uscita l'anomalia verificatasi, impostando l'apposito status code. Nel caso il file in questione esista, prima di procedere con l'apertura vera e propria dello stesso, viene fatta un'ulteriore verifica, ossia che esso non sia già stato

aperto per altre operazioni da qualche altra syscall. Se anche questa verifica va a buon fine, si incapsula il file all'interno di un'istanza di classe File, impostandolo come pronto per la scrittura, lo si aggiunge alla lista *fileList* dei file aperti, si ricava l'identificatore dell'handle assegnato al file, e si ricava il numero di byte ancora liberi in esso. Successivamente si incapsula l'oggetto di classe File così ottenuto all'interno di un'istanza di FileWriter, specificando che i dati, che saranno eventualmente scritti su di esso mediante un'operazione di scrittura, saranno aggiunti in calce a quelli già presenti nel file medesimo. Fatto ciò, il metodo inserisce l'oggetto così ottenuto in coda alla lista *fileOpenMode*, preparando in tal modo il file in questione per la scrittura di dati mediante i metodi messi a disposizione da FileWriter, ed invocati da una successiva chiamata a *NXTFileWrite*.

Una volta terminate le operazioni descritte, il metodo *fileOpenAppend()* salva nel cluster relativo alla syscall NXTFileOpenAppend i parametri di uscita richiesti, concludendo così la sua esecuzione.

### 4.7.4 NXTFileRead

Le attività compiute da questa chiamata a sistema sono implementate dal metodo *fileRead()* di classe FileAccessMethods, le cui istruzioni più rilevanti sono riportate a pagina seguente.

```
public synchronized int fileRead(int threadPriority, DSTOC[] table, int source2, ArrayList[] array,
int nArray){
        . . .
    int fileHandle = (int)table[source2+2].getValDefault();
    long length = table[source2+5].getLongValDefault();
    String buffer = "";
        . . .
    ArrayList temp = new ArrayList();
        . . .
    array[dest] = temp;
    int i = 0; //Contatore di byte letti nel File
    if(fileHandle != -1){ //File è già aperto
       try{
          for(i = 0; i < length; i++){
             int asciiCode = (Integer)((FileReader)fileOpenMode.get(fileHandle)).read();
             buffer += new Character((char) asciiCode).toString();
          }
          ((FileReader)fileOpenMode.get(fileHandle)).mark(i); //Segna fino a che carattere  si è
                                                              arrivati a leggere
       }catch(IOException e){ statusCode = 35328; } //Generic Error
       if(i == length){ statusCode = 0; } //Success - tutti i dati letti
       else{ statusCode = 35328; } //Generic Error
       for(int j = 0; j < i; j++){ //Riempie array (stringa) di destinazione
          array[dest].add(buffer.getBytes()[j]);
       }
       array[dest].add((byte) 0);
    }else{ statusCode = 34816; } //File Closed
    table[source2+1].setValDefault(statusCode);
    table[source2+2].setValDefault(fileHandle);
    table[source2+5].setLongValDefault(i);
        . . .
    return statusCode;
}
```

Vengono ora descritte le operazioni più importanti svolte dal metodo presentato qui sopra.

All'inizio si verifica se è presente un handle aperto per il file di testo che si desidera leggere: se tale verifica da esito positivo, il metodo prova a leggere (grazie al metodo *read()* dell'oggetto di classe FileReader, nel quale era stato precedentemente incapsulato il file) dal file medesimo il numero di caratteri in codice ASCII [15s] specificato in ingresso, salvandoli all'interno della stringa *buffer*. Terminata

l'operazione di lettura, si inseriscono i codici ASCII dei caratteri letti nel file all'interno della stringa di destinazione presente nel cluster di parametri della syscall, e gestita sotto forma di array di byte, come da specifiche firmware (vedi sottosezione 3.1.5).

Una volta concluse le operazioni illustrate, il metodo *fileRead()* salva nel cluster relativo alla syscall NXTFileRead gli altri parametri di uscita richiesti, portando così a termine la sua esecuzione.

### 4.7.5 NXTFileWrite

Le operazioni svolte da questa syscall sono implementate dal metodo *fileWrite()* della classe FileAccessMethods, le cui istruzioni più importanti sono allegate qui di seguito.

```
public synchronized int fileWrite(int threadPriority, DSTOC[] table, int source2,ArrayList[] array){
        . . .
    int fileHandle = (int)table[source2+2].getValDefault();
    long length = table[source2+5].getLongValDefault();
        . . .
    int i = 0; //Contatore di byte scritti nel File di destinazione
    if(fileHandle != -1){ //File è già aperto
        try{
            for(i = 0; i < length; i++){
                char character = buffer.charAt(i);
                ((FileWriter)fileOpenMode.get(fileHandle)).write(character);
            }
        }catch(IOException e){ statusCode = 35328; } //Generic Error
        if(i == length){ statusCode = 0; } //Success - tutti i dati scritti
        else{ statusCode = 33792; } //Partial Write
    }else{ statusCode = 34816; } //File Closed
    table[source2+1].setValDefault(statusCode);
    table[source2+2].setValDefault(fileHandle);
    table[source2+5].setLongValDefault(i);
        . . .
    return statusCode;
}
```

Sono ora illustrate le attività più degne di nota eseguite dal metodo riportato a pagina precedente.

Come prima cosa si verifica se è presente un handle aperto per il file di testo sul quale si intende andare a scrivere: qualora tale verifica dia esito positivo, il metodo prova a scrivere (grazie al metodo *write()* dell'oggetto di classe FileWriter, nel quale era stato precedentemente incapsulato il file) nel file medesimo il numero di caratteri in codice ASCII [15s] specificato in ingresso, e contenuti all'interno della stringa *buffer*, presente nel cluster di parametri della syscall. Successivamente, una volta terminate le operazioni appena esposte, il metodo *fileWrite()* salva nel cluster relativo alla chiamata NXTFileWrite i parametri di uscita richiesti, concludendo così la sua esecuzione.

### 4.7.6 NXTFileClose

Le attività eseguite da questa chiamata a sistema sono implementate dal metodo *fileClose()* di classe FileAccessMethods, le cui istruzioni più considerevoli sono proposte qui di seguito.

```
public synchronized int fileClose(int threadPriority, DSTOC[] table, int source2,ArrayList[] array){
        . . .
    int fileHandle = (int)table[source2+2].getValDefault();
    if(fileHandle != -1){ //Il File che si desidera chiudere è aperto
       try{ ((FileWriter)fileOpenMode.get(fileHandle)).close();
          fileOpenMode.remove(fileHandle);
          fileList.remove(fileHandle);
          statusCode = 0; //Success
       }catch(ClassCastException cCE){
          try{ ((FileReader)fileOpenMode.get(fileHandle)).close();
             fileOpenMode.remove(fileHandle);
             fileList.remove(fileHandle);
             statusCode = 0; //Success
          }catch(IOException e){ statusCode = 35328; } //Generic Error
       }catch(IOException e){ statusCode = 35328; } //Generic Error
    }else{ statusCode = 34816; } //File Closed
    table[source2+1].setValDefault(statusCode);
        . . .
    return statusCode;
}
```

Vengono ora presentate le operazioni più rilevanti eseguite dal metodo proposto a pagina precedente.

Come primo passo si verifica se è presente un handle aperto per il file di testo che si intende chiudere: qualora tale verifica dia esito positivo, si procede con la chiusura dello stesso, rimuovendo anche gli oggetti di classe FileWriter, (se il file era stato aperto in scrittura) o FileReader (se il file era stato aperto in lettura), e File, nei quali era stato incapsulato il file, rispettivamente dalla lista *fileOpenMode* e *fileList*. In seguito, una volta concluse le attività appena descritte, il metodo *fileClose()* salva nel cluster relativo alla chiamata NXTFileClose lo status code appropriato, terminando poi la sua esecuzione.

### 4.7.7 NXTFileResolveHandle

Le operazioni espletate da questa syscall sono implementate dal metodo *fileResolveHandle()* della classe FileAccessMethods, le cui istruzioni più significative sono riportate qui di seguito.

```
public synchronized int fileResolveHandle(int threadPriority, DSTOC[] table, int source2,
ArrayList[] array){
        . . .
    int fileHandle = -1;     int writeHandle = -1;     statusCode = 34816; //File Closed
        . . .
    for(int i = 0; i < fileList.size(); i++){
      if(fileName.equals(((File)fileList.get(i)).getName())){     fileHandle = i;
        try{
          FileWriter fW = (FileWriter) fileOpenMode.get(i);     writeHandle = 1;
        }catch(ClassCastException cCe){
          FileReader fR = (FileReader) fileOpenMode.get(i);     writeHandle = 0;
        }
        statusCode = 0; //Success     break;
      }
    }
    table[source2+1].setValDefault(statusCode);
    table[source2+2].setValDefault(fileHandle);
    table[source2+3].setValDefault(writeHandle);
        . . .
    return statusCode;
}
```

Sono ora esposte le attività più importanti svolte dal metodo allegato a pagina precedente.

All'inizio si verifica se all'interno della lista dei file aperti, ossia *fileList*, è presente l'handle relativo al file di cui si desidera ricavare l'identificatore dell'handle medesimo: qualora tale verifica dia esito positivo, si salva nella variabile *fileHandle* l'indice (l'identificatore) all'interno della suddetta lista, al quale reperire l'handle richiesto; tale valore verrà poi inserito tra i valori di ritorno del cluster di parametri della chiamata a sistema in questione. Sempre nel caso in cui la verifica di cui sopra vada a buon fine, viene successivamente controllato se l'handle è stato aperto per la scrittura o la lettura del file, e l'esito di tale verifica viene anch'esso posto tra i valori di ritorno del cluster di parametri della syscall. Al termine delle operazioni appena illustrate, il metodo *fileResolveHandle()* salva nel suddetto cluster lo status code di ritorno appropriato ed i parametri sopra citati, concludendo poi la sua esecuzione.

### 4.7.8 NXTFileRename

Le operazioni svolte da questa chiamata a sistema sono implementate dal metodo *fileRename()* di classe FileAccessMethods, le cui istruzioni più rilevanti sono allegate qui di seguito.

```
public synchronized int fileRename(int threadPriority, DSTOC[] table, int source2, ArrayList[]
array){          . . .
    int pos = -1;     boolean exists = (new File(oldFileName)).exists();
    boolean existsNewFile = (new File(newFileName)).exists();
    if(existsNewFile){ statusCode = 36608; } //File Exists
    if(exists && !existsNewFile){     File myFile = new File(oldFileName);
       for(int i = 0; i < fileList.size(); i++){
          if(oldFileName.equals(((File)fileList.get(i)).getName())){ pos = i; break; }
       }
       if(pos == -1){ //File da rinominare non è aperto
          myFile.renameTo(new File(newFileName));  statusCode = 0; //Success
       }else{ statusCode = 35584; } //File Busy
    }else{ if(!exists){ statusCode = 34560; } } //File Not Found
    table[source2+1].setValDefault(statusCode);
       . . .
    return statusCode;
}
```

Vengono ora descritte le attività più considerevoli effettuate dal metodo proposto a pagina precedente.

Come prima cosa si controlla se effettivamente esiste il file che si desidera rinominare, e che non esista già un altro file avente un nome identico a quello nuovo che si intende assegnare al file da rinominare. Qualora tali verifiche diano entrambe esito positivo, si procede con un ulteriore controllo: si verifica, infatti, che non vi sia un handle aperto per il file da rinominare. Se anche quest'ultima verifica da esito positivo, si procede con la modifica vera e propria del nome del file, invocando il metodo *renameTo()* dell'oggetto di classe File, nel quale è incapsulato il file medesimo, fornendogli come parametro il nuovo nome da assegnare a quest'ultimo. In seguito, una volta concluse le attività appena illustrate, il metodo *fileRename()* salva nel cluster relativo alla chiamata NXTFileRename lo status code appropriato, terminando poi la sua esecuzione.

### 4.7.9 NXTFileDelete

Le attività espletate da questa syscall sono implementate dal metodo *fileDelete()* di classe FileAccessMethods, le cui istruzioni più significative sono riportate qui di seguito.

```
public synchronized int fileDelete(int threadPriority, DSTOC[] table, int source2, ArrayList[]
array){
        . . .
    int pos = -1;     boolean exists = (new File(fileName)).exists();
    if(exists){
       File myFile = new File(fileName);
       for(int i = 0; i < fileList.size(); i++){
          if(fileName.equals(((File)fileList.get(i)).getName())){ pos = i; break; }
       }
       if(pos == -1){ //File da eliminare non è aperto
          myFile.delete();     statusCode = 0; //Success
       }else{ statusCode = 35584; } //File Busy
    }else{ statusCode = 34560; } //File Not Found
    table[source2+1].setValDefault(statusCode);
        . . .
    return statusCode;
}
```

Sono ora presentate le operazioni più importanti eseguite dal metodo riportato a pagina precedente.

All'inizio si controlla che effettivamente esista il file che si desidera eliminare: qualora tale verifica dia esito positivo, si svolge un'ulteriore controllo, ossia che il file in questione non abbia un handle aperto ad esso relativo. Se anche questa seconda verifica va a buon fine, si procede quindi con l'effettiva rimozione del file, per mano del metodo *delete()* dell'oggetto di classe File, nel quale lo si era incapsulato precedentemente. Una volta cancellato il file desiderato, il metodo *fileDelete()* salva nel cluster relativo alla chiamata NXTFileDelete lo status code appropriato, concludendo così la sua esecuzione.

Tutti i file in formato *.TXT*, con cui hanno a che fare le chiamate a sistema della famiglia in questione, sono salvati nel percorso seguente del file di distribuzione di *NXTSimulator*: *..\output\FileTXT*.

### 4.7.10 La Priorità delle Syscall del Gestore di File

Adesso che si è trattato il modo in cui si sono state implementate le chiamate a sistema del gestore di file su *NXTSimulator*, resta ancora da dipanare la questione relativa alla priorità che viene assegnata a ciascuna di esse, appena viene invocata all'interno della classe Execution.

Il meccanismo che verrà delineato è del tutto analogo a quello adottato nel caso della priorità delle syscall del display e del riproduttore di suoni (vedi rispettivamente sezioni 4.5 e 4.6.1).

Come già accennato in precedenza, appena viene lanciata, ogni chiamata appartenente alla famiglia delle syscall del gestore di file riceve un determinato e distinto valore di priorità. La classe FileAccessMethods gestisce, a sua volta, un altro valore di priorità collegato a quello appena citato: tale valore numerico si chiama *actualPriority* (il cui valore è impostato ad '1' all'inizio di ogni simulazione). Quest'ultimo specifica la priorità della successiva syscall del gestore di file che ha il diritto di eseguire le operazioni di cui essa è responsabile. Nel momento in cui una delle syscall in analisi, dall'interno della classe Execution, invoca il rispettivo metodo di classe FileAccessMethods ad essa corrispondente, quest'ultimo viene eseguito soltanto se la priorità della syscall invocante combacia col valore di *actualPriority*; qualora ciò non accada, la syscall si mette in attesa del proprio "turno". Solo quando

esso arriverà, la syscall potrà effettivamente eseguire le azioni da lei previste. Ogni volta che una syscall del tipo in analisi conclude il proprio operato, infatti, essa incrementa il valore di *actualPriority*, aggiornandolo alla priorità dell'eventuale successiva syscall di quel tipo da mandare in esecuzione, e notifica le eventuali syscall in attesa dell'evento appena verificatosi.

Con la soluzione adoperata le syscall vengono così eseguite una alla volta, nella sequenza stilata in base alla priorità attribuita a ciascuna di esse.

Se non si fosse proceduto nel modo appena descritto, due distinte syscall del gestore di file, presenti su due flussi distinti, ma eseguiti in parallelo, di uno stesso programma NXT-G, se eseguite in concomitanza avrebbero colliso tra di loro, potendo causare un comportamento anomalo nella gestione dei file di testo interessati.

Per controllare se una syscall del gestore di file ha la priorità idonea per essere mandata in esecuzione, sono state inserite le seguenti due istruzioni di controllo al principio di tutti i rispettivi metodi della classe FileAccessMethods invocati dalle syscall della famiglia in analisi al momento del lancio.

```
while(threadPriority != actualPriority)
    try{ wait(); } catch(InterruptedException e){}
```

Per aggiornare, invece, il valore di *actualPriority* e notificare eventuali syscall, in attesa di essere eseguite, che potrebbe essere giunto il loro turno, sono state inserite le seguenti due istruzioni al termine di ciascuno dei metodi di classe FileAccessMethods invocati dalle syscall della famiglia in analisi al momento del lancio nella classe Execution.

```
setActualPriority();
this.notifyAll();
```

## 4.8 Implementazione Bottoni del Brick

Il brick del robot Lego® Mindstorms® NXT è provvisto tra le altre cose di quattro *bottoni* in gomma, come si può anche notare nella figura 2.1 di pagina 20. Il bottone grigio scuro posto più in basso rispetto a tutti gli altri, qualora premuto, svolge la funzione di interruzione del programma che il robot sta eventualmente eseguendo. Gli altri tre pulsanti, invece, possono essere utilizzati dall'utente per interagire col robot stesso per diversi motivi. Un particolare programma che il robot sta eseguendo può, ad esempio, trovarsi ad un certo punto a scegliere quale tra due flussi distinti del programma medesimo eseguire (e.g: costrutto 'switch' di NXT-G): si può ipotizzare che venga eseguito il primo dei due flussi qualora sia stato premuto e rilasciato almeno una volta (dall'ultimo 'reset' - il cui significato lo si vedrà in seguito - del bottone stesso) il tasto più a destra del brick, ed il secondo in caso contrario. Quello presentato è soltanto un banale esempio di utilizzo dei bottoni del brick, ma è sufficiente per far comprendere la loro utilità al fine di permettere l'interazione tra robot ed utente.

Per dotare *NXTSimulator* di tali pulsanti, si è scelto di aggiungere al *javax.swing.JPanel* creato dalla classe *NXTSView.java* quattro istanze di *javax.swing.JButton*, una per ciascun bottone del brick, denominate abbastanza intuitivamente *leftArrow*, *centerSquare*, *rightArrow* e *bottomButton*. Il risultato così ottenuto si può osservare all'interno del riquadro verde della figura 4.2 di pagina 51.

Ora che è stato delineato l'aspetto strettamente grafico della realizzazione dei pulsanti del brick nel simulatore, si può passare alla descrizione delle righe di codice che hanno reso possibile l'implementazione vera e propria delle funzionalità di questi pulsanti. Tali funzionalità sono state ottenute modificando la classe *Execution.java* del progetto *NXTSimulator* e la già citata *NXTSView.java*; è stato, inoltre, necessario realizzare una nuova classe: *ButtonMethod.java*.

Per poter simulare i bottoni del brick, si è dovuta arricchire la parte di Execution (classe che, come visto nella sezione 4.5, interpreta ed esegue le istruzioni di un programma dato in pasto al simulatore) adibita ad eseguire le syscall del firmware con la seguente nuova chiamata a sistema:

- **NXTReadButton**: permette di leggere lo “stato” dei tre bottoni superiori del brick del robot. Nel cluster di parametri ad essa associato si trovano nell'ordine l'indice corrispondente al bottone di cui leggere lo stato, una variabile booleana in cui specificare in uscita se il bottone in questione è

premuto al momento dell'esecuzione della syscall, un contatore in cui specificare in uscita il numero di volte in cui il bottone in questione è stato premuto e rilasciato dall'ultimo reset del contatore stesso, ed un'altra variabile booleana che specifica se resettare o meno il contatore appena citato, dopo averne letto il valore.

Come un lettore particolarmente attento avrà notato, la syscall appena descritta considera soltanto i tre bottoni superiori del simulatore (e quindi anche del robot LEGO®); il bottone inferiore, infatti, qualora venga premuto, provoca appunto (come da specifiche firmware) l'arresto immediato dell'eventuale programma/simulazione in corso di esecuzione. Ciò avviene grazie all'invocazione, alla pressione dello stesso, del metodo per l'arresto della simulazione, implementato all'interno della classe NXTSView, *stopSim()*.

Anche per questa chiamata a sistema la scelta per la quale si è optato è stata quella di non sovraccaricare la parte di Execution ad essa relativa di tutte le istruzioni effettivamente richieste per simularla, ma di "spalmare" le stesse su più classi, adottando quindi un approccio più *modulare* . Essa è stata, infatti, ivi realizzata col solo codice sorgente che segue.

```
case 40: //OP_SYSCALL
   coppia[0] = NXTSView.sim.lett[pc + 2];
   coppia[1] = NXTSView.sim.lett[pc + 3];
   source1 = conv.getPosInt(coppia); //SysCallID
   coppia[0] = NXTSView.sim.lett[pc + 4];
   coppia[1] = NXTSView.sim.lett[pc + 5];
   source2 = conv.getPosInt(coppia); //Cluster dei Parametri di ingresso e uscita
   switch((int)source1){
            . . .
      case 20:
         NXTSView.buttonM.readBrickButton(NXTSView.sim.table,(int)source2);
         break;
            . . .
   }
            . . .
   break;
```

La struttura della syscall è del tutto simile a quella delle chiamate a sistema che si sono descritte nelle sezioni precedenti di questo capitolo. L'unica istruzione del

codice riportato a pagina precedente che merita particolare attenzione è quella successiva a '*case 20*'. Essa consiste in un'invocazione del metodo *readBrickButton()*, di cui si vedrà presto il ruolo ricoperto.

All'interno della classe NXTSView è definito un oggetto denominato *buttonM*, istanza di classe ButtonMethod. Tale classe è stata creata con lo scopo di salvare, una volta che il simulatore è stato avviato, lo "stato" dei tre bottoni superiori del brick presenti nell'interfaccia dello stesso. Nella fattispecie, per ognuno di essi si memorizza all'interno di una variabile booleana se il pulsante è premuto (*true*) o rilasciato (*false*), ed all'interno di un contatore il numero di volte in cui il pulsante è stato premuto e rilasciato dall'ultimo reset del contatore stesso. Ogni volta che l'utente compie un'azione su di un certo pulsante, vengono aggiornati di conseguenza anche i valori delle variabili ad esso relative. Appena un pulsante viene premuto con un click del mouse, la classe NXTSView invoca su *buttonM* il metodo *pressBrickButton()*, il quale modifica semplicemente lo stato del pulsante in questione da rilasciato a premuto. Quando, invece, un pulsante premuto viene rilasciato, la classe NXTSView invoca su *buttonM* il metodo *releaseBrickButton()*, il quale riporta lo stato del pulsante a rilasciato, ed incrementa di una unità il contatore relativo al pulsante in questione.

Ora che si è descritto come il simulatore aggiorna lo "stato" dei pulsanti del brick, ci si può chiedere come esso, invece, interroghi tale "stato"; la risposta a questo quesito è la seguente: attraverso il metodo *readBrickButton()*. Come già anticipato in precedenza, la syscall NXTReadButton invoca proprio il suddetto metodo, sull'oggetto *buttonM*, per leggere lo stato dei bottoni del brick.

A pagina seguente viene riportato il codice sorgente di *readBrickButton()*, nel quale sono state omesse soltanto alcune istruzioni non indispensabili al fine di comprenderne il funzionamento.

```
public synchronized int readBrickButton(DSTOC [] table, int source2){
                . . .
    index = (int)table[source2+2].getValDefault();  //indice del pulsante da 'interrogare'
    reset = (int)table[source2+5].getValDefault()==1 ? true : false; //verifica se resettare contatore
    statusCode = 0;
    switch(index){ //'switch' sull'indice del pulsante da considerare
      case 1:
        if(pressed[0]){ table[source2+3].setValDefault(1); } //verifica se pulsante è premuto...
        else{ table[source2+3].setValDefault(0); } //...o meno, e restituisce lo stato così rilevato
        table[source2+4].setValDefault(count[0]); //restituisce in uscita il valore del contatore
        if(reset){ resetCount(index); } //se richiesto, resetta il contatore
        break;
      case 2:
        if(pressed[1]){ table[source2+3].setValDefault(1); }
        else{ table[source2+3].setValDefault(0); }
        table[source2+4].setValDefault(count[1]);
        if(reset){ resetCount(index); }    break;
      case 3:
        if(pressed[2]){ table[source2+3].setValDefault(1); }
        else{ table[source2+3].setValDefault(0); }
        table[source2+4].setValDefault(count[2]);
        if(reset){ resetCount(index); }    break;
      default:
        statusCode = -1;    break;
    }
    table[source2+1].setValDefault(statusCode); //salvataggio codice di ritorno
                . . .
    return statusCode;
}
```

Il codice sopra proposto è piuttosto semplice da interpretare. Il metodo *readBrickButton()*, dopo aver ricavato dal proprio cluster di parametri di ingresso/uscita l'indice intero ('1' per la freccia destra, '2' per quella sinistra e '3' per il quadrato centrale) del pulsante di cui si desidera apprendere lo “stato”, tramite un semplice costrutto 'switch' estrapola e restituisce in uscita (salvandoli all'interno del suddetto cluster) i valori ai quali l'utente è interessato, resettando poi, se necessario, il valore del campo contatore del pulsante sotto analisi. Fatto ciò, il metodo ritorna in uscita un codice numerico (*statusCode*), il cui valore è pari a '0' se l'operazione di interrogazione dello “stato” è andata a buon fine, ed è pari a '-1' in caso contrario.

## 4.9 File di *log* dei Servomotori

Durante la fase di aggiornamento di *NXTSimulator* è emersa la necessità di poter originare durante una simulazione un file di *log* del movimento dei servomotori. Nella fattispecie la volontà è stata quella di poter creare per ogni singolo motore un semplice file *.TXT*, contenente un insieme di coppie del tipo *<istante - posizione angolare>*, che specificassero la posizione angolare (in gradi) dell'immagine rotante di un motore, utilizzata per simulare il movimento del medesimo ed esemplificata nei pannelli del riquadro 2 di figura 4.1 a pagina 50, nei rispettivi istanti di tempo (in millisecondi) successivi all'inizio della simulazione, in cui essa viene aggiornata. Così facendo, al termine della simulazione l'utente può contare su di un file di testo, contenente coppie di valori nel formato di cui sopra, e di cui poter disporre per fornirle, ad esempio, in ingresso ad un programma di disegno (*plotting*), che tracci un grafico del movimento nel tempo di un motore.

Per permettere al simulatore di poter creare file di log del tipo appena descritto, si sono dovute modificare le classi del progetto *NXTSView.java*, *CRController.java* e *MotorPanel.java*.

Si è deciso innanzitutto di ritoccare l'interfaccia grafica del pannello principale del simulatore, definita dalla classe NXTSView, in modo tale da dotarla di tre istanze di classe *javax.swing.JCheckBox* [16s], evidenziate in viola nella figura 4.2 di pagina 51, una per ciascuno dei tre motori collegabili al robot. Appena un dato motore viene aggiunto al pannello principale di *NXTSimulator*, l'istanza di JCheckBox ad esso relativa diventa subito attiva, e l'utente può scegliere se produrre (mettendo la spunta) o meno (omettendo la spunta) un file di log per il motore in questione, durante la simulazione che verrà effettuata. Qualora un motore venga rimosso dal pannello del simulatore, l'oggetto JCheckBox ad esso relativo viene subito reso inattivo, e ciò viene effettuato da delle semplici istruzioni di controllo inserite all'interno del metodo *componentRemoverMotor()* della classe CRController, invocato proprio per rimuovere un motore dal simulatore.

La classe NXTSView, oltre che dell'implementazione delle tre caselle di scelta di cui sopra, si occupa anche della definizione di tre variabili di esemplare di classe *java.io.FileWriter* [16s], una per ciascun motore, utili per scrivere gli eventuali file testuali di log richiesti dall'utente. Qualora si decida di produrre il file di log per un dato motore, al momento dell'avvio della simulazione viene creata un'istanza di

classe *java.io.File* [16s], che definisce un file di testo col nome nel formato <*ora,minuto,secondo* of *giorno-mese-anno* LogMotor(*X*)>, dove *X* specifica la porta a cui è collegato il motore in questione. Il file creato viene poi incapsulato [1t] all'interno dell'oggetto FileWriter di cui sopra, per disporre così del metodo di scrittura che quest'ultimo offre; esso viene poi utilizzato dalla classe MotorPanel, incaricata della simulazione del movimento del motore, per scrivervi di volta in volta mediante il metodo *write()* di classe FileWriter le coppie di valori nel formato discusso in precedenza. Una volta terminata la simulazione, la classe NXTSView si occupa poi di chiudere tutti file di log eventualmente prodotti, per renderli così disponibili agli scopi dell'utente. Quest'ultimo può reperire i file originati al percorso seguente della cartella di distribuzione del simulatore: ..*output**LogMotor*.

Per ulteriori informazioni riguardanti il modo in cui *NXTSimulator* simula il movimento dei servomotori, consultare l'elaborato citato al punto [6t] della bibliografia.

```
MotorA
0 0
62 0
109 0
156 0
202 0
296 0
343 39
390 73
452 100
499 121
546 208
608 213
626 213
633 238
636 263
642 285
675 306
690 327
753 0
800 0
862 0
909 0
956 0
```

**Figura 4.9:** *Estratto di un file di log di un servomotore*

## 4.10 Nuove istruzioni del Firmware

Oltre che delle molteplici chiamate a sistema, di cui si è discusso nelle sezioni precedenti di questo capitolo, il simulatore è stato arricchito anche di altre sei nuove istruzioni del firmware 1.28 del robot Lego® Mindstorms® NXT, che ancora non erano state implementate nella precedente versione, la 0.9b. Nella fattispecie esse sono state integrate all'interno della classe *Execution.java*, la quale, come già ribadito in precedenza nel corso di questo capitolo, è utilizzata da *NXTSimulator* per interpretare ed eseguire le istruzioni presenti nel file eseguibile del programma da simulare.

Tutte e sei le istruzioni in questione presentano una codifica *long*, delle caratteristiche della quale si è discusso nella sottosezione 3.2.4 di questo elaborato, e sono le seguenti:

- **OP_SQRT**: calcola la radice quadrata di un numero;
- **OP_ABS**: calcola il valore assoluto di un numero;
- **OP_STRINGTONUM**: converte un numero decimale presente all'interno di una stringa (rappresentata nel formato utilizzato dal firmware LEGO®, e di cui si è trattato nella sottosezione 3.1.5) in un numero intero;
- **OP_STRTOBYTEARR**: converte una stringa in un array di byte senza segno;
- **OP_BYTEARRTOSTR**: converte un array di byte senza segno in una stringa;
- **OP_WAIT**: impone di attendere che trascorra un certo intervallo di tempo, prima di passare all'istruzione successiva.

Saranno ora discussi i dettagli implementativi di ciascuna delle istruzioni appena brevemente illustrate.

### 4.10.1 OP_SQRT

Questa istruzione richiede due parametri in ingresso, nella fattispecie l'indirizzo di destinazione nel quale salvare il risultato dell'operazione di radice che viene effettuata, e quello del numero di cui calcolare la radice quadrata.

```
case 54: //OP_SQRT
    coppia[0] = NXTSView.sim.lett[pc + 2];
    coppia[1] = NXTSView.sim.lett[pc + 3];
    dest = conv.getPosInt(coppia); //Indirizzo in cui salvare il risultato dell'operazione
    coppia[0] = NXTSView.sim.lett[pc + 4];
    coppia[1] = NXTSView.sim.lett[pc + 5];
    source1 = conv.getPosInt(coppia); //Indirizzo in cui reperire il numero su cui operare
    NXTSView.sim.table[(int)dest].setValDefault((double)Math.
    sqrt(NXTSView.sim.table[(int)source1].getVal()));
       . . .
    break;
```

Il codice qui sopra riportato è quello che implementa l'istruzione in questione. Come si evince facilmente dall'analisi dello stesso, vengono dapprima ricavati i due parametri di cui si necessita, ossia i due indirizzi rispettivamente della destinazione in cui salvare il risultato dell'operazione di radice, e della sorgente in cui reperire il numero sul quale operare. Successivamente viene poi calcolata la radice quadrata di quest'ultimo, mediante il metodo *sqrt()* della classe *java.lang.Math* [16s], ed il risultato viene quindi salvato nell'indirizzo di destinazione di cui sopra. La classe Execution può quindi passare all'interpretazione della successiva istruzione del programma che si sta simulando.

### 4.10.2 OP_ABS

Questa istruzione necessita di due parametri in ingresso, ossia l'indirizzo di destinazione nel quale salvare il risultato dell'operazione di valore assoluto che viene effettuata, e quello della cifra di cui si desidera ricavare il valore assoluto.

```
case 55: //OP_ABS
    coppia[0] = NXTSView.sim.lett[pc + 2];
    coppia[1] = NXTSView.sim.lett[pc + 3];
    dest = conv.getPosInt(coppia); //Indirizzo in cui salvare il risultato dell'operazione
    coppia[0] = NXTSView.sim.lett[pc + 4];
    coppia[1] = NXTSView.sim.lett[pc + 5];
    source1 = conv.getPosInt(coppia); //Indirizzo in cui reperire il numero su cui operare
    NXTSView.sim.table[(int)dest].setValDefault((double)Math.
    abs(NXTSView.sim.table[(int)source1].getVal()));
        . . .
    break;
```

Il codice sorgente proposto qui sopra è quello che implementa l'istruzione in analisi. Come si deduce agevolmente dall'analisi dello stesso, si ricavano inizialmente i due parametri di cui si necessita, ossia i due indirizzi rispettivamente della destinazione in cui salvare il risultato dell'operazione di valore assoluto, e della sorgente in cui reperire il numero sul quale agire. In seguito viene poi computato il valore assoluto di quest'ultimo, mediante il metodo *abs()* della classe Math, ed il risultato viene poi salvato nell'indirizzo di destinazione citato in precedenza. La classe Execution può quindi passare all'interpretazione della successiva istruzione del programma da simulare.

### 4.10.3 OP_STRINGTONUM

Questa istruzione esige ben cinque parametri in ingresso: l'indirizzo nel quale salvare il risultato numerico dell'operazione da eseguire, l'indirizzo nel quale eventualmente salvare un indice (simbolicamente chiamato *IndexPast*), la cui utilità si evincerà in seguito, l'indirizzo in cui reperire la stringa sulla quale operare, l'indirizzo al quale trovare un indice (simbolicamente denominato *Index*), il cui utilizzo si desumerà in seguito ed, infine, l'indirizzo di un valore numerico (il cui nome simbolico è *Default*), la cui utilità la si comprenderà in seguito.

```
case 32: //OP_STRINGTONUM
    coppia[0] = NXTSView.sim.lett[pc + 2];
    coppia[1] = NXTSView.sim.lett[pc + 3];
    dest = conv.getPosInt(coppia); //Indirizzo di destinazione in cui salvare il numero convertito
    coppia[0] = NXTSView.sim.lett[pc + 4];
    coppia[1] = NXTSView.sim.lett[pc + 5];
    int indexPast = conv.getPosInt(coppia); //Indice: IndexPast
    coppia[0] = NXTSView.sim.lett[pc + 6];
    coppia[1] = NXTSView.sim.lett[pc + 7];
    source1 = conv.getPosInt(coppia); //Indirizzo stringa in cui cercare il numero da convertire
    coppia[0] = NXTSView.sim.lett[pc + 8];
    coppia[1] = NXTSView.sim.lett[pc + 9];
    index = conv.getPosInt(coppia); //Indice: Index
    if(index==65535){ //NOT_A_DS_ID
       index = 0;
    }else{ index = NXTSView.sim.table[(int)index].getValDefault(); }
    coppia[0] = NXTSView.sim.lett[pc + 10];
    coppia[1] = NXTSView.sim.lett[pc + 11];
    long def = conv.getPosInt(coppia); //Valore numerico: Default
    if(def==65535){ //NOT_A_DS_ID
       def = 0;
    }else{ def = NXTSView.sim.table[(int)def].getValDefault(); }
       . . .
    boolean integerFound = false; //'Flag' per segnalare se si trova o meno un intero valido
    int j;     int numValue = -1;
    for(j=(3+(int)index);j<NXTSView.sim.array[(int)source1].size()-1;j++){ //Scansiona la stringa
       try{
          numValue = (Integer)(NXTSView.sim.array[(int)source1].get(j));
             if((numValue >= 48) && (numValue <= 57)){ //Codici ASCII dei numeri: [0-9]
                integerFound = true;     break;
             }
       }catch(NumberFormatException nFE){ continue; }
    }
    if(integerFound == false){
       NXTSView.sim.table[(int)dest].setValDefault(def); //Utilizza valore di 'Default' se non trova
                                                          valori validi
    }else{
       NXTSView.sim.table[(int)dest].setValDefault(numValue);
       NXTSView.sim.table[indexPast].setValDefault(j+1);
    }
       . . .
    break;
```

Il codice sorgente esposto a pagina precedente è quello che implementa l'istruzione in questione. Come si evince da un'analisi dello stesso, per prima cosa si ottengono gli indirizzi di destinazione in cui salvare rispettivamente il risultato dell'operazione da effettuare e l'eventuale valore da attribuire a *IndexPast*, e quelli in cui reperire i parametri di ingresso di cui si necessita. Conclusa questa prima fase che si può considerare di "set up", si procede con le operazioni vere e proprie imposte dall'istruzione. Si inizia a scansionare la stringa in ingresso, partendo dal carattere in posizione *Index*, ed interrompendo il processo qualora ci si imbatta nel carattere di tipo numerico (ossia avente codice ASCII [15s] compreso tra i valori di 48 e 57) che si sta cercando. Terminato questo stadio di ricerca, si procede in due modi differenti, a seconda che il valore numerico ricercato sia stato trovato o meno all'interno della stringa. Qualora ci si trovi in quest'ultimo caso, viene semplicemente salvato nell'indirizzo di destinazione, di cui sopra, il valore numerico denominato *Default*; qualora, invece, la ricerca abbia dato esito positivo, nell'indirizzo di destinazione viene salvato il numero intero trovato, e nell'indirizzo denominato *IndexPast* viene salvato l'indice, nella stringa processata, che punta al carattere successivo rispetto a quello nel quale è contenuto il numero che si è trovato. La classe Execution può quindi passare all'interpretazione della successiva istruzione del programma che si sta simulando.

### 4.10.4 OP_STRTOBYTEARR

Questa istruzione richiede due parametri in ingresso, ossia l'indirizzo dell'array di destinazione nel quale salvare la stringa da trasformare in un array di byte senza segno, e quello al quale reperire la stringa medesima.

```
case 35: //OP_STRTOBYTEARR
    coppia[0] = NXTSView.sim.lett[pc + 2];
    coppia[1] = NXTSView.sim.lett[pc + 3];
    dest = conv.getPosInt(coppia); //Indirizzo array di byte senza segno destinazione
    coppia[0] = NXTSView.sim.lett[pc + 4];
    coppia[1] = NXTSView.sim.lett[pc + 5];
    source1 = conv.getPosInt(coppia); //Indirizzo dove reperire la stringa su cui operare
       . . .
    ArrayList tempr = new ArrayList();
       . . .
    for(int i=3;i<NXTSView.sim.array[(int)source1].size()-1;i++){
       int asciiCode = (Integer)NXTSView.sim.array[(int)source1].get(i);
       tempr.add(asciiCode);
    }
    NXTSView.sim.array[(int)dest] = tempr;
       . . .
    break;
```

Il codice sorgente qui sopra proposto è quello che implementa l'istruzione in analisi. Come si può notare attraverso un'analisi dello stesso, vengono inizialmente estrapolati i due parametri di ingresso di cui si necessita, ovvero i due indirizzi rispettivamente dell'array di byte senza segno, in cui salvare il risultato dell'operazione che verrà effettuata, e della stringa sulla quale eseguire la medesima. Successivamente viene creato l'array di supporto *tempr*, istanza di *java.util.ArrayList* [16s], nel quale sono poi inseriti ad uno ad uno tutti i byte senza segno corrispondenti ai codici ASCII dei caratteri della stringa da manipolare, eccezion fatta per l'ultimo byte (il cui valore è '0'), cioè il byte nullo di terminazione della stringa, del quale si è discusso nella sottosezione 3.1.5. Così facendo il contenuto di *tempr* non sarà una stringa, ma l'array di byte senza segno desiderato. L'istruzione si conclude salvando l'array così ottenuto nell'indirizzo di destinazione specificato in ingresso. La classe Execution può quindi passare all'interpretazione della successiva istruzione da simulare.

### 4.10.5 OP_BYTEARRTOSTR

Questa istruzione necessita di due parametri in ingresso, ossia l'indirizzo di destinazione nel quale salvare la stringa da ottenere, e quello dell'array i cui byte senza segno servono per originare la stringa medesima.

```
case 36: //OP_BYTEARRTOSTR
    coppia[0] = NXTSView.sim.lett[pc + 2];     coppia[1] = NXTSView.sim.lett[pc + 3];
    dest = conv.getPosInt(coppia); //Indirizzo destinazione dove salvare la stringa ottenuta
    coppia[0] = NXTSView.sim.lett[pc + 4];
    coppia[1] = NXTSView.sim.lett[pc + 5];
    source1 = conv.getPosInt(coppia); //Indirizzo dove reperire l'array di byte senza segno
       . . .
    ArrayList tempor = new ArrayList();
       . . .
    for(int i=3;i<NXTSView.sim.array[(int)source1].size();i++){
       int asciiCode = (Integer)NXTSView.sim.array[(int)source1].get(i);
       tempor.add(asciiCode);
    }
    tempor.add((byte)0); //Aggiunge terminatore nullo ('0')
    NXTSView.sim.array[(int)dest] = tempor;
       . . .
    break;
```

Il codice sorgente qui sopra allegato è quello che implementa l'istruzione in questione. Analizzando lo stesso, si deduce che vengono dapprima ricavati i due parametri di ingresso di cui si ha bisogno, ossia i due indirizzi degli array di byte senza segno necessari, cioè quello che conterrà la stringa risultante dall'operazione che verrà attuata, e quello da cui prelevare i byte necessari per originare la stringa medesima. In seguito viene istanziato l'array di supporto *tempor*, occorrenza di ArrayList, nel quale sono poi riversati ad uno ad uno tutti i byte senza segno dell'array sorgente. Terminata questa fase, viene quindi aggiunto in calce all'array ottenuto il byte '0', ovvero il byte nullo di terminazione della stringa, del quale si è discusso nella sottosezione 3.1.5, disponendo così della stringa desiderata all'interno di *tempor*. L'istruzione termina poi salvando quest'ultimo nell'indirizzo di destinazione specificato in ingresso. La classe Execution può quindi passare all'interpretazione della successiva istruzione del programma che si sta simulando.

### 4.10.6 OP_WAIT

Questa istruzione richiede due parametri in ingresso, tuttavia nel nostro contesto implementativo è sufficiente utilizzare solamente il secondo di essi, ossia quello che specifica l'intervallo di attesa (in millisecondi), che l'istruzione fa trascorrere prima di passare all'istruzione successiva del programma che si sta eseguendo.

```
case 52: //OP_WAIT
   coppia[0] = NXTSView.sim.lett[pc + 4];
   coppia[1] = NXTSView.sim.lett[pc + 5];
   source2 = conv.getPosInt(coppia); //Tempo di attesa
   if(source2 != 65535){ //source2 != NOT_A_DS_ID
      source2 = NXTSView.sim.table[(int) source2].getValDefault();
      Thread.sleep((long) (source2/NXTSView.fattoreScala));
   }
      . . .
   break;
```

Il codice sorgente qui sopra proposto è quello che implementa l'istruzione in analisi. Come si evince dall'analisi dello stesso, viene dapprima ricavato il valore del parametro di ingresso temporale di cui si necessita. Successivamente viene sospesa l'esecuzione dell'istanza di classe Execution, che sta processando il clump contenente l'istruzione in analisi, per un tempo proporzionale al parametro fornito ingresso, e dipendente anche dalla scala temporale alla quale si sta eseguendo la simulazione. Una volta trascorso il tempo di attesa imposto, l'istanza di classe Execution riprende la sua esecuzione, processando l'eventuale istruzione successiva.

## 4.11 Aggiornamento per tipologie di dati *TC_ULONG* e *TC_FLOAT*

Tra i vari aggiornamenti effettuati, la versione 0.9c di *NXTSimulator* ne ha previsto uno relativo alle tipologie di dati supportate all'interno dei programmi eseguibili in formato *.RXE* interpretabili dal simulatore. La versione 1.28 del firmware LEGO® prevede, infatti, tra le tante, anche la tipologia di dati *TC_FLOAT*, ossia numeri decimali (in virgola mobile) di 32 bit con precisione singola, la quale non era, però, supportata nella versione precedente del simulatore. Un'altra tipologia di dati contemplata dal firmware è la *TC_ULONG*, ossia numeri interi senza segno di 32 bit: quest'ultima era già supportata dalla versione 0.9b del simulatore, ma veniva gestita in modo scorretto. E' stato quindi necessario colmare le suddette lacune. Per una panoramica completa sui tipi di dati supportati dal firmware LEGO® si rimanda alla sottosezione 3.2.2 di questo elaborato.

Per realizzare quanto sopra descritto, si sono dovute apportare delle opportune modifiche alle classi *DSTOC.java*, *insDstoc.java* e *Converter.java* del simulatore.

La classe DSTOC definisce i singoli record contenuti nella Dataspace Table of Contents di un programma *.RXE*, specificandone i campi illustrati nella tabella 3.6 di pagina 40, ed uno aggiuntivo, atto a contenere l'eventuale valore numerico che il dato definito da un certo record assume.

La classe insDstoc si occupa di creare, prima di iniziare la simulazione vera e propria di un programma, l'intera DSTOC, popolandola con tutti i dati statici e dinamici (per approfondimenti a riguardo consultare la sottosezione 3.1.6) presenti nel programma da simulare, ed assegnando ai medesimi gli eventuali valori di default per essi previsti.

La classe Converter contiene dei metodi invocati per convertire i byte rappresentati in formato little-endian [13s], presenti nei file eseguibili *.RXE*, e relativi ad esempio ai valori di default di cui sopra, nei numeri corrispondenti. Le funzionalità di questa classe sono quindi fondamentali per convertire, ad esempio, i byte che definiscono il valore di default di un dato di tipo *TC_UWORD* nel valore Java di tipo *int* corrispondente.

Verranno ora descritte nel dettaglio le modifiche che sono state apportate alle classi appena brevemente presentate.

Nella versione 0.9b del simulatore, la classe DSTOC definiva una variabile di esemplare intera (del tipo Java *int*), denominata *valoreDefault*, atta a contenere l'eventuale valore numerico che il dato definito da un certo record assumeva. Essendo di tipo *int* [1t], la variabile sopra citata poteva gestire solamente numeri di tipo intero e nel seguente intervallo: [-2147483648 ; 2147483647]. I dati di tipo TC_ULONG venivano gestiti anch'essi come *int*, e ciò non era corretto: un numero del tipo in questione con tutti i bit della propria rappresentazione binaria pari ad '1' corrisponde, infatti, al valore intero '4294967295', di gran lunga maggiore al massimo numero rappresentabile con una variabile di tipo *int*. La gestione dei dati di tipo TC_ULONG come *int* poteva quindi provocare un'eventuale situazione di cosiddetto *overflow* [1t]. I dati di tipo TC_FLOAT non erano, invece, come detto, gestiti da *NXTSimulator* 0.9b ed, essendo di tipo decimale, non sarebbero stati nemmeno questi gestibili mediante una variabile intera di tipo *int*.

Per consentire alla classe DSTOC di trattare correttamente i dati di tipo TC_ULONG, e di gestire anche quelli di tipo TC_FLOAT, si sono dovute aggiungere due ulteriori variabili di esemplare (rispettivamente dei tipi Java *long* e *float*), eloquentemente chiamate *valoreDefaultLong* e *valoreDefaultFloat*, da utilizzare per salvarvi all'occorrenza l'eventuale valore numerico da assegnare al dato definito dal record di tipo rispettivamente TC_ULONG e TC_FLOAT in questione. All'interno della medesima classe erano presenti anche un metodo per l'impostazione del valore numerico da assegnare ad un determinato dato (*setValDefault()*), ed uno che, invece, restituiva tale valore (*getValDefault()*). Per supportare correttamente anche i dati di tipo TC_ULONG e TC_FLOAT, questi metodi si sono rivelati, però, obsoleti, in quanto essi permettevano rispettivamente l'assegnazione e la restituzione soltanto di numeri di tipo *int*. Per ovviare al suddetto inconveniente, si è modificato setValDefault(), in modo tale che esso accetti in ingresso dati del tipo Java *double* e non più *int*, e che operi nel modo seguente: qualora il dato su cui viene invocato il metodo sia di tipo TC_ULONG, il valore numerico fornitogli come parametro di ingresso viene assegnato alla variabile *valoreDefaultLong*; qualora, invece, il dato sia di tipo TC_FLOAT, esso viene assegnato alla variabile *valoreDefaultFloat*; qualora, infine, il dato non sia di alcuno dei due tipi di cui sopra, esso viene assegnato alla variabile *valoreDefault*.

```
public void setValDefault(double vD){
    if(tipo.equals("TC_Float")){
        valoreDefaultFloat = (float) vD;
    }else if(tipo.equals("TC_ULong")){
        valoreDefaultLong = (long) vD;
    }else{ valoreDefault = (int) vD; }
}
```

Il metodo getValDefault() è stato, invece, rimpiazzato dal metodo *getVal()*, il quale ritorna in uscita il valore di una delle tre variabili di cui sopra, scegliendo sulla base del tipo di dato in questione, come si può agevolmente intuire analizzando il codice sorgente del metodo, riportato qui sotto.

```
public double getVal(){
    if(tipo.equals("TC_Float")){
        return valoreDefaultFloat;
    }else if(tipo.equals("TC_ULong")){
        return valoreDefaultLong;
    }else{ return valoreDefault; }
}
```

I metodi della classe insDstoc per l'assegnazione degli eventuali valori di default a dati statici e dinamici sono stati semplicemente arricchiti di semplici istruzioni di controllo, che possono così assegnare (mediante il metodo setValDefault()) valori di default anche ai nuovi dati di tipo TC_FLOAT, oltre che a quelli già supportati in precedenza. Per portare a termine tale processo di assegnamento dei valori di default a tutti i dati che li possiedono, la classe insDstoc, oltre che di setValDefault(), si avvale anche dell'operato di alcuni metodi di conversione della classe Converter, di cui si è brevemente discusso in precedenza. Quest'ultima era, però, sprovvista di un metodo che convertisse i numeri di tipo TC_FLOAT dalla rappresentazione little-endian utilizzata nei file *.RXE* a quella Java di tipo *float*; si è resa quindi necessaria l'implementazione di un metodo che adempiesse a questo compito: esso è stato denominato *getFloatSingleReal()*, ed il codice sorgente ad esso relativo è allegato nelle due pagine seguenti.

```
public float getFloatSingleReal(byte[] in){
    . . .
  byte mask = Byte.valueOf("-128"); //Maschera (1 0 0 0 0 0 0 0) per scoprire il segno del num.
  String segno = ""; //Stringa che conterrà il segno del numero
  int sign = in[3];  //Byte più significativi del numero float, espressi come intero
  if((mask & sign) == 0) //Calcola segno del numero
  { segno += "+"; } else { segno += "-"; }
  int num1 = provv[3]*256 + provv[2]; //Due byte più signif. del numero, espressi come intero
  int num2 = provv[1]*256 + provv[0]; //Due byte meno signif. del numero, espressi come intero
  byte[] arr1 = bitConverter(num1); //Trova rappr. binaria dei due byte più signif. del numero
  byte[] arr2 = bitConverter(num2); //Trova rappr. binaria dei due byte meno signif. del numero
  byte[] bit = new byte[32]; //Fondo arr1 ed arr2, trovando così la rappresentazione binaria
                             completa del numero float
  for(int i=0; i<16; i++){ bit[i] = arr1[i]; }
  for(int i=16; i<32; i++){ bit[i] = arr2[i-16]; }
  int index = 31; //Indice che scorre nell'array dei bit del numero, partendo dal meno signif.
  int mantissa = 0;
  while(index > 8){ //Calcola la parte frazionaria della mantissa del numero float
    mantissa += bit[index]*(Math.pow(2, (31-index)));
    index--;
  }
  int mFraz = mantissa; //Contiene la parte frazionaria della mantissa
  int e = 0;
  int indice = 0;
  while(index > 0){ //Calcola l'esponente del numero float
    e += bit[index]*(Math.pow(2,indice++));
    index--;
  }
  int esponente = e - 127; //Valore finale dell'esponente
  double numero = 0;
  String num = "";
  float v = 0;
  if((e == 0) && (mFraz == 0)){ //Mantissa ed 'e' nulli => numero = 0
    num = segno + Double.toString(numero);
    v = Float.valueOf(num);
  }else if((e == 0) && (mFraz != 0)){ //Mantissa non nulla ed 'e' nullo => numero con mantissa
                                      non normalizzata (parte intera = 0)
    esponente = -126;
    String mant = "0." + Integer.toString(mantissa);
    numero = (Float.valueOf(mant)) * (Math.pow(2, (-23))) * (Math.pow(2, esponente));
    num = segno + Double.toString(numero);
    v = Float.valueOf(num);
```

```
    }else if((segno.equals("+")) && (e == 255) && (mFraz == 0)){ //Mantissa nulla ed 'e'=255 =>
                                                                    numero = +infinito
      numero = Float.POSITIVE_INFINITY;
      num = Double.toString(numero);
      v = Float.valueOf(num);
    }else if((segno.equals("-")) && (e == 255) && (mFraz == 0)){ //Mantissa nulla ed 'e'=255 =>
                                                                    numero = -infinito
      numero = Float.NEGATIVE_INFINITY;
      num = Double.toString(numero);
      v = Float.valueOf(num);
    }else if((e == 255) && (mFraz != 0)){ //Mantissa non nulla ed 'e' = 255 => numero = NaN
      numero = Float.NaN;
      num = Double.toString(numero);
      v = Float.valueOf(num);
    }else{ //Numero float 'normale' => calcola il suo valore
      mantissa += Math.pow(2, 23);
      numero = mantissa * (Math.pow(2, (-23))) * (Math.pow(2, esponente));
      num = segno + Double.toString(numero);
      v = Float.valueOf(num);
    }
    return v;
}
```

Come si evince dall'analisi del codice sorgente appena proposto, il metodo getFloatSingleReal() inizia computando il segno del numero decimale che si desidera ricavare. Successivamente esso passa dalla rappresentazione little-endian del numero a quella binaria di 32 bit a mantissa ed esponente [4t], salvandola all'interno di un array, il quale contiene il bit più significativo del numero (bit di segno) nella cella di indice 0, e quello meno significativo in quella di indice 31. Una volta ottenuta, la suddetta rappresentazione binaria viene utilizzata dal metodo per ricavare il valore effettivo del numero *float*, secondo le regole dei numeri rappresentati in quel modo, e restituirlo poi come valore di ritorno.

La suddetta rappresentazione è chiamata anche *floating-point* (ossia in virgola mobile, o standard IEEE 754): essa prevede la rappresentazione binaria del numero in questione secondo lo schema riportato a pagina seguente.

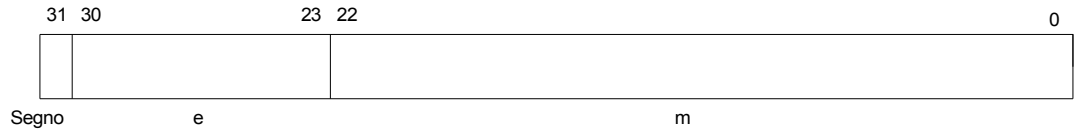


**Figura 4.10:** *Rappresentazione a mantissa ed esponente di un numero*

Come si può vedere qui sopra, il bit più significativo della sequenza (bit nella posizione 31) è il bit che specifica il segno del numero ('0' sta per '+' ed '1' sta per '-'). I bit dal 30 al 23, e denotati con *e* in figura, servono per calcolare il cosiddetto esponente, di cui si vedrà tra poco l'utilità. I bit dal 22 allo 0, ed indicati con *m* in figura, servono, invece, per calcolare il valore della cosiddetta mantissa. Il valore effettivo del numero *R* rappresentato in tale forma è dato dalla formula seguente:

$$R = M * 2^E$$

dove *M* rappresenta la mantissa, ed il proprio valore è dato dalla seguente:

$$M = \{segno\}1.m$$

ed *E* è chiamata *caratteristica*, ed il proprio valore è dato dalla seguente:

$$E = e - 127$$

Come si può intuire quindi, a seconda dei diversi valori assunti da mantissa ed esponente, varia anche il numero rappresentato. Anche il metodo getFloatSingleReal() ricava il valore del numero desiderato, sulla base dei diversi valori che mantissa ed esponente possono assumere. Per ulteriori approfondimenti a riguardo, si rimanda al testo citato alla voce [4t] della bibliografia.

Con l'attuazione delle modifiche descritte in questa sezione, la versione 0.9c del simulatore supporta quindi correttamente anche i dati di tipo TC_ULONG e TC_FLOAT.

# Capitolo 5

# Correzione bug del Simulatore

In questo capitolo verranno discussi i principali bug che sono stati rilevati nella versione 0.9b di *NXTSimulator*, ed il modo in cui essi sono stati risolti. Saranno omesse dalla trattazione discussioni riguardanti bug considerati di poco rilievo, ma che sono comunque stati corretti nel simulatore: ciò per evitare di rendere troppo prolisso codesto capitolo dell'elaborato. All'interno del progetto NetBeans di *NXTSimulator* 0.9c ogni correzione od implementazione di nuove funzioni è stata comunque corredata da adeguati commenti al codice sorgente, con tanto di nome dell'autore, così da rendere maggiormente chiare le scelte implementative effettuate.

## 5.1 Informazioni sulla Simulazione

La versione 0.9b del simulatore prevedeva che, mentre vi era la simulazione di un programma eseguibile in corso, tutti i passaggi che il software eseguiva, venissero accompagnati da una stampa a standard output che descrivesse ciò che veniva fatto. Man mano che veniva creata la Dataspace Table of Contents, venivano, ad esempio, visualizzati i campi di ogni record in essa contenuto, od ancora, quando si simulava una particolare istruzione del firmware, veniva stampato il nome della stessa, i parametri sui quali essa agiva, e l'eventuale risultato numerico prodotto.

Quanto appena descritto avveniva, però, in un modo alquanto caotico e lacunoso, dimostrandosi di fatto di poca utilità per chi ne volesse usufruire. Si è deciso quindi di ritoccare ed integrare, ove necessario, le istruzioni di stampa presenti all'interno delle classi *insClumpRecord.java*, *insDstoc.java*, *Execution.java* ed *Interpreter.java* del simulatore, le quali sono le responsabili di tutte le stampe di cui sopra,

rispettando le convenzioni per la localizzazione, precisate all'interno dell'elaborato citato al punto [5t] della bibliografia.

Qui sotto viene riportato un estratto delle stampe ordinate, esaurienti e facilmente comprensibili che *NXTSimulator* ora visualizza mentre esegue una simulazione.

```
Non ci sono clump dipendenti

clump #0: 0 0 0
clump #1: 1 0 312
clump #2: 1 0 342
clump #3: 1 0 351
clump #4: 1 0 367
clump #5: 1 0 408
clump #6: 1 0 443
clump #7: 1 0 563
clump #8: 1 0 615
clump #9: 1 0 675
clump #10: 1 0 720
clump #11: 1 0 738
clump #12: 1 0 746
clump #13: 1 0 772
clump #14: 1 0 789
clump #15: 1 0 852

0000000000101001 OP_SHORT_ACQUIRE NOT SUPPORTED
----------------------------
0000001101001000 OP_SHORT_MOV: 96 <= 100.0
----------------------------
0000001001001011 OP_SHORT_SUBCALL

Inizia il clump num: 3 Caller ID: 1
Code Start Offset: 351
Valore della variabile arg1: -1.0
----------------------------
0000000010000000 OP_ADD 96 <= 100.0 + 24.0
Effetto dell'operazione          TC_UByte       1       325        124
----------------------------
0000000010000000 OP_ADD 99 <= 24.0 + 1.0
Effetto dell'operazione          TC_UByte       1       328        25
----------------------------
0000010010000000 OP_DIV 100: 124.0, 25.0
Effetto dell'operazione          TC_UByte       1       329        4
```

**Figura 5.1:** *Informazioni stampate durante una simulazione*

## 5.2 Caricamento Configurazione delle Porte

Già dalla versione precedente all'attuale 0.9c il simulatore permetteva all'utente di caricare da file una particolare configurazione delle porte, esemplificate nei riquadri 2 e 3 della figura 4.1 a pagina 50. Accedendo alla voce del menù di *NXTSimulator*, rappresentato in alto nella figura 4.2 di pagina 51, denominata *File*, e selezionando la voce *Carica configurazione* l'utente poteva, infatti, cercare nel proprio calcolatore un file di configurazione in formato *.DAT*, da caricare nel simulatore e dal quale leggere la configurazione delle porte da applicare al medesimo. Tale meccanismo, tuttavia, non funzionava correttamente. Qualora, ad esempio, un utente impostasse dapprima un sensore (tra quelli disponibili) per ciascuna porta di ingresso, ed un motore per ciascuna porta di uscita del simulatore, e poi decidesse di caricare, mediante la funzionalità di cui sopra, una data configurazione, avente un solo motore ed un solo sensore, il risultato ottenuto non sarebbe stato corretto. Il simulatore, infatti, avrebbe soltanto sostituito, nella porta desiderata, il sensore già presente col nuovo sensore, qualora di tipo diverso l'uno dall'altro, ed avrebbe lasciato al proprio posto tutti gli altri sensori e servomotori già presenti. Il comportamento appena descritto è, però, da considerarsi errato, in quanto venivano sì allestiti nel simulatore i componenti della nuova configurazione, ma ciò veniva fatto ignorando l'eventuale presenza di altri sensori e/o motori precedentemente impostati, rendendo di fatto la configurazione non conforme a quella specificata nel file *.DAT* caricato dall'utente.

Per risolvere il bug di cui si è appena discusso, si sono dovute apportare delle semplici modifiche alla classe *Configuration.java* del simulatore. All'interno di quest'ultima è presente il metodo *setConfiguration()*, il cui compito è quello di caricare una determinata configurazione, specificata all'interno di un file *.DAT*, nel pannello principale di *NXTSimulator*. Si è deciso di anteporre a tutte le istruzioni del suddetto metodo una chiamata ad un altro metodo, implementato *ad hoc*, e denominato *cleanConfiguration()*. Quest'ultimo, il cui codice sorgente è riportato a pagina seguente, non fa altro che “pulire” il pannello del simulatore da tutti gli eventuali sensori e/o motori ivi presenti. Così facendo, la configurazione che viene poi caricata da setConfiguration() rispetta fedelmente quanto specificato nel file *.DAT*.

```
private void cleanConfiguration(){
    for(int i=1;i<=4;i++){ //Rimuove eventuali sensori di input presenti nella cfg.
        boolean tes=CRController.componentRemoverSensor(NXTSView.getPort(i),i);
    }
    for(int i=5;i<=7;i++){ //Rimuove eventuali servomotori presenti nella cfg.
        boolean tes = CRController.componentRemoverMotor(NXTSView.getPort(i),i);
    }
}
```

## 5.3 Bug Istruzioni Firmware

Come già detto nella sottosezione 3.1.1 di questo elaborato, il firmware del robot Lego® Mindstorms® NXT è in grado di interpretare ed eseguire le istruzioni presenti all'interno dei programmi, le quali costituiscono la componente principale dei medesimi. All'interno di *NXTSimulator*, il ruolo di interprete ed esecutore di queste istruzioni è svolto fondamentalmente, come già detto nel corso del capitolo precedente, dalla classe *Execution.java*.

Visti i molteplici malfunzionamenti che la versione 0.9b del simulatore presentava, riguardanti soprattutto il funzionamento dei servomotori, si è deciso di revisionare tutto il codice sorgente relativo alla suddetta classe, per verificarne la correttezza. La simulazione del movimento dei servomotori, infatti, come quella di tutti gli altri dispositivi del robot, non può prescindere dal supporto di numerose tra le istruzioni presenti nella classe Execution. Prima di procedere con la risoluzione del bug relativo ai motori del robot, si è rivelato quindi necessario procedere con quella di eventuali bug delle istruzioni del firmware 1.28.

Per portare a termine quanto appena detto, si sono analizzate ad una ad una tutte le istruzioni di cui sopra, presenti all'interno di Execution, verificando se esse erano state implementate rispettando le specifiche imposte dal firmware LEGO® [1s] e, per quelle che non le rispettavano, si è proceduto con le necessarie correzioni.

Le istruzioni che hanno avuto bisogno di essere ritoccate sono state molteplici, ed aventi errori di entità e tipologie differenti. Ecco una panoramica delle modifiche effettuate:

- **OP_DIV**: istruzione che effettua la divisione aritmetica tra due numeri. Essa necessita di tre parametri di ingresso, ossia dell'indirizzo in cui salvare il risultato della divisione, e di quello dei due operandi. E' stato corretto l'accesso che l'istruzione faceva al divisore dell'operazione: veniva, infatti,

utilizzato erroneamente come operando l'indice del record della DSTOC in cui reperire il medesimo, e non l'effettivo valore numerico del dato.

- **OP_MOD**: istruzione che effettua l'operazione di modulo tra due numeri (e.g: 7 % 3 = 1). Essa necessita di tre parametri di ingresso, ovvero dell'indirizzo in cui salvare il risultato dell'operazione, e di quello dei due operandi. Era stata omessa la verifica preventiva, che l'istruzione richiede, sul valore numerico del divisore: qualora, infatti, quest'ultimo sia uguale a '0', l'operazione di modulo non è effettuata, ed il risultato della medesima è il dividendo stesso.
- **OP_NOT**: istruzione che effettua l'operazione booleana di NOT su di un numero. Essa necessita di due parametri di ingresso, ossia dell'indirizzo in cui salvare il risultato dell'operazione, e di quello dell'operando su cui eseguirla. E' stato corretto il tipo di operazione che veniva effettuata sul dato in ingresso: era, infatti, erroneamente computato un NOT bit a bit, mediante l'operatore Java '~', e non il NOT booleano richiesto, il quale restituisce, invece, come risultato '1' se l'operando è uguale a '0', e '0' altrimenti.
- **OP_INDEX**: istruzione che copia un elemento, presente in una determinata cella di un array sorgente, in una determinata destinazione. Essa necessita di tre parametri di ingresso, ovvero dell'indirizzo di destinazione in cui salvare il dato da prelevare, di quello dell'array sorgente da cui prelevarlo, e dell'indice della cella di quest'ultimo in cui reperire il dato medesimo. In questo caso è stata ristrutturata nel complesso tutta l'istruzione, giacché essa non gestiva correttamente qualsiasi tipo di dato che poteva esserci nella casella dell'array sorgente di cui effettuare la copia: non veniva gestita adeguatamente, ad esempio, la situazione in cui la sorgente era un array di cluster, ciascuno contenente a sua volta altri cluster.
- **OP_REPLACE**: istruzione che rimpiazza un certo sottoinsieme di elementi di un'array sorgente con copie identiche di un certo dato, e salva l'array così ottenuto in un determinato array destinazione. Essa necessita in ingresso di quattro parametri, ossia dell'indirizzo dell'array di destinazione, di quello dell'array sorgente, dell'indice della cella di quest'ultimo da cui partire col rimpiazzo dei dati, e del dato da utilizzare come elemento di rimpiazzo. Anche in questo caso è stata ristrutturata nel complesso tutta l'istruzione: innanzitutto venivano assegnati dei valori errati ai parametri dell'istruzione,

ed era, inoltre, gestita in modo lacunoso la copia multipla del dato, di cui sopra, nell'array sorgente, e successivamente dell'array sorgente in quello di destinazione.

- **OP_ARRBUILD**: istruzione che costruisce un array, formato dalla concatenazione di un certo numero di elementi. Essa necessita in ingresso di un numero variabile di parametri, dipendente dal numero di elementi da inserire nell'array da costruire. Il primo parametro è la dimensione dell'istruzione medesima in numero di byte, il secondo è l'indirizzo di destinazione in cui salvare l'array che si ottiene, ed i rimanenti sono i dati da concatenare per ottenere l'array desiderato. In questo caso era stata omessa la gestione del caso in cui i dati da concatenare fossero dei cluster contenenti al loro interno altri cluster.
- **OP_ARRSUBSET**: istruzione che salva in un array destinazione un certo numero di elementi di un array sorgente, partendo da una data cella di quest'ultimo. Essa necessita in ingresso di quattro parametri, ovvero dell'indirizzo dell'array destinazione, di quello dell'array sorgente, dell'indice della cella di quest'ultimo da cui iniziare a prelevare i dati, e del numero di essi da prelevare. Qui era presente un banale errore di indicizzazione, presente nel contesto di copia di eventuali cluster dall'array sorgente a quello destinazione.
- **OP_ARRINIT**: istruzione che inizializza un certo array destinazione con un determinato numero di copie di un certo dato. Essa necessita in ingresso di tre parametri, ossia dell'indirizzo dell'array destinazione, del dato con cui riempirlo, e del numero di copie di quest'ultimo da inserire nell'array medesimo. In questo caso, qualora il dato da copiare fosse stato un array, era stata omessa la distinzione tra array di dati numerici ed array di cluster, i quali vanno gestiti in modo differente. Nel caso in cui, invece, il dato in questione fosse stato un cluster, non era stata considerata la possibilità che all'interno di esso vi fossero altri cluster.
- **OP_MOV (codifica *Short*)**: istruzione che copia un determinato dato in una determinata destinazione. Essa necessita in ingresso di due parametri, ovvero dell'indirizzo della destinazione in cui salvare il dato da copiare, e di quello della sorgente in cui reperirlo. In questo caso non veniva gestita correttamente la situazione in cui la sorgente era un cluster, contenente a sua volta altri

cluster.

- **OP_MOV (codifica *Long*)**: istruzione assolutamente analoga alla precedente. Sono state apportate le medesime modifiche descritte per l'omonima istruzione con codifica *short* appena presentata ed, inoltre, è stato corretto l'utilizzo che veniva fatto dei parametri di ingresso: in alcune circostanze essi erano stati, infatti, scambiati tra di loro, dando vita ad un comportamento ovviamente anomalo dell'istruzione.
- **OP_NUMTOSTRING**: istruzione che converte un numero in una stringa testuale. Essa necessita di due parametri di ingresso, ossia dell'indirizzo della destinazione in cui salvare salvare la stringa ottenuta, e di quello del numero da convertire. In questo caso si è modificata l'istruzione, permettendole di gestire correttamente anche numeri da convertire di tipo TC_FLOAT o TC_ULONG, nonché di supportare due diverse formattazioni della stringa ottenuta, rispettivamente nel caso in cui essa sia più grande del display del robot (simulatore) o viceversa.
- **OP_STOP**: istruzione che termina l'esecuzione del programma in corso, qualora il valore numerico dell'unico parametro che essa riceve in ingresso sia diverso da zero. In questo caso era stata innanzitutto omessa una verifica preventiva sul valore del parametro di cui sopra: qualora, infatti, l'indice della DSTOC (fornito in ingresso), al quale reperire il dato, sia pari a '65535' (*NOT_A_DS_ID* [1s]), va assegnato al parametro medesimo un qualsiasi valore diverso da zero; in caso contrario, invece, si va ad estrapolare il valore effettivo del dato, mediante l'apposito record della DSTOC puntato dall'indice ricevuto in ingresso. Nella versione 0.9b del simulatore, oltre ad essere stata omessa la verifica appena discussa, veniva anche utilizzato erroneamente come operando l'indice della DSTOC in cui reperire il parametro, e non il suo effettivo valore numerico. Veniva, inoltre, commesso un altro grave errore: si forzava, infatti, l'arresto del solo clump che aveva eseguito questa istruzione, e non anche quello di eventuali altri clump in esecuzione in quel momento, dando vita quindi ad un comportamento anomalo della simulazione.
- **OP_FINCLUMP**: istruzione che termina l'esecuzione del clump che l'ha lanciata. Essa riceve in ingresso due valori numerici interi, da utilizzare come indici nella lista dei clump che dipendono da quello di cui si è imposta la terminazione. Qualora questi indici siano entrambi interi positivi, tutti i

clump presenti a partire dalla posizione nella suddetta lista, puntata dal primo dei due indici, e fino a quello puntato dal secondo dei due, se pronti per essere eseguiti (ossia tutti i clump da cui essi dipendono sono terminati) vengono mandati in esecuzione. Qualora gli indici, invece, siano negativi, essi sono ignorati e non viene schedulato nessun altro clump. In questo caso l'istruzione era stata implementata in modo lacunoso, in quanto non venivano schedulati tutti i clump pronti eventualmente per l'esecuzione dopo la terminazione del clump che l'aveva lanciata, ma veniva eseguito soltanto il primo di essi, cagionando quindi un comportamento anomalo del simulatore. Per ulteriori approfondimenti sulle correzioni effettuate a questa istruzione, consultare la sezione 5.5 di questo elaborato.

## 5.4 Bug Servomotori

L'aggiornamento di *NXTSimulator* alla versione 0.9c ha previsto un notevole sforzo per risolvere i problemi relativi alla simulazione dei servomotori del robot LEGO®. Nella versione 0.9b del simulatore quest'ultima presentava, infatti, delle gravi anomalie e non emulava fedelmente il comportamento del robot. Una simulazione che prevedesse anche il movimento di uno o più servomotori offriva dei risultati errati e densi di errori, i più rilevanti dei quali erano i seguenti:

- la simulazione spesso entrava in un *loop* infinito, il quale poteva essere interrotto soltanto arrestando manualmente la medesima;
- durante una simulazione avveniva di frequente il lancio di un'eccezione Java del tipo *java.lang.ArithmeticException* (divide by zero) [16s], lanciata da una classe responsabile della simulazione dei motori;
- qualora si desse in pasto al simulatore un programma che, ad esempio, imponesse ad un servomotore di percorrere un certo numero di gradi, quest'ultimo non si arrestava nel punto esatto specificato dal programma medesimo, ma percorreva una quantità di gradi a volte minore ed a volte maggiore del dovuto, dimostrando quindi un comportamento errato, oltre che non deterministico;
- il movimento in senso antiorario di un motore presentava un comportamento a dir poco bizzarro, in quanto l'immagine rotante del simulatore (esemplificata, in arancio, nelle immagini di figura 5.4 a pagina 130), che

rappresenta il movimento del motore in questione, oscillava in maniera del tutto incontrollata in senso alternativamente orario ed antiorario;

- il movimento sincronizzato di due servomotori, previsto dal robot LEGO®, non era supportato correttamente nel simulatore: i motori da sincronizzare procedevano, infatti, in modo asincrono;
- il fattore di sterzata previsto per il movimento di due motori era stato implementato in modo scorretto, utilizzando, per realizzarlo nel simulatore, una legge empirica non conforme alle specifiche Lego® Mindstorms® NXT.

Per risolvere le anomalie appena discusse, e colmare anche altre lacune che la simulazione dei servomotori presentava, si sono modificate le classi del progetto *MotorData.java*, *OutputPortConfigurationProperties.java* e *MotorPanel.java*.

La classe MotorData definisce tutte le proprietà di un servomotore, secondo le specifiche del firmware LEGO® [1s].

La classe OutputPortConfigurationProperties mette a disposizione essenzialmente il metodo *motor()*, il quale viene utilizzato per impostare all'occorrenza i valori di certe proprietà dei servomotori, sulla base di quanto specificato nel programma da simulare.

La classe MotorPanel si occupa di far vedere all'utente il movimento vero e proprio dei servomotori.

Per ciascun servomotore da simulare è definita un'istanza di classe MotorData ed una di classe MotorPanel: quest'ultima aggiorna a brevi intervalli di tempo (qualche millisecondo) la posizione angolare dell'immagine che rappresenta la rotazione del motore corrispondente, sulla base dei valori delle proprietà specificate in MotorData, ed impostate di volta in volta da OutputPortConfigurationProperties. Per ulteriori dettagli riguardanti il *modus operandi* appena descritto, consultare l'elaborato citato al punto [6t] della bibliografia.

### 5.4.1 MotorData.java

Già nella versione 0.9b del simulatore questa classe era implementata in modo sostanzialmente corretto, eccezion fatta per il modo in cui era stata definita una proprietà dei motori, ossia quella denominata *TACH_LIMIT*. Quest'ultima, secondo le

specifiche LEGO® [1s], può assumere valori interi nell'intervallo [0 ; 4294967295]; nella classe MotorData essa era stata implementata con il tipo di dati Java *int*, il quale permette, però, di definire numeri interi soltanto nell'intervallo [-2147483648 ; 2147483647], dando vita quindi a potenziali errori di *overflow* [1t] e di conseguenza ad anomalie nella simulazione, giacché non permetteva di coprire tutti i valori assumibili dalla suddetta proprietà.

Per risolvere il bug sopra citato, si è semplicemente ridefinita la variabile di esemplare relativa a TACH_LIMIT, convertendola al tipo Java *long*, il quale copre l'intervallo di valori che segue, più che adatto, come si vede, in questo contesto:

[-9223372036854775808 ; 9223372036854775807].

### 5.4.2 OutputPortConfigurationProperties.java

Come già accennato nel corso di questa sezione, la classe OutputPortConfigurationProperties mette a disposizione un metodo chiamato *motor()*, utile per impostare all'occorrenza le proprietà dei servomotori, sulla base delle istruzioni presenti nel programma *.RXE* che si sta simulando. Nella versione 0.9b di *NXTSimulator* il metodo in questione possedeva delle righe di codice che permettevano di impostare anche certe proprietà dei servomotori che il firmware LEGO® considera, però, di sola lettura, come ad esempio *ACTUAL_SPEED* e *TACH_COUNT* [1s]. La presenza delle suddette istruzioni non comportava di fatto alcun problema funzionale, giacché nessun programma eseguibile, che rispettasse le specifiche del firmware, le andava ad eseguire, ma costituiva un errore propriamente "concettuale": esse sono state quindi rimosse dal codice del metodo motor().

All'interno del suddetto metodo si è reso anche necessario l'inserimento di un segmento di codice di controllo atto a ritardare, in certe circostanze, l'aggiornamento delle proprietà dei motori di qualche millisecondo. Studiando il funzionamento del firmware LEGO® si è, infatti, appreso che esso opera nel modo che verrà ora descritto, qualora si verifichi la circostanza seguente. Nel caso in cui un programma ordini a due servomotori di percorrere sincronizzati un certo numero di gradi e poi di arrestarsi, il firmware fa innanzitutto partire i due motori, per poi interrogare periodicamente lo stato di soltanto uno di essi, per apprendere quando abbia terminato la propria percorrenza. Quando viene riscontrato tale fatto, vengono posti entrambi i motori ad *idle* ("disoccupati"), mediante l'impostazione di alcune loro proprietà. Ciò avviene poiché essi procedono sincronizzati e quindi la terminazione

del lavoro di uno coincide anche con quella dell'altro. Anche il simulatore attua il *modus operandi* appena descritto, ma nella versione 0.9b era presente un'anomalia: durante l'interrogazione periodica, di cui sopra, dello stato di uno dei due motori, poteva, infatti, accadere che l'istanza di MotorPanel ad esso relativa terminasse di simulare qualche millisecondo prima di quella dell'altro la percorrenza desiderata (per questioni di performance del calcolatore su cui veniva eseguito il simulatore), ed imponesse quindi l'arresto di entrambi. Così facendo, però, accadeva che, al termine della simulazione, un motore si trovasse nella posizione finale desiderata, mentre l'altro no. L'inserimento del codice sorgente di cui sopra impone, invece, al motore posto sotto controllo di attendere eventualmente che anche l'altro termini il suo compito, prima di forzare il salto di entrambi alla modalità *idle*, risolvendo così il problema in questione e favorendo di fatto il sincronismo dei servomotori. Notare che, nel caso in cui il motore non sottoposto ad "interrogazione" termini per primo, non si verificano, invece, problemi, giacché esso passa da sé allo stato *idle*, senza imporlo anche all'altro servomotore.

**Fattore di Sterzata**

Il robot LEGO® non può solamente avanzare seguendo sempre una linea retta, ma può anche curvare o addirittura girare su sé stesso. Per far curvare il robot, è necessario allestire una configurazione del medesimo con due motori sincronizzati, simile a quella di figura 5.3, ed assegnare una potenza distinta a ciascuno di essi. Assegnando, ad esempio, una maggiore potenza ad un motore rispetto all'altro, il robot avanza curvando verso il motore al quale è stata assegnata la potenza minore. Per ottenere il risultato appena descritto, è necessario specificare per i motori in questione un cosiddetto *fattore di sterzata*, sulla base del valore del quale viene poi applicata ai motori in questione la potenza adatta per farli procedere nel modo desiderato. All'interno dei programmi *.RXE* per il robot, il fattore di sterzata è specificabile impostando il valore della proprietà chiamata *TURN_RATIO*, la quale può assumere valori interi nell'intervallo [-100 ; 100]. Il valore di tale proprietà comporta una modifica della potenza da applicare ai motori in questione (proprietà *SPEED*), secondo la legge rappresentata nella figura a pagina seguente [21s].

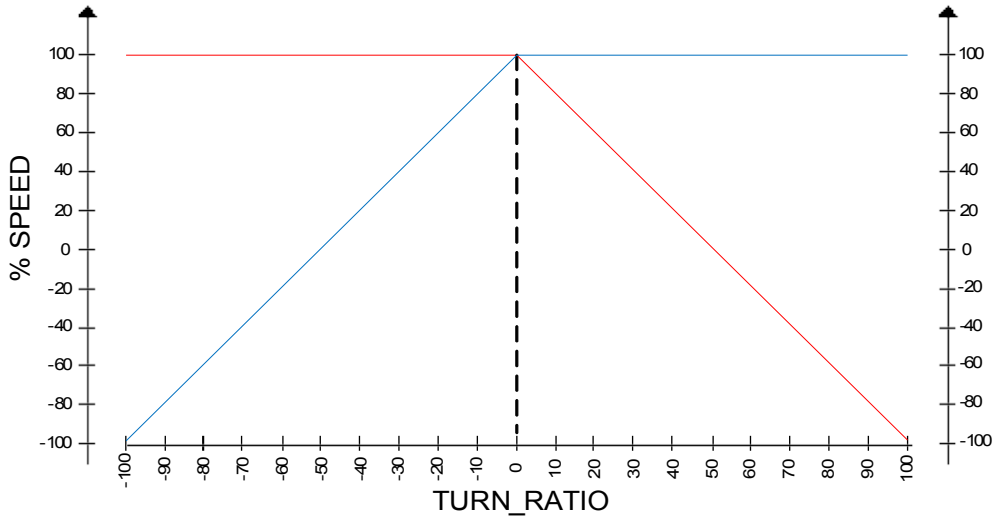


**Figura 5.2:** *Legge fattore di sterzata servomotori*

Come si evince dall'analisi della figura riportata qui sopra, si possono scorgere essenzialmente due comportamenti distinti dei motori, in funzione del valore di TURN_RATIO, e nella fattispecie i seguenti:

- qualora il fattore di sterzata (TURN_RATIO) sia pari a '0', entrambi i motori in questione ricevono dal robot il 100% della potenza specificata per essi dalla proprietà *SPEED* e non vi è alcuna sterzata da parte del robot medesimo;
- qualora, invece, il fattore di sterzata sia diverso da '0', ci si può trovare in uno dei casi seguenti:
  - se esso è maggiore di zero, al motore posizionato alla sinistra, guardando frontalmente il display del robot, è assegnato il 100% del valore specificato da SPEED, mentre a quello posizionato alla destra viene assegnata la percentuale di potenza data dalla seguente:

    ***%SPEED = 100 - 2*TURN_RATIO***

  - se esso, invece, è minore di zero, al motore posizionato alla destra, guardando frontalmente il display del robot, è assegnato il 100% del valore specificato da SPEED, mentre a quello posizionato alla sinistra viene assegnata la percentuale di potenza data dalla seguente:

    ***%SPEED = 100 + 2*TURN_RATIO***

  Nei due sotto-casi appena descritti si può quindi osservare che un valore assoluto di TURN_RATIO pari a '50' provoca il funzionamento di un solo motore, mentre un valore assoluto pari a '100' provoca la rotazione dei motori

in direzioni opposte, ma ad uguale potenza. Notare come, con un valore di TURN_RATIO pari ad uno dei due appena citati, ed una configurazione del robot simile a quella della figura sottostante, quest'ultimo ruoti su sé stesso.


**Figura 5.3:** *Possibile configurazione del robot LEGO®*

Il fattore di sterzata implementato nel modo appena descritto simula così correttamente quello che è il comportamento dei motori sottoposti a "sterzo", secondo le specifiche LEGO®, cosa che, invece, non avveniva con l'implementazione del medesimo che era sopravvissuta fino alla versione 0.9b del simulatore, e documentata nell'elaborato riportato al punto [7t] della bibliografia.

### 5.4.3 MotorPanel.java

Questa classe si è rivelata essere la responsabile della maggior parte dei bug relativi alla simulazione dei servomotori da parte di *NXTSimulator*. Verrà ora proposta una panoramica delle modifiche correttive più rilevanti che sono state ad essa apportate.

Innanzitutto è stata ritoccata l'interfaccia grafica, utile per la visualizzazione della simulazione dei singoli servomotori, e definita dalla classe MotorPanel. Nella figura a pagina seguente sono riportate rispettivamente la versione antecedente e quella conseguente al ritocco.

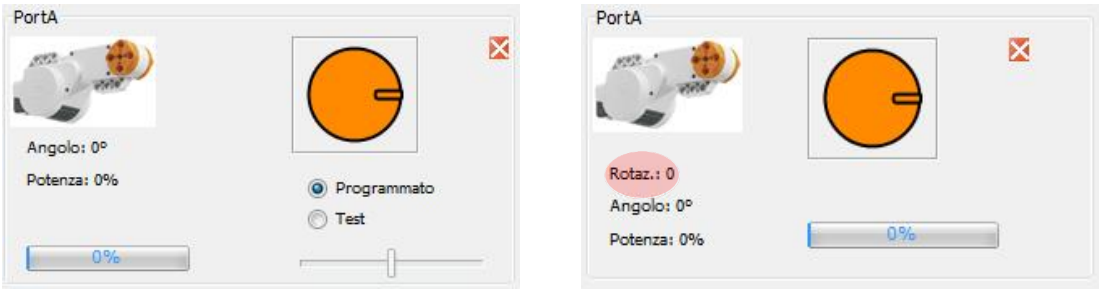


**Figura 5.4:** *Vecchia e nuova interfaccia grafica dei servomotori nel simulatore*

Come emerge dal confronto tra le due immagini allegate qui sopra, si nota che nella nuova versione della grafica dei servomotori sono stati rimossi i pulsanti tramite i quali era possibile scegliere se far girare il motore in questione mediante le specifiche istruzioni del programma *.RXE* da simulare (*Programmato*), o se incaricare l'utente di impostare la potenza da applicare al medesimo (*Test*), specificandola mediante l'apposita barra scorrevole rappresenta in figura. Tale distinzione, infatti, si è rivelata a lungo andare inutile, e si è quindi propenso per far funzionare i motori solamente con la prima delle due modalità appena descritte. Si è, inoltre, aggiunto un oggetto di tipo *javax.swing.JLabel* [16s], denominato *rotationLab* e cerchiato in rosso in figura 5.4, utilizzato per visualizzare durante una simulazione il numero di giri completi (rotazioni di 360°) effettuati da un motore.

Oltre ad aver ritoccato l'aspetto grafico della classe MotorPanel, si è anche rivisitata e modificata la sua porzione di codice sorgente responsabile dell'aggiornamento periodico della grafica medesima, attuato per simulare il movimento del servomotore. Il codice in questione è quello contenuto all'interno del costruttore della classe, il quale essenzialmente istanzia un oggetto di classe *javax.swing.Timer* [16s], il quale funge da timer temporale, che si occupa di invocare di continuo l'aggiornamento della posizione angolare del servomotore, sulla base delle proprietà specificate in ingresso per esso. Durante una simulazione l'aggiornamento della grafica relativa ad un motore avveniva, nella versione 0.9b di *NXTSimulator*, attuando una distinzione sulla base della modalità operativa in cui si trovava di volta in volta ad operare il servomotore. Quest'ultimo poteva trovarsi in una tra cinque distinte modalità: ciò che esso faceva in ognuna di esse era tuttavia implementato in modo errato o lacunoso. Si è dunque dovuto procedere di conseguenza.

Nella versione 0.9c del simulatore il *modus operandi* adottato per simulare un motore ha previsto la conservazione dell'approccio di cui sopra, ritoccando, però, ove necessario, l'implementazione delle singole modalità operative.

Le cinque modalità in cui un servomotore può agire sono le seguenti: *running*, *rampdown*, *rampup*, *idle* e *coast*. Verrà ora descritto per ciascuna di queste, il modo in cui essa è stata ritoccata nella versione 0.9c del simulatore.

La modalità *running* prevede semplicemente che un servomotore percorra un certo numero di gradi ruotando con una determinata potenza, specificata dalla proprietà *SPEED* dell'istanza di MotorData ad esso relativa. Questa modalità era stata già implementata in modo sostanzialmente corretto, eccezion fatta per il modo in cui era aggiornata di volta in volta la posizione angolare del motore, nel caso di rotazione in senso antiorario: era, infatti, adottata una formula errata per il calcolo della medesima, e ciò comportava il comportamento bizzarro della simulazione, di cui si è parlato all'inizio di questa sezione. L'errore appena discusso era presente, ed è stato perciò corretto, anche nelle modalità rampdown e rampup.

La modalità *rampdown* obbliga un motore a percorrere un certo numero di gradi, diminuendo gradualmente la propria potenza iniziale, fino ad azzerarla. Questa modalità era stata implementata in modo completamente scorretto. La formula utilizzata per il decremento graduale della potenza era, infatti, errata e provocava il lancio dell'ArithmeticException, di cui si è parlato al principio di questa sezione, causata da una divisione di un numero per zero, che in certe circostanze veniva effettuata.

Per far ridurre progressivamente la potenza di un motore si è scelto di utilizzare la legge fisica del *moto circolare uniformemente accelerato* [8t], secondo la quale la velocità angolare $\omega$ di un corpo in un certo istante di tempo $t$ è data dalla seguente:

$$\omega(t) = \omega_0 + \alpha(t - t_0)$$

dove $\omega_0$ è la velocità angolare iniziale del corpo, $\alpha$ è la sua accelerazione angolare, e $t_0$ è l'istante di tempo iniziale; la posizione angolare $\theta$ del corpo all'istante di tempo $t$ è data, invece, dalla seguente:

$$\theta(t) = \theta_0 + \omega_0(t - t_0) + 1/2*\alpha(t - t_0)^2$$

dove $\theta_0$ è la posizione angolare iniziale del corpo.

Dal momento che nel moto circolare uniformemente accelerato l'accelerazione angolare è costante, ossia in ogni istante essa assume il medesimo valore, per calcolare quest'ultimo è sufficiente fissare un certo istante di tempo ed andare a ricavarne il valore in quel preciso istante: vediamo ora come è stato applicato quanto detto nel nostro caso. Impostando come tempo iniziale l'istante $t_0$ pari a '0', e

considerando le due formule precedenti all'istante di tempo $\bar{t}$, in cui sono stati percorsi i gradi desiderati e si è raggiunta la velocità obiettivo, si ha che $\omega(\bar{t})$ è uguale alla velocità obiettivo, che chiamiamo per comodità $\omega_F$, mentre $\theta(\bar{t})$ è uguale alla posizione angolare finale del motore, data dalla somma tra la sua posizione angolare iniziale e i gradi percorsi, e quindi pari alla somma di $\theta_0$ con quest'ultimi, i quali chiamiamo per comodità $\theta_P$. Andando ora a sostituire questi valori nelle formule soprastanti, otteniamo le seguenti:

$$\omega_F = \omega_0 + \alpha * \bar{t}$$

$$\theta_0 + \theta_P = \theta_0 + \omega_0 * \bar{t} + 1/2 * \alpha * \bar{t}^2$$

dalle quali, mediante semplice manipolazione algebrica, si ottiene la formula per determinare il valore dell'accelerazione angolare in gradi al millisecondo quadrato:

$$\alpha = (\omega_F^2 - \omega_0^2)/(2 * \theta_P)$$

Il valore costante dell'accelerazione angolare così ottenuto si può quindi applicare alla formula standard della velocità angolare riportata a pagina precedente, per determinare ad ogni istante di tempo il valore della medesima.

Giacché nel nostro contesto non si opera propriamente riducendo di volta in volta il valore della velocità angolare di un servomotore, ma bensì quello della potenza ad esso applicata, è stato necessario utilizzare anche la formula seguente, al fine di convertire il valore della velocità angolare effettiva del motore nel corrispondente valore di potenza da applicare ad esso, per farlo ruotare a quella velocità:

$$Potenza = (100 * \omega(t))/0.9612$$

dove *0.9612* esprime la velocità angolare (in gradi al millisecondo) alla quale ruota un servomotore al quale è fornita massima potenza, ossia potenza pari a 100 [6t].

Con l'implementazione appena descritta la modalità rampdown funziona ora correttamente, permettendo la graduale diminuzione della potenza di un motore, fino a farlo arrestare, e non provocando più l'eccezione di cui era, invece, responsabile nella versione 0.9b del simulatore.

La modalità *rampup* impone ad un servomotore di percorrere un certo numero di gradi, accrescendo gradualmente la propria potenza iniziale, fino a toccare una determinata potenza obiettivo, specificata dalla proprietà *SPEED* dell'istanza di MotorData ad esso relativa. Questa modalità presentava gli stessi problemi della precedente, ed è stata ristrutturata in modo assolutamente analogo a quest'ultima.

L'unica differenza tra queste due modalità sta nel fatto che la rampdown provoca una diminuzione progressiva della potenza di un motore, mentre la rampup ne suscita un aumento graduale.

La modalità *idle* disabilita semplicemente qualsiasi entità di potenza per un motore, rendendolo di fatto "disoccupato". La modalità *coast* prevede, invece, che il motore ruoti liberamente, senza alcun freno o potenza applicata. Queste due modalità sembrano, a prima vista, causare il medesimo comportamento di un servomotore, ma in realtà non è così. Studiando attentamente il comportamento del firmware LEGO® [1s], si è, infatti, appreso che un motore entra in modalità idle quando è già fermo e quindi non ha alcuna potenza ad esso applicata. Quando, invece, un servomotore è in modalità coast, significa che esso è ancora in movimento e che è lasciato ruotare liberamente, un po' come accade quando si mette in folle un'automobile lanciata in corsa. Nella versione precedente del simulatore l'implementazione di queste due modalità era stata di fatto "invertita", facendo fare alla modalità idle ciò che, invece, avrebbe dovuto fare la coast, e viceversa. Si è quindi dovuto provvedere a risolvere tale malfunzionamento, ed a raffinare qualche semplice dettaglio implementativo ad esse relativo. La modalità idle non svolge quindi alcun tipo di operazione sul servomotore in questione, mentre la coast diminuisce di una unità la potenza residua del motore, ad ogni intervento del timer per aggiornare la grafica del motore nel simulatore, fino a che il motore medesimo non si arresta.

Grazie agli aggiornamenti correttivi effettuati sulle classi MotorData, OutputPortConfigurationProperties e MotorPanel si sono risolti tutti i bug relativi alla simulazione dei servomotori, descritti all'inizio di questa sezione, ottenendo così una versione finalmente corretta e funzionante degli stessi all'interno di *NXTSimulator.*

## 5.5 Ristrutturazione schedulazione *clump*

Come si è discusso nella sottosezione 3.2.3 di questo elaborato, un programma eseguibile dal robot LEGO® e dal simulatore è costituito da uno o più clump (blocchi di istruzioni correlate), la cui schedulazione a *run-time* avviene sulla base delle informazioni contenute all'interno dei clump record definiti per ciascuno di essi. La corretta schedulazione dei clump nel corso dell'esecuzione di un programma è quindi una componente fondamentale, affinché un programma svolga regolarmente il

compito per il quale è stato realizzato.

Nella versione 0.9b di *NXTSimulator* l'algoritmo mediante il quale venivano schedulati i clump nel corso di un programma era errato e ciò, in molti casi, provocava un andamento anomalo della simulazione. Il lancio dei primi clump di un programma avveniva nel modo seguente. All'inizio di una simulazione si consultavano, ad uno ad uno, i campi *Fire Count* di tutti i clump da cui era composto il programma: questo campo specifica quando un clump è pronto per l'esecuzione. Qualora fire count fosse stato pari a '0' (clump pronto per essere eseguito), il clump veniva mandato in esecuzione, istanziando e lanciando un thread di classe *Execution.java*, operante poi nel modo descritto nella sezione 4.5 di questa tesi; in caso contrario, invece, il thread non veniva nemmeno creato. Così facendo, tutti i clump, che all'inizio di un programma erano già pronti per essere eseguiti, venivano mandati in esecuzione. Fino a qui, la strategia adottata è da ritenersi corretta. Il problema, però, sorgeva quando, ad un certo punto, durante l'esecuzione di un programma, un certo clump terminava, ed era prevista la successiva invocazione di uno o più clump da esso dipendenti (mediante l'istruzione *OP_FINCLUMP*, discussa nella sezione 5.3). In questo caso si andava a consultare la lista dei clump dipendenti dal clump appena concluso, e veniva mandato in esecuzione soltanto il primo di essi, verificando preventivamente se fossero terminati o meno tutti gli altri eventuali clump (oltre a quello appena concluso) da cui esso dipendeva. Nel caso in cui la lista dei clump dipendenti da quello terminato dall'istruzione OP_FINCLUMP fosse stata composta da un solo elemento, la schedulazione, seppur con un procedimento errato, procedeva comunque correttamente; nel caso in cui, invece, tale lista fosse stata composta da più elementi, la schedulazione avveniva erroneamente. Oltre all'errore appena descritto, anche il modo in cui avveniva la verifica sui clump pronti per essere mandati in esecuzione era scorretto: essa veniva, infatti, effettuata sfruttando una serie di variabili gestite all'interno della classe Execution, e non, come impone il firmware Lego® NXT, mediante il campo fire count dei clump record. Accadeva, perciò, che, dopo l'esecuzione dell'istruzione OP_FINCLUMP, andasse eventualmente in esecuzione soltanto uno tra i clump che dipendevano da quello appena terminato, e che la verifica, se esso fosse effettivamente pronto o meno per essere eseguito, fosse operata in modo scorretto. Il firmware prevede, infatti, che tale verifica venga effettuata consultando il campo fire count di ciascun clump in esame e di mandarlo in esecuzione soltanto nel caso in cui esso sia pari a '0'. E' bene far notare che il campo fire count non è altro che un contatore del numero di clump da

cui un clump dipende: appena un clump termina di eseguire, va quindi decrementato di una unità il fire count di tutti i clump da esso dipendenti, e vanno mandati in esecuzione quelli che, con questa operazione, hanno eventualmente azzerato il proprio fire count.

Per implementare all'interno del simulatore la schedulazione dei clump di un programma nel modo imposto dal firmware Lego® NXT, si è modificata l'istruzione OP_FINCLUMP, nel modo presentato nel corso della sezione 5.3, e si è introdotto l'utilizzo in modo adeguato del campo fire count dei clump record, per stabilire la corretta schedulazione *run-time* dei clump.

Un altro grave errore, rilevato nell'esecuzione dei clump di un programma, è stato quello relativo alla gestione dei dati statici e dinamici, cui i clump accedono in lettura e/o scrittura durante la simulazione. Ogni thread di classe Execution che veniva creato all'inizio di una simulazione disponeva, infatti, di una propria copia dei dati statici e dinamici, inficiando così, di fatto, la possibilità da parte dei clump di operare su dati condivisi, e dando vita quindi ad un comportamento errato della simulazione. Il problema appena presentato è stato risolto creando un'unica istanza dei suddetti dati, accessibile da tutti i thread di classe Execution.

Con gli interventi apportati, e descritti in questa sezione, la schedulazione dei clump e la gestione dei dati all'interno dei programmi eseguiti al simulatore avviene quindi ora in modo corretto, ricalcando esattamente quello che è il *modus operandi* del firmware Lego® NXT.

# Capitolo 6

# Manuale Utente

## Introduzione

Sebbene sia ragionevolmente possibile realizzare un simulatore completo per il robot Lego® Mindstorms® NXT, ossia un sistema grafico 2D/3D capace di rappresentare uno scenario completo con il robot medesimo, il piano in cui esso si muove, oggetti quali ostacoli, muri, piste, etc., ed un interprete capace di eseguire un programma simulando con precisione i movimenti coerentemente a quanto specificato nel programma, la sua realizzazione è un lavoro alquanto complesso, ed è difficile rappresentare tutte le incertezze che un sistema reale può "vantare". Inoltre, sotto il punto di vista educativo, il processo di sviluppo di un progetto del tipo *try-error-correct* è il modo più produttivo per favorire nuove conoscenze, e perciò l'interazione con il robot reale e le variabili incognite di uno scenario reale è sempre fondamentale. Ciò nonostante la disponibilità di un *tool* che permetta all'utente di effettuare un *debug* preventivo di un programma per il robot, al fine di correggere gli errori più grossolani ed evidenti, può essere vantaggiosa. *NXTSimulator* risponde perfettamente a questo requisito. Esso include un'interfaccia grafica 2D in grado di riprodurre il comportamento dei dispositivi di base del robot LEGO®: motori, sensori standard, display, bottoni del brick e riproduttore di suoni. L'interprete integrato nel simulatore è compatibile con la versione ufficiale 1.28 del firmware NXT: ciò significa che esso è in grado di eseguire un file eseguibile in formato *.RXE*, contenente il bytecode generato mediante l'ambiente di sviluppo NXT-G o il compilatore NXC. In seguito sarà spiegato come salvare file nel suddetto formato, dopo una compilazione tramite NXT-G.

L'interazione con l'ambiente esterno è simulata inoltrando dei valori appropriati ai sensori “collegati” al robot simulato, e che quest'ultimi poi restituiscono, come se li avessero effettivamente estrapolati dall'ambiente. Durante una simulazione il valore ritornato da un sensore è fornito in ingresso dall'utente, il quale può agevolmente impostarlo mediante la semplice interfaccia grafica dei simulatore. In alternativa la variazione nel tempo del valore di un sensore può essere fornita in termini di una funzione continua, la quale è data da un'interpolazione di campioni riportati dall'utente in un apposito file.

## Preparazione del file eseguibile

Il formato (*.RXE*) del file eseguibile dal simulatore è quello del bytecode prodotto dalla compilazione di un file sorgente NXT-G o NXC. Mentre, però, la compilazione di un file NXC di solito produce un file in tale formato in un altro file separato, la versione standard dell'ambiente NXT-G, invece, non lo fa. E' necessario quindi aggiungere un file (*DownloadToFile.llb*, una libreria LabVIEW™) alla distribuzione standard del programma NXT-G, nella fattispecie nella cartella *<installing dir>\LEGO Software\LEGO MINDSTORMS Edu NXT\engine\project*. Fatto ciò, l'interfaccia grafica di NXT-G include una nuova voce di menù (*Tools → Download to File...*), la quale permette il salvataggio del file nel formato *.RXE* desiderato.

## Il pacchetto *NXTSimulator*

Il pacchetto di distribuzione del simulatore consiste in un unico archivio *zip*. Scompattando tale archivio in una cartella si ottengono i seguenti elementi:

```
02/12/2010  17.05    <DIR>          output
02/12/2010  17.05    <DIR>          lib
02/12/2010  17.05    <DIR>          src
12/09/2010  15.05                43 NXTSimulator.bat
02/12/2010  10.15         3.884.088 NXTSimulator.exe
01/12/2010  19.25         3.857.976 NXTSimulator.jar
29/04/2008  16.07           481.407 DownloadToFile.llb
13/09/2010  10.18                 0 NXTSimulator.out
30/11/2010  19.08                25 pref.properties
27/04/2010  22.55                39 NXTSimulator.sh
```

Il file *jar* contiene tutte le classi Java del simulatore. Nella cartella *lib* ci sono delle librerie necessarie per il funzionamento del software. Nella cartella *output* vengono salvati dei file di testo diagnostici, che vengono prodotti durante ogni simulazione, e

sono presenti le cartelle *FileTXT* e *LogMotor*, nelle quali vengono salvati rispettivamente eventuali file di testo scritti mediante la funzione di scrittura file del robot (simulatore) ed eventuali file di log del movimento dei motori che l'utente può scegliere di produrre durante una simulazione che coinvolga anche i servomotori. La cartella *src* include le risorse, quali immagini e file audio, necessarie al funzionamento del software. Il file *exe* permette di avviare il simulatore in ambiente Windows, mentre il file *bat* fa altrettanto, ma permette anche di osservare delle informazioni sulla simulazione in una finestra di comando. Il file *out* permette di avviare il simulatore in ambiente UNIX/Linux, mentre il il file *sh* svolge il medesimo ruolo di quest'ultimo, ma permette anche di visualizzare informazioni sulla simulazione nella Shell. Il file *pref.properties* contiene delle impostazioni di configurazione del software, tra le quali si annovera la lingua nella quale eseguirlo (attualmente sono disponibili Italiano ed Inglese). La libreria *DownloadToFile.llb*, di proprietà di LabVIEW™, è necessaria per permettere, come detto in precedenza, di salvare il bytecode eseguibile prodotto mediante NXT-G in un apposito file. Per ottenere questa funzionalità bisogna copiare la libreria nella cartella presente al percorso *..\LEGO Software\LEGO MINDSTORMS Edu NXT\engine\project* della distribuzione del software NXT-G.

## Esecuzione del simulatore

E' importante far notare che l'interfaccia grafica di *NXTSimulator* è impostata in modo tale da funzionare correttamente con una risoluzione del monitor pari a 1024x768 o superiore; risoluzioni più basse danno vita ad una visualizzazione errata del layout del software. Appena si avvia il simulatore, appare una schermata identica a quella rappresentata nella figura 6.1 riportata a pagina seguente. Come si può notare, sulla sinistra sono presenti le aree dedicate ai servomotori che possono essere connessi alle porte A, B o C. In alto a destra vi sono le aree destinate ai sensori che si possono collegare alle porte 1, 2, 3 o 4. Nella parte più in basso a sinistra, sono presenti le caselle tramite le quali scegliere se produrre o meno un file di log del movimento dei motori durante una simulazione, ed una barra scorrevole per impostare la velocità della simulazione. Nella parte in basso a destra, infine, vi sono i bottoni del brick, il tasto di avvio/arresto della simulazione, e le aree per il display ed il riproduttore di suoni.

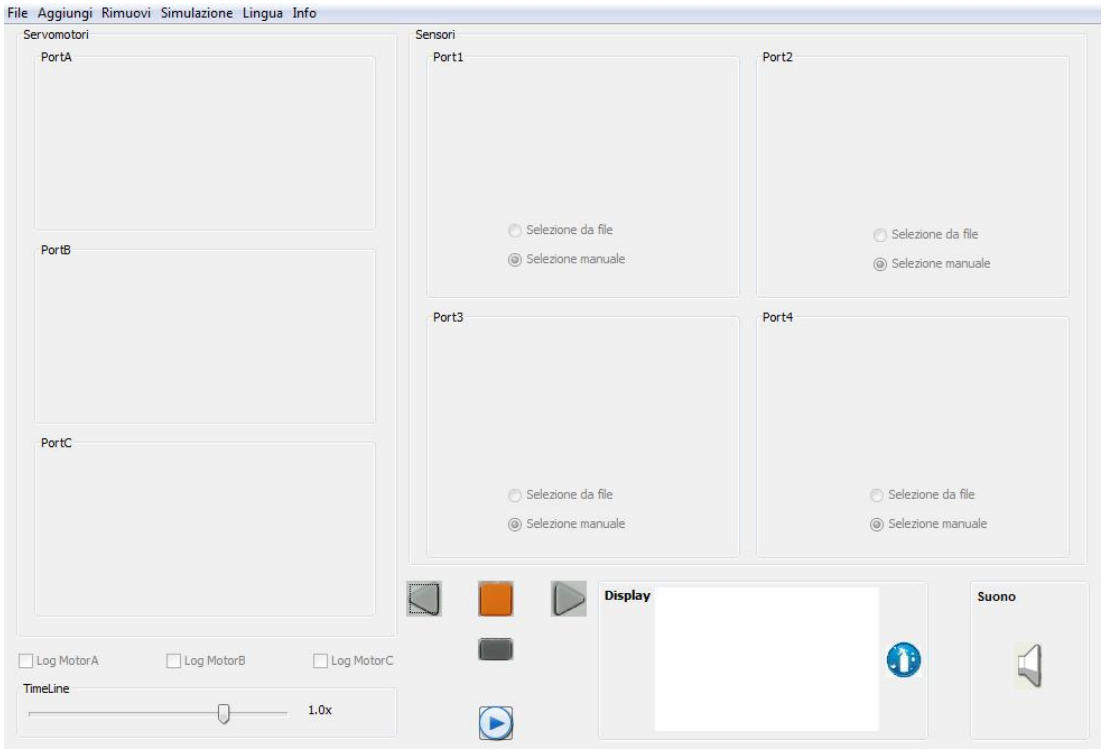


**Figura 6.1:** *Layout del simulatore*

## Aggiunta di un servomotore

Per aggiungere uno o più servomotori alla configurazione del robot, bisogna scegliere la voce di menù *Aggiungi* → *Servomotore* → *Port(X)*, selezionando la porta a cui si desidera collegare il motore (figura 6.2).

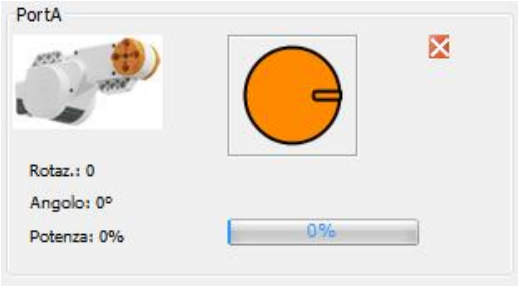


**Figura 6.2:** *Servomotore*

Una volta connesso, un servomotore può essere rimosso semplicemente cliccando sul tasto *X* presente nell'angolo in alto a destra della sua rappresentazione grafica. La voce di menù *Rimuovi* → *Servomotori* "sconnette", invece, tutti gli eventuali motori collegati.

### Aggiunta di un sensore

La versione corrente (0.9c) del simulatore supporta le seguenti quattro tipologie standard di sensori NXT:

- Luce (figura 6.3)

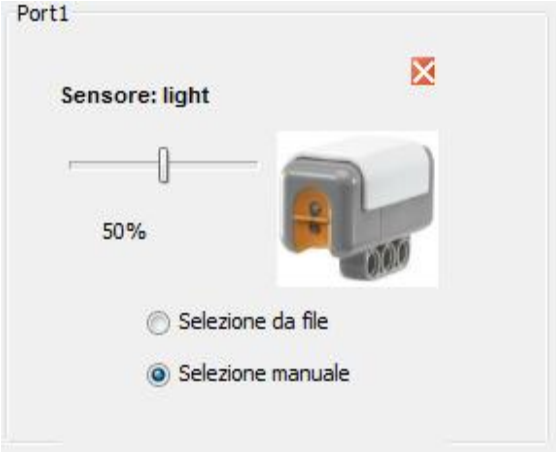


**Figura 6.3:** *Sensore di luce*

- Suono (figura 6.4)

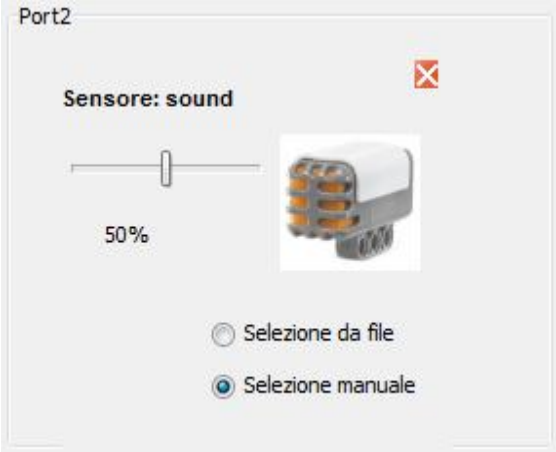


**Figura 6.4:** *Sensore di suono*

- Tocco (figura 6.5)


**Figura 6.5:** *Sensore di tocco*

- Ultrasuoni (figura 6.6)


**Figura 6.6:** *Sensore ad ultrasuoni*

Per aggiungere uno o più sensori all'interfaccia grafica del simulatore, scegliere la voce di menù *Aggiungi* → *Sensore* → <*tipo sensore*>: a questo punto appare un pop-up menù dal quale è possibile selezionare a quale porta connettere il sensore scelto. Qualora alla porta prescelta sia già collegato un altro sensore, quest'ultimo viene rimpiazzato dal nuovo sensore.

Una volta connesso, un sensore può essere rimosso semplicemente cliccando sul tasto *X* presente nell'angolo in alto a destra della sua rappresentazione grafica. La voce di menù *Rimuovi* → *Sensori* "sconnette", invece, tutti gli eventuali sensori collegati.

Notare che la voce di menù *Rimuovi* → *Tutti* permette di rimuovere tutti i servomotori ed i sensori eventualmente presenti nell'interfaccia grafica del software.

## Caricamento di un file eseguibile (*.RXE*)

Per caricare un file eseguibile in formato *.RXE* nel simulatore è sufficiente accedere alla voce di menù *File* → *Carica .RXE*: quest'ultima favorisce, infatti, l'apertura di una finestra, tramite la quale navigare all'interno del proprio file system, alla ricerca del file che si desidera simulare.

## Salvataggio/Caricamento di una configurazione

Tramite la voce di menù *File* → *Salva configurazione* è possibile salvare la configurazione attuale che si è allestita per i pannelli relativi a servomotori e sensori.

Accedendo a tale voce, infatti, è possibile specificare, in un percorso del proprio file system, il nome col quale salvare la suddetta configurazione, la quale verrà memorizzata in un file con estensione *.DAT*.

Mediante la voce di menù *File → Carica configurazione* è possibile, invece, caricare una configurazione già esistente per i pannelli del simulatore. Cliccando su tale voce, infatti, avviene l'apertura di una finestra, mediante la quale navigare all'interno del proprio file system, per ricercare il file contenente la configurazione che si intende allestire.

## Impostazione della Lingua

Come già accennato in precedenza, il simulatore può essere eseguito in due lingue diverse: Italiano ed Inglese. Il passaggio da una lingua all'altra avviene mediante la voce di menù *Lingua → <lingua non corrente>*; una volta selezionata, la variazione della lingua richiede, però, un riavvio del software, al fine di rendere effettivo il cambiamento richiesto.

## Informazioni su *NXTSimulator*

Tramite la voce di menù *Info → Info su NXTSimulator* è possibile visualizzare una finestra contenente informazioni riguardanti la versione attuale del programma, e gli sviluppatori del medesimo (figura 6.7).


**Figura 6.7:** *Informazioni sul simulatore*

## Guida all'utilizzo

Ora che si è tracciata una panoramica delle funzionalità offerte all'utente dall'interfaccia grafica del simulatore, si possono delineare i singoli passi di una normale sequenza d'utilizzo del medesimo:

- tramite la voce di menù *File → Carica .RXE* scegliere il file eseguibile che si intende simulare;
- allestire la corretta configurazione delle porte del robot, aggiungendo ad uno ad uno gli eventuali servomotori e/o sensori nelle porte di destinazione, o caricandola, se presente in un file già esistente, mediante la voce di menù *File → Carica configurazione*;
- impostare eventuali parametri iniziali dei sensori, come ad esempio, lo stato del sensore di tocco;
- mettere la spunta sulle apposite caselle, se si desidera produrre eventuali file di log del movimento dei servomotori;
- pulire il display mediante la pressione dell'apposito tasto rotondo blu e bianco posto alla sua destra, eliminando così eventuali "tracce" lasciate su di esso da simulazioni precedenti;
- avviare la simulazione tramite la voce di menù *Simulazione → Avvia*, o mediante il più comodo tasto di avvio rapido della medesima;
- osservare quindi l'andamento della simulazione fino al suo termine; qualora si desideri interromperla anzitempo, è sufficiente accedere alla voce di menù *Simulazione → Arresta*, o cliccare sul comodo tasto di arresto rapido della medesima, oppure ancora sul pulsante inferiore tra quelli del brick, incaricato anche nel robot reale di interrompere l'esecuzione del programma in corso;
- una volta terminata la simulazione, è possibile ripetere la medesima, o allestire un'altra configurazione, caricando quindi un altro file eseguibile, oppure semplicemente uscire dal programma.

E' importante che l'utente sia consapevole del fatto che è necessario porre particolare attenzione nell'allestimento della configurazione adatta per il file eseguibile da simulare. *NXTSimulator* non è, infatti, in grado di determinare a priori, sulla base del programma datogli in ingresso, gli eventuali sensori e motori, e le porte a cui collegarli. Una configurazione errata può comportare un comportamento anomalo del simulatore. Questo problema si verifica comunque anche nel caso del robot reale, qualora venga allestito in modo non conforme al programma da eseguire.

Durante una simulazione, che preveda l'utilizzo di uno o più servomotori, il loro movimento è simulato dalla rotazione dell'immagine arancione della figura 6.2 di

pagina 140. Nel pannello relativo ad un motore sono, inoltre, presenti una barra che specifica la potenza ad esso attualmente applicata, il numero di giri completi (rotazioni di 360°) da esso compiuti, la sua posizione angolare rispetto a quella occupata ad inizio simulazione, ed il valore numerico della potenza ad esso applicata, identico a quello riportato nella barra di cui sopra.

### File di *log* dei motori

Durante una simulazione, in cui si utilizzino uno o più servomotori, l'utente può anche scegliere di far produrre al simulatore un file di *log* che tenga traccia delle variazioni temporali della posizione angolare dell'immagine che simula il movimento di un motore, di cui si è parlato al termine della sottosezione precedente. Mettendo la spunta sull'apposita casella denominata *Log Motor(X)*, ed avviando poi la simulazione, viene prodotto, al termine della medesima, un file *.TXT* contenente coppie di valori nel formato *<istante - posizione angolare>*, che riporta tutte le variazioni di posizione (in gradi), rispetto a quella iniziale (denotata col valore '0'), che l'immagine relativa al motore *X* ha subito durante la simulazione ed i rispettivi istanti (in millisecondi) successivi all'avvio della medesima, in cui tali variazioni sono avvenute. Così facendo l'utente può disporre di una serie di valori da dare magari successivamente in ingresso ad una funzione di disegno (*plotting*) che tracci un grafico del movimento del motore.

### Valori dei sensori

Ciascun sensore, tra quelli disponibili nel simulatore, può avere due tipi distinti di input, specificabili mediante i *radio button* [22s] rappresentati nelle figure dalla 6.3 alla 6.6, denominati rispettivamente *Selezione da file* e *Selezione manuale*. Come quest'ultimi già suggeriscono, la selezione da file permette all'utente di specificare l'ingresso del sensore mediante un file di testo, mentre quella manuale impone all'utente stesso di far variare il valore letto dal sensore tramite la barra scorrevole (o gli appositi pulsanti nel caso del sensore di tocco) presente nella grafica ad esso relativa. Qualora si scelga di fornire manualmente il valore da leggere ad un sensore, ed esso non sia un sensore di tocco, questo va specificato selezionando, appunto, il valore desiderato mediante l'apposita barra scorrevole, la quale permette di specificare un valore percentuale nell'intervallo [0 ; 100] per i sensori di luce e di

suono, ed un valore in centimetri nell'intervallo [0 ; 250] per quello ad ultrasuoni. Nel caso si abbia a che fare con un sensore di tocco, invece, vi sono due tasti sui quali poter agire: un tasto *Bump*, da premere per simulare una pressione esercitata sul sensore, ed un tasto *Switch* da premere una volta per simulare una situazione di pressione costante sul sensore, e da premere nuovamente per tornare alla situazione di pressione nulla. Qualora, invece, si scelga di fornire un input da file al sensore, la faccenda diventa un po' più complicata. Vediamo quindi di fare chiarezza a riguardo.

Il file di testo in formato *.TXT* dato in ingresso al sensore può essere pensato come diviso in sezioni: ogni sezione rappresenta un intervallo con una determinata funzione di interpolazione, ed è formata da una stringa che indica al simulatore quale metodo di interpolazione applicare ai valori successivi, seguita da un certo numero di coppie $<x - y(x)>$, dove $x$ è un istante temporale e $y(x)$ è il valore della funzione in quell'istante. Dopo un numero di campioni arbitrariamente lungo (a discrezione dell'utente) il file può terminare o presentare un'altra sezione: anche qui la prima riga dovrà contenere la specifica indicante il metodo di interpolazione. Quest'ultima si suppone essere diversa da quella della sezione che la precede, ma non sono imposti vincoli in tal senso: la scelta resta esclusivamente di pertinenza dell'utente. I metodi di interpolazione disponibili sono i seguenti:

- *const*: rappresenta l'elaborazione costante. Una volta letto un valore, questi viene mantenuto fisso fino alla lettura di una nuova coppia $<x - y(x)>$. A questo punto il nuovo valore $y(x)$ diventerà il valore di uscita per tutti i valori campionati fino alla lettura di una nuova coppia e così via;
- *linear*: elaborazione di tipo lineare. Per ogni intervallo di valori si calcola la retta che congiunge i due estremi e si ottiene il valore della funzione nel punto calcolando il valore di tale retta nell'istante desiderato;
- *pol3*: elaborazione tramite polinomio di terzo grado. Oltre a volere che la curva di interpolazione passi per gli estremi dell'intervallo, si impone che la sua derivata prima ivi si annulli. Questo permette di avere una funzione interpolante derivabile in tutti i suoi punti, dandole così raccordi smussati;
- *sigm*: elaborazione tramite sigmoide. Una curva ottenuta tramite interpolazione sigmoidale è una funzione esponenziale, simile ad un polinomio di terzo grado, che varia molto lontano dagli estremi dell'intervallo, mentre ha variazioni meno veloci in vicinanza di essi, dandole

una forma ad 'S';

- *sin*: interpolazione tramite seno. Si fa in modo che una funzione del tipo A+B*sin(ωx+φ) passi per i punti estremi dell'intervallo;
- *cos*: interpolazione tramite coseno. Si fa in modo che una funzione del tipo A+B*cos(ωx+φ) passi per i punti estremi dell'intervallo;
- *spline*: interpolazione mediante spline. La spline è un polinomio di terzo grado, in cui si impone l'annullamento della derivata seconda nei punti estremi di ogni intervallo. In tali punti cioè si vuole avere un cambio di flesso.

Nell'immagine proposta qui di seguito è riportato un esempio del contenuto di un file di input per i sensori, che utilizza un'interpolazione di tipo sigm.

Ulteriori approfondimenti riguardanti l'input da file per i sensori sono disponibili nell'elaborato citato al punto [9t] della bibliografia.

```
sigm
0 125
4 240
8 35
12 240
16 35
18 240
20 35
```

**Figura 6.8:** *Esempio di file di input per i sensori*

## Esempio di utilizzo

Viene ora proposto un esempio completo di utilizzo del simulatore: si parte con la costruzione del programma mediante NXT-G, per arrivare poi alla simulazione vera e propria del file compilato.

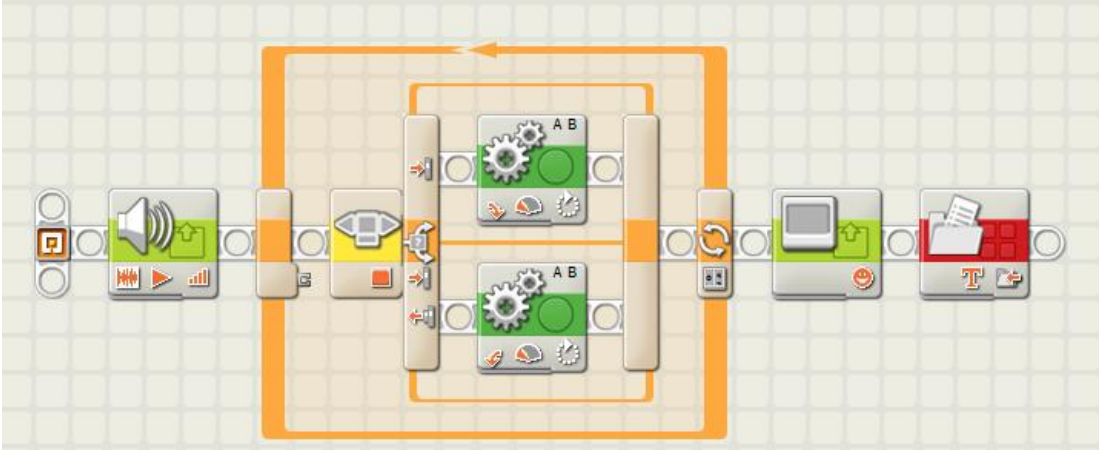


**Figura 6.9:** *Programma NXT-G*

Il programma creato mediante NXT-G, e riportato qui sopra, riproduce inizialmente un file audio e poi entra in un *loop* che viene eseguito per cinque volte, all'interno del quale è presente un costrutto *switch*, il quale esegue il flusso raffigurato in alto in figura 6.9 qualora il tasto centrale del brick sia premuto, mentre esegue quello più in basso in caso contrario: nel primo flusso il motore collegato alla porta A ad ogni esecuzione del loop compie un giro completo in senso antiorario, mentre quello collegato alla porta B lo compie in senso orario; nel secondo flusso, invece, accade l'esatto contrario. Terminato il loop, viene visualizzata un'immagine sul display e viene prodotto un file di testo, nel quale viene scritta una frase di prova.

Vengono di seguito elencati i passi che l'utente deve seguire per creare il programma NXT-G appena descritto.

- Il primo blocco deve avere impostata come azione quella di riproduzione di un file audio, selezionando quest'ultimo a piacere dalla lista di quelli messi a disposizione da NXT-G;
- il costrutto *loop* va impostato col contatore del numero di iterazioni da eseguire pari al valore '5';
- il costrutto *switch* va impostato col campo *Control* uguale a *Sensor*, il campo *Sensor* uguale a *NXT Buttons*, il campo *Button* uguale a *Enter button*, ed il campo *Action* uguale a *Pressed*;


**Figura 6.10:** *Impostazioni blocco 'switch'*

- nel primo flusso dello switch viene posizionato un blocco che impone il movimento dei motori collegati alle porte A e B, direzione in avanti, potenza pari a '41', campo *Duration* impostato a '360' gradi, e barra del fattore di sterzata spostata tutta a destra ('100');


**Figura 6.11:** *Impostazioni motori del flusso superiore*

- il secondo flusso dello switch deve avere le stesse specifiche del precedente, eccezion fatta per la barra del fattore di sterzata, la quale deve essere spostata tutta a sinistra ('-100');
- il penultimo blocco deve avere impostata come azione quella di visualizzazione di un'immagine, e quest'ultima va selezionata a piacere dal *pool* di quelle messe a disposizione da NXT-G;
- l'ultimo blocco è di tipo *File Access*, e specifica come azione la scrittura di un file di testo, con nome e contenuto del medesimo impostati a piacere dall'utente.


**Figura 6.12:** *Impostazioni blocco File Access*

Una volta realizzato il programma desiderato, basta accedere alla voce di menù *Tools → Download to File...*, scegliere il percorso dove salvare il file eseguibile *.RXE* da originare, e cliccare *Download* per crearlo.

Per procedere ora con la simulazione del programma appena prodotto, è necessario avviare *NXTSimulator*, ed eseguire i seguenti passi:

- cliccare sulla voce di menù *File → Carica .RXE* e sfogliare il file system del proprio sistema, per reperire e caricare il file *.RXE* originato in precedenza;
- accedere alla voce di menù *Aggiungi → Servomotore* per aggiungere i due motori alle porte A e B del simulatore;
- porre la spunta sulle apposite caselle di scelta se si desidera produrre un file

di log per uno od entrambi i motori;

- cliccare sul tasto di avvio rapido della simulazione, per avviare la medesima;
- durante la simulazione la pressione o meno del tasto centrale del brick da parte dell'utente va a determinare, di volta in volta, quale dei due flussi dello switch di cui sopra eseguire;
- attendere il termine della simulazione, od arrestarla anzitempo mediante la pressione del tasto di arresto rapido della medesima.

# Conclusioni

L'attività di tesi presentata nel corso di questo elaborato ha portato al rilascio di una nuova versione del simulatore per il robot Lego® Mindstorms® NXT: *NXTSimulator* 0.9c. L'applicativo così ottenuto è stato sottoposto a notevoli miglioramenti rispetto alla sua versione precedente, ossia la 0.9b. Come prefissatisi inizialmente, si è reso il software capace di simulare ulteriori dispositivi e funzionalità del robot, come ad esempio il display, il riproduttore di suoni, il gestore di file, ed i bottoni del brick. Si è, inoltre, risolta una grande quantità di bug presenti nel simulatore, i più rilevanti dei quali sono sicuramente quelli relativi alla simulazione dei servomotori, ed alla schedulazione dei clump di un programma, che ormai da due anni affliggevano il software, e la cui risoluzione è stata quindi provvidenziale.

Il risultato raggiunto si può considerare alquanto soddisfacente, ed ha permesso ad *NXTSimulator* di progredire notevolmente verso un ormai prossima versione definitiva.

Per quanto riguarda i possibili sviluppi futuri del software, si cita la possibilità di permettergli di simulare nuove tipologie di sensori accessori rispetto a quelli inclusi nella versione standard del robot LEGO®, come ad esempio la bussola, il giroscopio, o l'accelerometro. Un lavoro più complesso potrebbe, invece, essere quello di pensare ad un'estensione dell'operato del simulatore in un ambiente 3D.

# Bibliografia

## Testi


[1t] Cay S. Horstmann, "*Concetti di informatica e fondamenti di Java*", III Edizione, Apogeo, 2005.

[2t] Nicola Andreose, "*Sviluppo di aggiornamenti per il simulatore didattico NXTSimulator*", Tesi di Laurea, Università degli Studi di Padova, 2010.

[3t] Giorgio Clemente, Federico Filira, Michele Moro, "*Sistemi Operativi (Architettura e Programmazione concorrente)*", II Edizione, Libreria Progetto Padova, 2006.

[4t] Sergio Congiu, "*Architettura degli Elaboratori (Organizzazione dell'hardware e programmazione in linguaggio Assembly)*", V Edizione, Pàtron Editore, 2007.

[5t] Marco Zanchetta, "*Simulatore funzionale per il robot didattico Mindstorms NXT*", Tesi di Laurea, Università degli Studi di Padova, 2009.

[6t] Andrea Donè, "*Realizzazione di un semplice simulatore per il robot didattico Lego Mindstorms*", Tesi di Laurea, Università degli Studi di Padova, 2009.

[7t] Filippo Buletto, "*Realizzazione di un simulatore semplificato del robot educativo Mindstorm NXT*", Tesi di Laurea, Università degli Studi di Padova, 2009.

[8t] P. Mazzoldi, M. Nigro, C. Voci, "*Elementi di Fisica (Meccanica - Termodinamica)*", EdiSES, 2001.

[9t] Marco Pierobon, "*Simulazione continua nel tempo dei sensori del robot didattico Mindstorm NXT*", Tesi di Laurea, Università degli Studi di Padova, 2009.


## Siti Internet


[1s] *http://mindstorms.lego.com*.

[2s] *http://it.wikipedia.org/wiki/LEGO_Mindstorms*.

[3s] *http://www.adrirobot.it/lego/mindstorms_nxt/mindstorms_ntx.htm*.

[4s] *http://www.hitechnic.com/downloadnew.php?category=13*.

[5s] *http://it.wikipedia.org/wiki/NetBeans*.

[6s] *http://www.ni.com/labview/*.

[7s] *http://bricxcc.sourceforge.net/*.

[8s] *http://www.terecop.eu/*.

[9s] *http://www.microsoft.com/robotics/*.

[10s] *http://it.wikipedia.org/wiki/LeJOS*.

[11s] *http://en.wikipedia.org/wiki/Vex_Robotics_Design_System*.

[12s] *http://en.wikipedia.org/wiki/Robotc*.

[13s] *http://it.wikipedia.org/wiki/Ordine_dei_byte*.

[14s] *http://it.wikipedia.org/wiki/Assembly*.

[15s] *http://www.amolamatematica.it/appunti/ascii.pdf*.

[16s] *http://download.oracle.com/javase/6/docs/api/*.

[17s] *http://www.ensta.fr/~diam/java/online/notes-java/GUI-lowlevel/graphics/43buffimage.html*.

[18s] *http://www.jfugue.org/*.

[19s] *http://www.anyexample.com/programming/java/java_play_wav_sound_file.xml*

[20s] *http://bricxcc.sourceforge.net/utilities.html*.

[21s] *http://www.hempeldesigngroup.com/lego/pblua/tutorial/pbluabasicmotors/*.

[22s] *http://it.wikipedia.org/wiki/Radio_button*.