Westfälische Wilhelms-Universität Münster

Ausarbeitung

Robot-Programmiersprachen

im Rahmen des Seminars Unterstützung von Landminendetektion durch Bildauswertungsverfahren und Robotereinsatz

Christian Hermanns

Klaus Schaefers

Themensteller: Prof. Dr. Xiaoyi Jiang
Betreuer: Dr. Dietmar Lammers
Institut für Informatik

Inhaltsverzeichnis

1 Einleitung 4

2 Robotics Invention System 2.0 6

2.1 Einführung 6
2.2 Grundlegende Elemente 7
2.3 IO-Funktionen 9
2.4 Beispiel 12

3 Not Quite C 13

3.1 Einführung: 13
3.2 Grundlegende Elemente 13
3.3 IO-Funktionen 15
3.4 Sonstiges 18
3.5 Beispiel 19

4 leJOS 19

4.1 Einführung 19
4.2 IO-Funktionen 20
4.3 Programmarchitektur: 23
4.4 Beispiel 24

5 brickOS 25

5.1 Einführung 25
5.2 IO-Funktionen 26
5.3 Beispiel 31

6 pbFORTH 32

6.1 Einführung 32
6.2 Grundlegende Elemente 33
6.3 IO-Funktionen 37
6.4 Beispiel 41

7 Industrieroboter 43

7.1 Einführung 43
7.2 Onlineprogrammierung 45
7.3 Offlineprogrammierung 46

8 Zusammenfassung 48

Literaturverzeichnis..50

1 Einleitung

Die vorliegende Arbeit beschäftigt sich mit Roboter-Programmiersprachen, im Speziellen mit Sprachen, die sich für die Programmierung von Lego Mindstorms eignen. Um einen Vergleich zur professionellen Programmierung zu ermöglichen, werden im Anschluss an die Vorstellung der Lego-Programmiersprachen noch einige Konzepte und Methoden zur Programmierung moderner Industrieroboter vorgestellt.

Zunächst werden Programmiersprachen behandelt, die mit der Lego-Firmware arbeiten. Dabei handelt es sich um das Robotics Invention System 2.0 (RIS) und Not Quite C (NQC). Im Anschluss werden die Sprachen leJOS, brickOS und pbForth vorgestellt, die jeweils mit einer eigenen Firmware arbeiten und so einige Einschränkungen der Lego-Firmware umgehen. Zu Beginn jedes Abschnitts stellen werden zunächst die grundlegenden Elemente der Sprache, wie Syntax und Kontrollstrukturen, erläutert und anschließend die Input- / Outputfunktionen zur Programmierung des Lego-Bricks vorgestellt. Den Abschluss bildet jeweils ein kleines Programm, welches die in Abbildung 1 beschriebene Funktionalität in den entsprechenden Sprachen implementiert. Dieses erscheint didaktisch sinnvoll, da der Leser so schnell auf die grundlegenden Unterschiede der einzelnen Sprachen aufmerksam wird. Das letzte Kapitel dieser Arbeit beschäftigt sich mit der Programmierung moderner Industrieroboter, da diese sich deutlich von der Programmierung der Lego Mindstorms-Produkte unterscheidet.

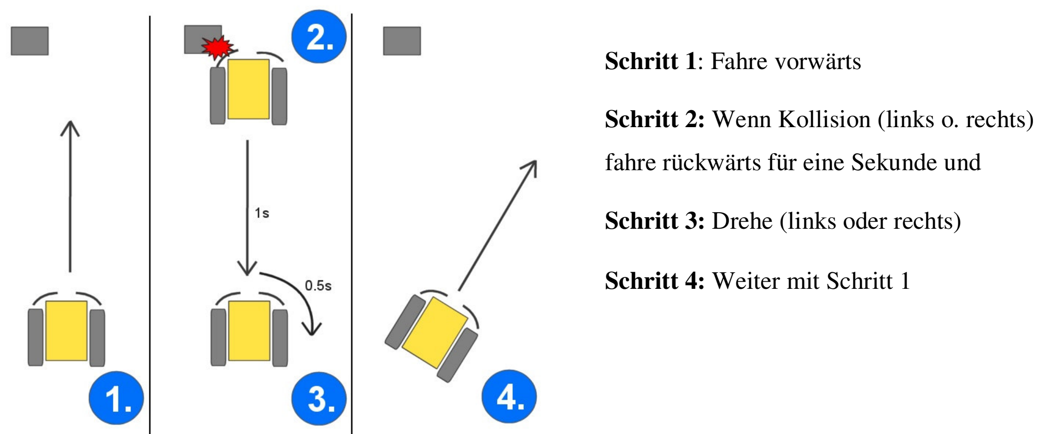


Abbildung 1: Der LegoBot -Algorithmus

Doch bevor auf die einzelnen Sprachen im Detail eingegangen wird, folgt eine Vorstellung der zugrunde liegenden Technologie. Kern eines jeden Lego Mindstorms Roboters ist der so genannte RCX-Brick. Hierbei handelt es sich um einen Mikrocomputer, der für die Steuerung der Motoren sowie für die Verarbeitung der Sensorwerte verantwortlich ist. Im Brick arbeitet ein Hitachi H8 / 3292 Prozessor, welcher mit 16 Mhz getaktet ist. Über einen 8 Bit Datenbus und einen 16 Bit Adressbus verwaltet er das 16 KB große ROM und den 32 KB umfassenden RAM. Für Input-Output Operationen stehen dem Brick drei Sensoreingänge, drei Motorausgänge, ein Display, ein Soundmodul und zusätzlich ein Infrarot-Port zur Verfügung. Von besonderer Bedeutung sind für einen Roboter in der Regel die Motoren und Sensoren, welche aus diesem Grund genauer Beschrieben werden. Ein Motor kann sich in folgenden Zuständen befinden:

Modus	Bedeutung
`FLOAT`	Motor ist frei beweglich
`OFF`	Motor ist gebremst
`ON`	Motor dreht sich

Tabelle 1: Motorzustände des RCX

Natürlich kann auch die Drehrichtung bestimmt werden. Damit der RCX die Sensorwerte richtig interpretieren kann, muss der Software zuvor mitgeteilt werden, um welche Art Sensor es sich handelt und in welchem Modus die Daten ausgeben werden sollen. Die Tabellen 2 und 3 fassen die wichtigsten Konfigurationen zusammen

Sensor	Beschreibung
`NONE`	beliebiger passiver Sensor
`TOUCH`	Berührungssensor
`TEMPERATURE`	Temperatursensor
`LIGHT`	Lichtsensor
`ROTATION`	Rotationssensor

Tabelle 2: Sensortypen des RCX

Modus	Beschreibung
`RAW`	`Rohwert von 0 bis 1023`
`BOOL`	`boolescher Wert (0 oder 1)`
`EDGE`	`Zählwert für Impulsflanken`
`PULSE`	`Zählwert für Impulse`
`PERCENT`	`Werte von 0 bis 100`

Tabelle 3: Sensormodi des RCX

Die Systemarchitektur besteht grob vereinfacht aus vier Schichten. Die unterste Schicht bildet der Hitachi Prozessor, der den Maschinencode ausgeführt. Die zweite Schicht stellt das ROM dar. Es ermöglicht grundlegende Systemaufrufe und ist mit dem BIOS moderner Personalcomputer vergleichbar. Die vorletzte Schicht bildet die Firmware, sie ist im RAM gespeichert und kein fester Bestandteil des Bricks. Die Aufgabe besteht darin, den Bytecode der Userprogramme zu interpretieren und durch den Aufruf entsprechender ROM-Routinen auszuführen. Da die Firmware im RAM residiert, kann sie leicht gegen neue Firmware ausgetauscht werden. Viele der vorgestellten Programmiersprachen machen von dieser Möglichkeit Gebrauch. Die vierte und damit letzte Schicht bilden die individuellen Userprogramme, die ebenfalls im RAM gespeichert sind.

2 Robotics Invention System 2.0

2.1 Einführung

Das von Lego erhältliche Robotics Invention System 2.0 (RIS) enthält neben dem RCX-Brick und den Legobausteinen zum Konstruieren von Robotern ebenfalls eine gleichnamige Entwicklungsumgebung zur Programmierung des RCX-Bricks. Bei der RIS-Software handelt es sich um eine graphische Entwicklungumgebung. Programme können daher mit dem RIS, durch die Anordnung von vorgegebenen graphischen Elementen, für z.B. Kontrollstrukturen oder Ein- und Ausgabebefehlen, erstellt werden. Diese graphischen Elemente werden auch als *Blöcke* bezeichnet. Im Gegensatz zu den klassischen, textbasierten Programmiersprachen ermöglicht dieser Ansatz einen besonders leichten und intuitiven Einstieg in die Programmierung des RCX-Bricks. Der Programmierer kann sich von Beginn an intensiver mit der Struktur und dem Ablauf seiner Programme befassen, da die Syntax der Sprache, welche durch die graphischen Elemente vorgegeben ist, nicht erst erlernt werden muss. Das RIS ermöglicht deshalb eine relativ einfache und intuitive Programmierung des RCX.

Die Nachteile des RIS machen sich besonders bei komplexeren Programmen bemerkbar. Während die graphische Entwicklungsumgebung die Übersichtlichkeit und Lesbarkeit kleinerer Programme deutlich erleichtert, erweist sie sich im Vergleich zu textbasierten Entwicklungsumgebungen, besonders bei aufwendigeren und längeren Programmen, als wesentlich unübersichtlicher. Hinzu kommt, dass die RIS-Programme im

Vergleich zu den anderen hier vorgestellten Programmiersprachen den stärksten Einschränkungen unterliegen (vgl. Tabelle 4).

Einschränkungen des RIS: (maximal)	
5	Programmslots
27	Variablen
16	Ereignismonitore
9	Routinen

Tabelle 4: Einschränkungen des RIS. Zur Erläuterung der Elemente vgl. Absatz 2.2

Den Ablauf des Programmierprozesses wird durch die in der Abbildung 2 dargestellte Softwarearchitektur des RIS verdeutlicht. Ein Programm wird auf dem PC mit Hilfe des graphischen Editors des RIS erstellt und in Bytecode, welcher von der Lego-Firmware ausgeführt werden kann, umgewandelt. Über den USB-Port und den IR-Tower werden die Programme in den für Benutzerprogramme reservierten Teil des RAMs des RCX geschrieben. Die übertragenen Programme können durch die Firmware, welche ebenfalls im RAM gespeichert ist ausgeführt werden.

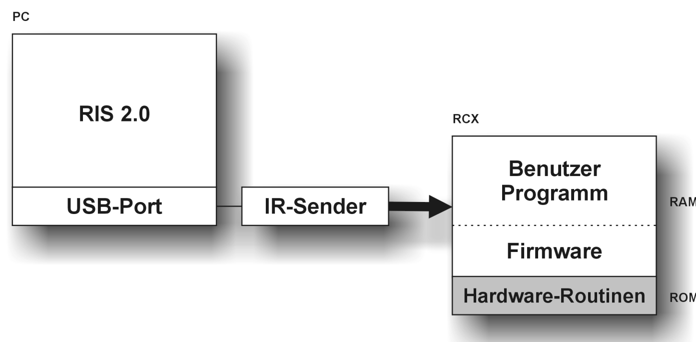


Abbildung 2: Softwarearchitektur des RIS

Als Ergänzung zu der folgenden Beschreibung der RIS-Entwicklungsumgebung können die Online-Dokumentation des RIS und die Einführung in die RIS-Programmierung von Knudsen aus [Kn99] herangezogen werden.

2.2 Grundlegende Elemente

Routinen

Ein RIS-Programm besteht aus maximal 9 Routinen. Die erste Routine besteht immer aus dem Hauptprogramm und wird beim Start des Programms aufgerufen. Wahlweise kann das Hauptprogramm um bis zu acht weitere Routinen, den Sensormonitoren, ergänzt werden. Die Ausführungsreihenfolge der Routinen wird durch zugewiesene Prio-

ritäten, welche sich auch nachträglich ändern lassen (vgl. erweiterte Funktionen), geregelt. Die Priorität wird immer dann berücksichtigt, wenn zwei Sensor-Monitore (s.u.) gleichzeitig aktiviert werden.

Priorität	Routine
1	höchste Priorität - nicht vergeben
2	Variablen-Monitore
3	Berührungssensor-Monitore
4	Lichtsensor-Monitore
5	Temperatur-Monitore und Drehungssensor-Monitore
6	IR-Sensor-Monitore
7	Timer-Sensor-Monitore
8	Hauptroutine

Tabelle 5: Standard-Prioritäten der RIS-Routinen

Sensormonitore

Ein Sensormonitor ist eine Routine, deren Ausführung durch einen überwachten Sensor getriggert wird. Ein Sensormonitor wird genau dann ausgeführt, wenn der von ihm überwachte Sensor einen ausgewählten Zustand annimmt. Es können alle bekannten Sensoren (Tast-, Licht-, Temperatur- und Drehsensor) sowie der IR-Port und Variablen überwacht werden.

Ereignismonitore

Zu den Ereignismonitoren zählen alle Sensormonitore, sowie die bedingten Kontrollstrukturen „`warte bis`“ und „`wiederhole bis`“ (s.u.). Es sind insgesamt nur 16 Ereignismonitore erlaubt.

Kontrollstrukturen

Das RIS unterstützt drei Arten von Kontrollstrukturen, die „`Warten`“-, „`Wiederholen`“- und „`Ja oder Nein`“-Blöcke. Mit den „`Warten`“ Blöcken ist es möglich, einen bestimmten Zeitraum, oder bis zum eintreten eines Ereignisses zu warten. Die „`Wiederholen`“-Blöcke unterstützen vier Arten von Schleifen, die for-Schleife („`Wiederholen`“), eine Endlosschleife („`Ständig wiederholen`“) und zwei repeat-Schleifen („`Wiederholen während`“, „`Wiederholen bis`“). „`Wiederholen bis`“ überwacht die Bedingung die ganze Zeit und bricht sofort mit dem Eintreten des Ereignisses ab und ist daher ist es auch ein Ereignismonitor.

2.3 IO-Funktionen

Die vom RIS zur Verfügung gestellten Ein- und Ausgabebefehle werden als „kleine Blöcke“ bezeichnet. Im Folgenden werden die einzelnen Ausgabebefehle näher erläutert.

Motorsteuerung

Die Blöcke für die Motorsteuerung gehören zur Kategorie „Leistung“. Motoren können an- und ausgeschaltet werden. Leistung und Drehrichtung sind ebenfalls einstellbar (vgl. Tabelle 6)

Block	Beschreibung
Ein	Aktiviert die ausgewählten Motoren
Ein für	Aktiviert die Motoren für eine bestimmte Dauer
Aus	Deaktiviert die Ausgewählten Motoren
Leistung einstellen	Stellt die Leistung der ausgewählten Motoren ein
Richtung einstellen	Stellt die Richtung der ausgewählten Motoren ein
Richtung umkehren	Kehrt die Richtung der ausgewählten Motoren um

Tabelle 6: Blöcke für die Motorsteuerung

Töne

Das RIS unterstützt sowohl das Abspielen von voreingestellten Tönen als auch das Erzeugen von Tönen beliebiger Frequenz und Länge. Zusätzlich ist es möglich den Lautsprecher zu deaktivieren um das Abspielen sämtlicher Töne zu unterdrücken.

Block	Beschreibung
Kurzton	Spielt einen von 6 voreingestellten Tönen
Ton	Spielt einen Ton mit angegebener Frequenz und Dauer
Ton aus	Deaktiviert das Abspielen von Tönen
Ton ein	Aktiviert das Spielen von Tönen

Tabelle 7: RIS-Blöcke für die Tonerzeugung

Display

Auf dem Display können mit Hilfe der Kommunikations-Blöcke Zahlen sowie die Werte von Variablen, IR-Meldungen, Timern und Sensoren ausgegeben werden. Zudem

lässt sich auch anzeigen, wie lange das laufende Programm bereits ausgeführt wird (vgl. Tabelle 8).

Block	Beschreibung
`Wert anzeigen`	Zeigt einen Wert (-9999..9999), eine Meldung, eine Variable, einen Timer oder einen Sensorwert auf dem Display an
`Uhr anzeigen`	Zeigt auf dem Display an, wie lange das laufende Programm bereits ausgeführt wird

Tabelle 8: RIS-Blöcke für die Displaysteuerung

Infrarot

Die für den Austausch von Daten über die IR-Schnittstelle zur Verfügung gestellten Blöcke ermöglichen lediglich das Senden bzw. Empfangen eines einzigen Bytes (vgl. Tabelle 9). Außerdem ist zu beachten, dass der Empfangs-Puffer ebenfalls nur ein Byte speichern kann. Um weitere Bytes empfangen zu können muss der Puffer nach jedem eintreffenden Byte wieder gelöscht werden. Für den Austausch von größeren Datenmengen ist das RIS daher kaum geeignet.

Block	Beschreibung
`IR-Meldung senden`	Sendet ein Byte über den IR-Sender
`IR-Meldung löschen`	löscht den IR-Empfangspuffer (Es kann immer nur ein Byte empfangen werden)

Tabelle 9: RIS-Blöcke für die IR-Kommunikation

Variablen

Das RIS erlaubt eine maximale Anzahl von 27 Variablen, die Werte im Bereich von -3276,7 bis +3276,8 annehmen können. Für die Speicherung einer Variablen werden daher 2 Byte benötigt. Mit den in Tabelle 10 dargestellten Blöcken können Variablen durch Zuweisung, Addition, Subtraktion und Division von Werten manipuliert werden. Die für die Manipulation verwendeten Werte können Zahlen, andere Variablen, Sensor-Werte, Timer-Werte und empfangene IR-Meldungen sein. Des Weiteren besteht die Möglichkeit Variablen zu negieren sowie die Betragsoperation durchzuführen.

Block	Beschreibung
`Einstellen`	Weist einer Variablen einen Wert zu
`Addieren`	Addiert einen Wert zu einer Variablen
`Subtrahieren`	Subtrahiert einen Wert von einer Variablen
`Multiplizieren`	Multipliziert einen Wert mit einer Variablen
`Dividieren`	Dividiert eine Variable durch einen Wert
`Positiv`	Weist eine Variablen den Betrag ihres aktuellen Wertes zu
`Negiere`	Dreht das Vorzeichen des Wertes einer Variablen um

Tabelle 10: RIS-Blöcke für Variablenoperationen

Zurücksetzen

Mit Hilfe der Zurücksetzen-Blöcke können Sensoren und Timer zurückgesetzt und kalibriert werden (vgl. Tabelle 11)

Block	Beschreibung
`Licht zurücksetzen`	Kalibriert den Lichtsensor neu
`Timer zurücksetzen`	Setzt den Timer zurück
`Temp zurücksetzen`	Kalibriert den Temperatursensor neu
`Drehung zurücksetzen`	Setzt den Zähler des Drehsensors zurück

Tabelle 11: RIS-Blöcke für das Zurücksetzen und Kalibrieren von Sensoren

Erweiterte Funktionen

Tabelle 12 zeigt Blöcke aus der Kategorie „erweiterte Funktionen“. Neben der Festlegung der Ausführungs-Priorität von Routinen können einzelne Motoren deaktiviert und deren Laufrichtung eingestellt werden. Der „Programm Ende“-Block beendet ein Programm, indem zusätzlich zum Hauptprogramm auch alle vorhandenen Sensormonitore beendet werden. Dies ist notwendig, da nach dem beenden des Hauptprogramms die Sensor-Monitore nicht automatisch beendet werden.

Block	Beschreibung
`Priorität festlegen`	Legt die Priorität einer Routine fest
`Globale Umkehr`	Kehrt die Richtung der Motoren für alle weiteren Befehle um
`Globale Richtung`	Setzt die Richtung der Motoren für alle weiteren Befehle
`Leistung aus`	Schaltet die Stromzufuhr der Motoren ab
`Leistung ein`	Hebt die Wirkung von „Leistung aus“ auf
`Programm Ende`	Beendet das Programm (Hauptproramm und Sensormonitore)

Tabelle 12: RIS-Blöcke für erweiterte Funktionen

2.4 Beispiel

Das RIS-Beispiel zur Steuerung des LegoBots besteht aus 3 Routinen. Dem Hauptprogramm (1), dass mit einer Endlosschleife den LegoBot vorwärts bewegt,.und den beiden Sensormonitoren (2,3), die die beiden Berührungssensoren überwachen und die Ausweichmanöver durchführen. Nach der der Unterbrechung durch einen Sensormonitor wird die Ausführung der Hauptroutine fortgesetzt. Besonders bei der Umsetzung der relativ einfachen LegoBot-Steuerung zeigen sich die Vorteile der graphischen Entwicklungsumgebung. Die Steuerungslogik des LegoBots lässt sich fast intuitiv in ein RIS-Programm umsetzen und die vorhandenen Einschränkungen gegenüber anderen Programmiersprachen fallen hier nicht ins Gewicht.

Abbildung 3: RIS 2.0 LegoBot-Beispielprogramm

3 Not Quite C

3.1 Einführung:

Not Quite C oder abgekürzt NQC ist eine weitere Programmiersprache für die Lego Mindstormsprodukte. Es werden neben dem RCX ebenfalls Lego Cybermaster und Lego Scout unterstützt. NQC erfordert im Gegensatz zum Robot Invention System eine textbasierte Programmierung, diese bietet aber deutlich mehr Funktionalität und lässt eine bessere Programmgliederung zu. NQC arbeitet ebenfalls mit der original Lego Firmware, denn der NQC-Compiler erzeugt Lego Assembler Code (LASM). Aus diesem Grund unterscheidet sich die Architektur nur geringfügig vom RIS. Statt durch das RIS wird der Bytecode auf dem Hostrechner mit Hilfe des NQC-Übersetzers erzeugt. Folgende Grafik verdeutlicht die Architektur.

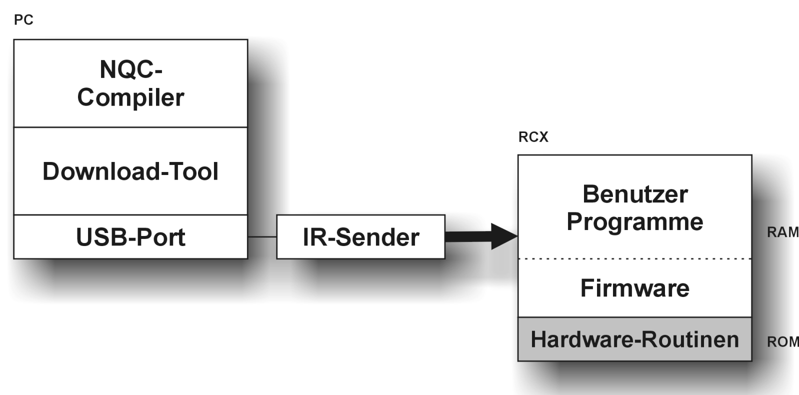


Abbildung 4: NQC Softwarearchitektur

Weitere interessante Features sind die Unterstützung von Multitasking und die Bereitstellung einer umfangreichen API zur Arbeit mit den Motoren und Sensoren. Eine Beschreibung der Sprache lässt sich auch bei [Ba03] und [Kn99] finden.

3.2 Grundlegende Elemente

Syntax

Wie der Name suggeriert, hat NQC viele Ähnlichkeiten mit C. Besonders die Syntax von NQC entspricht exakt der C-Syntax. Weshalb wird im Rahmen dieser Ausarbeitung auf eine Beschreibung der grundlegenden Elemente der Programmiersprache verzichtet. Eine genaure Beschreibung der Syntax kann z. B. [Lo99] entnommen werden

In einem NQC-Programm stehen nur 32 globale Variablen zur Verfügung, seit RIS 2.0 aber, noch weitere zusätzliche 16 Lokale. Diese Restriktion ist auf die Verwendung der Lego Firmware zurück zuführen. Eine globale Variable kann im ganzen Programm benutzt werden, während eine lokale Variable nur in ihrem Block (Schleife, Funktion, etc) Gültigkeit besitzt. Eine weitere Restriktion der Lego Firmware spiegelt sich in der Tatsache wieder, dass alle Variable vom Typ „16 BIT Signed Integer“ sind. Eine Deklaration erfolgt durch das Schlüsselwort „int“. Mit dem RIS 2.0 wurde NQC um Arrays erweitert, allerdings verkleinert sich die Anzahl der freien Variablen um die Größe des Arrays.

Strukturelemente

Um eine vernünftige Gliederung der Programme zu gewährleisten, stellt NQC die drei Strukturelemente task, function und sub zur Verfügung. Jedes Element hat besondere Eigenschaften und Beschränkungen.

NQC-Programme bestehen maximal aus zehn Tasks, mindestens aber aus einem Task namens „main“, welcher auch als Einstiegpunkt der Programme fungiert. Tasks werden mit Hilfe des Schlüsselwortes „task“ und folgender Syntax definiert:

```
task <task_name>()
      {
      … // Programmcode
      }
```

Auf Grund der Multitaskingfähigkeit von NQC, können auch mehrer Tasks parallel arbeiten, dazu müssen sie mit der Anweisung „start <task_name>“ gestartet werden. Um einen Task anzuhalten existiert der Befehl „stop <task_name>“, die API-Funktion „StopALLTasks“ stoppt alle laufenden Tasks.

Eine weitere Möglichkeit den Quelltext zu strukturieren bieten Funktionen. In NQC handelt es sich dabei um so genannte „Inline-Funktionen“, d.h. beim Compilieren werden die Anweisungen der Funktion an die entsprechende Stelle im Sourcecode eingefügt. Das hat zur Folge, dass eine häufige Verwendung von Funktionen die Programmgröße unnötig ansteigen lässt. Obwohl Funktionen Argumente haben können, besitzen sie keine Rückgabewerte. Aus diesem Grund müssen sie immer mit dem Schlüsselwort „void“ vereinbart werden, dies führt zu folgender Syntax:

```
void <function_name> (Argumentliste)
      {
      … // Anweisungen
      }
```

Die Argumentliste kann leer sein, es können aber auch ein oder mehrere Argumente übergeben werden. Vor dem Argumentnamen muss der Typ angegeben werden. Es stehen vier Typen zur Auswahl, die zwar alle 16 Bit Signed Integer sind, aber unterschiedliche Einschränkungen in der Benutzung haben. Einerseits ist eine Wertübergabe möglich, andererseits eine Adressübergabe, zusätzlich kann zwischen konstanten und variablen Argumenten unterschieden werden.

Typ	Bedeutung	Einschränkung
`int`	Wertübergabe	keine
`const int`	Wertübergabe	die Werte können in der Funktion nicht geändert werden.
`int&`	Adressübergabe	nur Variablen können übergeben werden
`const int&`	Adressübergabe	die Werte können in der Funktion nicht geändert werden.

Tabelle 13: NQC Funktionsparameter-Typen

Eine Verwendung von „int“ und „const int“ sollte aber vermieden werden, da zur Laufzeit Hilfsvariablen eingeführt werden, welche die Anzahl der freien Variablen reduzieren.

Die dritte Möglichkeit, um Anweisungen zusammen zu fassen bieten Unterprogramme. Sie werden im Gegensatz zu Funktionen nur einmal in den Quelltext eingefügt, allerdings ist die maximale Anzahl auf acht beschränkt. Eine Argumentübergabe ist nicht möglich, und auch der rekursive Aufruf weiterer Unterprogramme ist nicht erlaubt. Der Aufbau eines Unterprogramms sieht wie folgt aus:

```
sub <sub_name> ()
    {
    … // Programmcode
    }
```

3.3 IO-Funktionen

Damit die Roboter mit ihrer Umwelt agieren können, besitzt NQC eine umfangreiche API für die Input / Outputfunktionen, und gestatten dadurch dem Entwickler einen bequemen Zugriff auf die Ressourcen des RCX-Brick.

Motoren

Die Motoren stellen die primäre Outputmöglichkeit des RCX dar. Sie werden in Not Quite C durch die Schlüsselwörter „OUT_A“ „OUT_B“ „OUT_C“ angesprochen, zu-

sätzlich sind sie von 0 bis 2 nummeriert. Für die drei Modi der Motoren sind folgende Schlüsselwörter reserviert:

Schlüsselwort	Modi
OUT_ON	ON: Motor ist aktiviert.
OUT_OFF	OFF: Motor ist aktiv gebremst
OUT_FLOAT	FLOAT: Motor kann frei bewegt werden.

Tabelle 14: NQC Motor-Konstanten

Die Drehrichtung wird durch die Konstanten „OUT_FWD“ für vorwärts und „OUT_REV“ für rückwärts spezifiziert. Die Aktivierung eines Motors geschieht durch den Befehl „setOutput (Motoren, Modus)“, das Bestimmen der Drehrichtung durch „setDirection (Motoren, Richtung)“. Das nachstehende Beispiel verdeutlicht die Verwendung der Anweisungen:

```
setOutput (OUT_A, OUT_ON);      // Motor A ist aktiviert
setDirection (OUT_A, OUT_FWD); // Motor A läuft vorwärts
```

NQC besitzt auch vereinfachte Funktionsaufrufe, z.B. „On (Motoren)“, „Off (Motoren)“, „Float (Motoren)“, „Fwd (Motoren)“ und „Rev (Motoren)“. Die Verwendung der Kurzformen hat zum Ergebnis:

```
On (OUT_A); );  // Motor A ist aktiviert
Fwd (OUT_A); // Motor A läuft vorwärts
```

Eine Veränderung der Motorleistung kann durch den Aufruf von „setPower (Motoren, Leistungsstufe)“ realisiert werden, die Leistungstufen reichen von null bis sieben.

Display

Die zweite Outputmöglichkeit ist das Display des RCX-Bricks. Es kann durch die Befehle „setUserDisplay (Zahl, Kommastelle)“ und „selectDisplay (Sensor)“ angesteuert werden. Soll die Zahl „12,34“ ausgegeben werden, ist folgendes Statement nötig:

```
setUserDisplay (1234,2); // Display zeigt 12,34
```

Wenn Sensorwerte direkt auf dem Display gezeigt werden sollen, geschieht dies durch:

```
selectDisplay (DISPLAY_SENSOR_1); // Display zeigt die
                                  // Werte von Sensor 1
```

Sensoren

Die drei Sensoren sind intern in NQC von 0 bis 2 nummeriert. Alternativ sind die Wörter „SENSOR_1“, „SENSOR_2“ und „SENSOR_3“ reserviert. Um die Sensoren nutzen

zu können, muss dem System zuerst ihr Typ und ihr Modus mitgeteilt werden. Dies erfolgt durch die Anweisungen „SetSensorType (Sensor, Typ)“ und „SetSensorMode (Sensor, Modus)“. Die definierten Werte für die Sensortypen und Modi sind nachfolgenden Tabellen zu entnehmen.

NQC - Schlüsselwort	Sensortyp
SENSOR_TYPE_NONE	beliebiger passiver Sensor
SENSOR_TYPE_TOUCH	Berührungssensor
SENSOR_TYPE_TEMPERATURE	Temperatursensor
SENSOR_TYPE_LIGHT	Lichtsensor
SENSOR_TYPE_ROTATION	Rotationssensor

Tabelle 15. NQC Sensortypen

Schlüsselwort	Sensormodus
SENSOR_MODE_RAW	Rohwert von 0 bis 1023
SENSOR_MODE_BOOL	boolescher Wert (0 oder 1)
SENSOR_MODE_EDGE	Zählwert für Impulsflanken
SENSOR_MODE_PULSE	Zählwert für Impulse
SENSOR_MODE_PERCENT	Werte von 0 bis 100

Tabelle 16: NQC Sensormodi

Wenn z.B. Sensor_1 ein Berührungssensor ist und boolesche Werte zurück liefern soll, erfolgt die Konfiguration durch die Anweisungen:

```
SetSensorTyp(SENSOR_1,SENSOR_TYPE_TOUCH);
SetSensorMode(SENSOR_1,SENSOR_MODE_BOOL);
```

Eine verkürzte Schreibweise zur Sensorkonfiguration ist durch die API-Funktion „setSensor (Sensor, Konfiguration)“ in NQC implementiert worden. Die Konfigurationen sind Kombinationen von Sensortypen und Modi. Für die meisten Sensoren ist eine passende Konfiguration gespeichert, wie an Tabelle 17 zu ersehen ist.

Konfiguration	Sensortyp	Sensormodus
SENSOR_TOUCH	SENSOR_TYPE_TOUCH	SENSOR_MODE_BOOL
SENSOR_LIGHT	SENSOR_TYPE_LIGHT	SENSOR_MODE_PERCENT
SENSOR_ROTATION	SENSOR_TYPE_ROTATION	SENSOR_MODE_ROTATION
SENSOR_CELSIUS	SENSOR_TYPE_TEMPERATURE	SENSOR_MODE_CELSIUS
SENSOR_PULSE	SENSOR_TYPE_TOUCH	SENSOR_MODE_PULSE
SENSOR_EDGE	SENSOR_TYPE_TOUCH	SENSOR_MODE_EDGE

Tabelle 17: NQC Konstanten für Sensor die Sensorkonfiguration

Nachdem ein Sensor registriert wurde, ist er einsatzbereit und kann ausgelesen werden. Dazu muss einer Variablen nur „SensorValue(n)“ zugewiesen werden, „n“ steht dabei für die interne Nummer des Sensors. Um die Daten von SENSOR_1 zu erhalten genügt folgende Zeile Code:

```
x = SensorValue(0); // x wird der momentane Wert von
                    // SENSOR_1 zugewiesen
```

3.4 Sonstiges

Datalog

Der RCX-Brick bietet zusätzlich zu den Variablen noch ein so genanntes „Datalog“. Es handelt sich dabei um einen Speicher der mit Variablen, Sensorwerten etc. gefüllt werden kann. Vor der Benutzung muss jedoch das Datalog erst mit der Operation „Create-Datalog(n)“ erzeugt werden, „n“ ist die Anzahl der möglichen Einträge. Einträge werden mit dem Befehl „AddToDatalog(Wert)“ gespeichert. Allerdings gibt es einige Einschränkungen bei der Benutzung. Es ist z.B. nicht möglich, das Datalog mit NQC auszulesen. Die Daten können nur am Hostrechner ausgelesen werden. Dazu muss NQC mit den Parametern „datalog_full“ bzw. „-datalog“ ausgeführt werden.

Timer

Wenn sich der Programmierer für die Benutzung von NQC entschieden hat stehen ihm vier 16 Bit Timer zur Verfügung, sie werden im Abstand von 100 ms inkrementiert, dass heißt ein Überlauf findet nach ca. 55 Min statt. Ausgelesen werden die Timer ähnlich wie Sensoren, indem einer Variablen der Ausdruck „Timer (n) zugewiesen wird. Mit „ClearTimer(n)“ lässt sich ein Timer zurück stellen, mit „SetTimer (n, Wert)“ auf einen beliebigen Wert setzen. Wird eine präzisere Zeitauflösung gebraucht, kann auf die Anweisung „FastTimer(n)“ zurückgreifen.

IR-Port

Der Infrarotport kann in NQC ebenfalls benutzt werden um einfache Nachrichten zu versenden. Jede Nachricht ist genau ein Bit groß und hat einen Wert zwischen 0 und 255. Die zuletzt empfangene Nachricht kann mit „Message()“ vom Nachrichtenpuffer gelesen werden. Sollte aus irgendeinem Grund die Löschung des Puffers nötig sein, muss der Befehl „ClearMessage()“ angewendet werden. Das Versenden einer Nachricht verfolgt durch die Anweisung „SendMessage(Wert)“, dabei ist zu beachten, dass nur ein

Bit übertragen wird. Ist das Argument größer, werden nur die acht niederwertigsten Bits versendet.

3.5 Beispiel

Der Algorithmus zur Steuerung des Legobots sieht in NQC so aus:

```
task main ()
{
   SetSensor (SENSOR_1, SENSOR_TOUCH); // Sensor 1 aktivieren
   SetSensor (SENSOR_3, SENSOR_TOUCH); // Sensor 3 aktivieren
   On (OUT_A+OUT_C); // Motor A und C einschalten
   Fwd(OUT_A+OUT_C); // Beide Motoren  vorwärts fahren lassen
   while(true)
   {
      If (SENSOR_1) // Zusammenstoß vorne links
      {
         Rev (OUT_A + OUT_C); // Rückwärts für 1 s.
         Wait (100);
         Fwd(OUT_A); // Nach rechts drehen
         Rev(OUT_C);
         Wait(50);
      }
      if (SENSOR_3)
      {
         Rev (OUT_A + OUT_C);
         Wait (100);
         Fwd(OUT_C);
         Rev(OUT_A);
         Wait(50);
      }
      Fwd(OUT_A+OUT_C);
   }
}
```

Abbildung 5: NQC LegoBot-Beispielprogramm

4 leJOS

4.1 Einführung

leJOS ist ein Ersatz für die originale Firmware der RCX-Brick. Genauer gesagt handelt es sich dabei um eine reduzierte Java Virtual Maschine, die eine Programmierung des RCX in Java wird ermöglicht. Die Verwendung einer neuen Firmware hebt viele Beschränkungen der alten Firmware auf, so unterstützt z.B. leJOS Preemtive Threads, multidimensionale Arrays, Rekursion, Gleitkommaarithmetik, Exceptions und Synchronisation. Im Gegensatz zur original Sun VM existiert keine Garbage-Collection. Beim leJOS-Compiler handelt es sich um einen statischen Linker. Da zur Laufzeit keine Klassen vom Hostrechner nachgeladen werden können, müssen alle eventuellen Klassenauf-

rufe mitkompiliert und auf den Brick geladen werden. Die Architektur illustriert die Abbildung 6.

Jedes leJOS-Programm muss das Packet „josx.platform.rcx.*“ importieren, ansonsten unterscheidet es sich nicht viel von einem normalen Java-Programm. Der Einstiegspunkt ist ebenfalls die Methode „public static void main (String [] args)“, die Kontrollstrukturen, die Schleifen, die Zuweisungen und die Vergleichsoperatoren entsprechen genau der Java-Syntax. Aus diesem Grund wird an dieser Stelle hierauf nicht weiter eingegangen sondern den Besonderheiten von leJOS Beachtung geschenkt. Eine ausführliche Beschreibung der Programmiersprache Java kann beispielsweise [Bi98] entnommen werden.

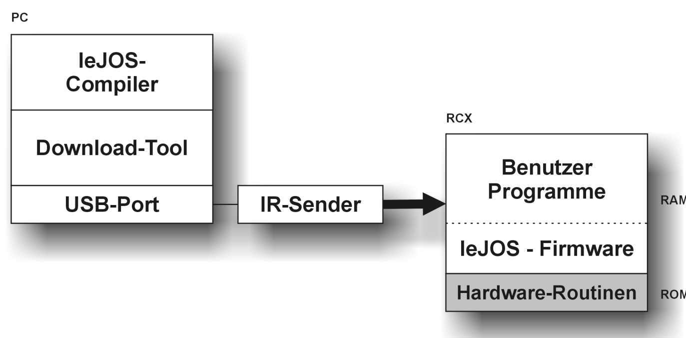


Abbildung 6: leJOS Softwarearchitektur

Weitere Informationen zur Programmierung mit leJOS lassen sich [FL02] entnehmen.

4.2 IO-Funktionen

Motoren

Die Benutzung der Motoren erfolgt durch die Klasse „Motor“. Die Motoren werden durch die statischen Mitglieder „Motor.A“, „Motor.B“, „Motor.C“ repräsentiert. Zur Steuerung der Motoren besitzt die Klasse verschiedene Methoden, die Einzelheit können der nachstehenden Tabelle entnommen werden:

Methode	Funktion	Beispiel
forward()	Aktiviert den Motor und stellt ihn auf vorwärts	Motor.A.forward();
backward()	Aktiviert den Motor und stellt ihn auf rückwärts	Motor.A.backward();
stop()	Bremst den Motor	Motor.A.stop();
flt()	Stellt den Motor auf float	Motor.A.flt();
setPower(n)	Stellt die Geschwindigkeit von 0 bis 7 ein	Motor.A.setPower(7);

Tabelle 18: NQC Funktionen für die Motorsteuerung

Sound & Display

Außer den Motoren stehen leJOS weiterhin das Soundmodul und das Display für Output zur Verfügung. Einer der sechs Systemtöne kann durch die Methode „playTone (n)“ ausgegeben werden, ansonsten lassen sich Töne mit einer beliebigen Frequenz (31 Hz bis 10 Khz) für ein bestimmtes Zeitintervall erzeugen.

Das Display lässt sich über die Methoden der Klasse „TextLCD“ steuern. Die wichtigsten Methoden sind „print(String)“ und „clear()“. Um das Wort „hello“ auf das Display zu schreiben ist folgende Anweisung nötig:

```
TextLCD.print ('hello');
```

Sensoren

Programme, die auf die Sensoren zugreifen wollen, müssen dies mit Hilfe der Klasse „Sensor“ tun. Sie besitzt ähnlich der Motorklasse drei statische Mitglieder, in diesem Fall „Sensor.S1“, „Sensor.S2“ und „Sensor.S3“. Zusätzlich werden die Sensoren durch das Array „Sensor.SENSORS[]“ repräsentiert. Damit der Brick korrekte Sensorwerte liefert, müssen die Sensoren zuerst richtig initialisiert werden. Dies geschieht mit Hilfe des Interface „SensorConstans“. Es enthält folgende Konstanten für die Sensortypen und Sensormodi:

Konstante	Sensortyp
SensorConstants.SENSOR_TYPE_TOUCH	Berührungssensor
SensorConstants.SENSOR_TYPE_LIGHT	Lichtsensor
SensorConstants.SENSOR_TYPE_ROT	Rotationssensor
SensorConstants.SENSOR_TYPE_TEMP	Temperatursensor
SensorConstants.SENSOR_TYPE_RAW	Kein Typ

Tabelle 19: leJOS Sensortyp-Konstanten

Konstante	Sensormodus
SensorConstants.SENSOR_MODE_BOOL	Boolean
SensorConstants.SENSOR_MODE_RAW	Rohwert von 0 bis 1023
SensorConstants. SENSOR_MODE_PCT	Prozent
SensorConstants.SENSOR_MODE_ANGLE	Winkel
SensorConstants.SENSOR_MODE_PULSE	Anzahl der Impulse
SensorConstants.SENSOR_MODE_EDGE	Anzahl der Impulsflanken
SensorConstants.SENSOR_MODE_DEGC	Grad Celsius

Tabelle 20: leJOS Sensormodi-Konstanten

Ein Sensor wird mit der Methode „setTypeAndMode(Sensortyp, Sensormodus)“ initialisiert. Das nachstehende Beispiel legt fest, dass Sensor.S1 ein Berührungssensor ist, der boolesche Werte zurück liefert:

```
Sensor.S1.setTypeAndMode(SensorConstants.SENSOR_TYPE_TOUCH,
     SensorConstants.SENSOR_MODE_BOOL);
```

Licht- und Rotationssensoren müssen vor der Nutzung noch aktiviert werden, dafür steht die Methode „activate()“ bereit. Das Passivieren geschieht durch den Aufruf von „passivate()“.

Nachdem dem Programm mitgeteilt wurde um welche Art Sensoren es sich handelt, liefert die Objektmethode „readValue()“ die Sensorwerte zurück. Folgender Codeausschnitt soll die Benutzung verdeutlichen:

```
int x = Sensor.S1.readValue();  // x bekommt den Wert von
                                //Sensor 1 zugewiesen
```

Dies ist eine Möglichkeit Sensorwerte zu lesen, die bessere Vorgehensweise ist aber die Implementierung des Interface „SensorListener“. Ähnlich einem Element in GUI-Element von AWT können in leJOS für einzelne Sensoren ebenfalls Listener registriert werden. Dies hat nicht nur einen besser strukturierten Quellcode zur Folge, sondern führt auch zu einer massiven Verbesserung des Laufzeitverhaltens, da kein aktives Warten (Polling) mehr nötig ist. Damit das Interface fehlerfrei implementiert werden kann, muss die Methode „public void stateChanged(Sensor s ,int old, int new)“ vorhanden sein. Der Parameter „s“ steht für die Nummer des Sensors, „old“ für den alten Wert und „new“ für den Wert der den Aufruf der Methode bewirkte. Eine Klasse, die das Interface benutzt sieht dementsprechend folgendermaßen aus:

```
class SensorClass implements SensorListener{
     public void stateChanged(Sensor s,int old,int nu)
     {
```

```
        ... // Programmcode
        }
}
```

Die Registrierung eines passenden Sensors erfolgt durch die Anweisung:

```
Sensor.S1.addSensorListener(new SensorClass());
```

4.3 Programmarchitektur:

Java und somit auch leJOS sind objektorientierte Programmiersprachen. Sie unterscheiden sich in der Art der Programmierung deutlich von den prozeduralen Sprachen wie z.B. NQC. Im Folgenden werden zwei mögliche Architekturen für Programme in leJOS aufgezeigt. Die erste Variante macht intensiven Gebrauch vom Interface SensorListener, die zweite implementierte ein weiteres Interface namens „Behavior“.

Architektur 1

Diese Programmstruktur orientiert sich stark am Interface „SensorListener“. Die Grundidee hinter diesem Konzept ist, dass für jeden Sensor eine eigene Sensorklasse existiert. Um die Multitaskingfähigkeiten des RCX voll auszuschöpfen, werden die eigentlichen Aktionen aber nicht in der „stateChanged()“ Methode ausgeführt, sondern in einem separaten Thread. Im Unterschied zur Sun VM verfügt leJOS nicht über eine Garbage-Collection, so dass die Anzahl der Threads begrenz ist. Aus diesem Grund sollte der Aktions-Thread zu Begin gestartet und sofort mit „sleep(m)“ schlafen gelegt werden. Erst bei Bedarf wird der Thread von der Sensorklasse geweckt. Kern eines solchen Programms ist eine Hauptklasse, in unserem Fall „LegoBot“. Sie ist nur für die Initialisierung der Sensoren und das Starten der Aktions-Threads zuständig. Das nachstehende Klassendiagramm verdeutlicht den Aufbau:

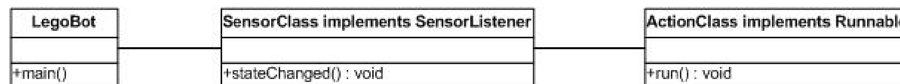


Abbildung 7: leJOS Klassendiagramm für die Architektur 1

Architektur 2

Die zweite Architektur basiert auf dem Interface „Behavior“. Klassen die diese Schnittstelle implementieren, müssen die Methoden „public void action()”,"public void suppress()” und "public boolean takeControl()” besitzen. Die Methode „takeControl()“ beschreibt, bei welchem Ereignis die Klasse aktiviert werden soll. Die Anweisungen für

die jeweiligen Aktionen sind in „public void action()"gespeichert. Falls ein Verhalten abgebrochen werden muss, wird "public void suppress()" ausgeführt. Die Verwaltung und Aktivierung der einzelnen Behaviorklassen übernimmt die Klasse „Arbitrator". Ihrem Konstruktor muss eine Array mit Behaviorklassen übergeben werden, dabei ist zu beachten dass die Reihenfolge im Array die Prioritäten mit absteigendem Rang festlegt. Zur Verdeutlichung dient das nachstehende Klassendiagramm:

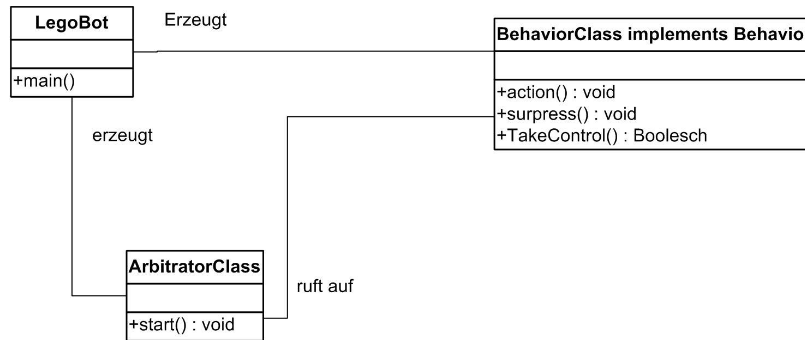


Abbildung 8: leJOS Klassendiagramm für die Architektur

4.4 Beispiel

Das folgende Beispiel implementiert die zweite Architektur.

(1)

```
import josx.robotics.*;
import josx.platform.rcx.*;
public class DriveForward implements
    Behavior {

public boolean takeControl()
      {
      return true;
      }

public void suppress()
      {
      Motor.A.stop();
      Motor.C.stop();
      }

public void action()
      {
      Motor.A.forward();
      Motor.C.forward();
      }
}
```

(2)

```
import josx.platform.rcx.*;
public class LegoBot
{
public static void main(String [] args)
      {
      Behavior b1 = new DriveForward();
      Behavior b2 = new Hit_Left();
      Behavior b3 = new Hit_Right ();
      Behavior [] bArray = {b1, b2, b3};
      Arbitrator arby = new
         Arbitrator(bArray);
      arby.start();
  }
}
```

(3)

```
import josx.robotics.*;
import josx.platform.rcx.*;

public class Hit_Left implements Behavior
{

   public boolean takeControl()
   {
      return
      Sensor.S1.readBooleanValue();
   }

   public void suppress()
      {
        Motor.A.stop();
        Motor.C.stop();
      }

   public void action()
      {
        Motor.A.backward();
        Motor.C.backward();

        try{Thread.sleep(1000);}
        catch(Exception e) {}
        Motor.A.forward();
        try{Thread.sleep(500);}
        catch(Exception e) {}
      }
}
```

Abbildung 9: leJOS LegoBot-Beispielprogramm mit Behavior- (1,3) und Arbitrator-Klasse (2)

5 brickOS

5.1 Einführung

brickOS ist eine Funktionsbibliothek für die Programmiersprache C/C++, die die Lego-Firmware des RCX vollständig ersetzt und so die Einschränkungen der Lego-Firmware besitzt umgeht. Mit brickOS ist es daher möglich, den vollen Funktionsumfang des RCX zu nutzen. Da brickOS keine eigene Programmiersprache darstellt, sondern eine Bibliothek für C/C++, ist ein brickOS-Programm ein vollwertiges C-Programm, welches auf die brickOS-Bibliotheken zurückgreift. Die Syntax von brickOS entspricht deswegen exakt der C-Syntax, weshalb auf eine Beschreibung der grundlegenden Elemente der Programmiersprache brickOS im Rahmen dieser Ausarbeitung nicht weiter eingegangen wird. Eine Einführung in die Programmiersprache C findet sich, wie oben bereits erwähnt, in [Lo98].

brickOS-Programme können mit Hilfe eines Cross-Compilers für den Hitachi H8 Prozessor des RCX kompiliert und anschießend in den RAM des RCX übertragen werden. Ingesamt stehen für die Benutzer-Programme 10 Programmslots zur Verfügung. Der Aufbau der Softwarearchitektur von brickOS ist in Abbildung 10 dargestellt.

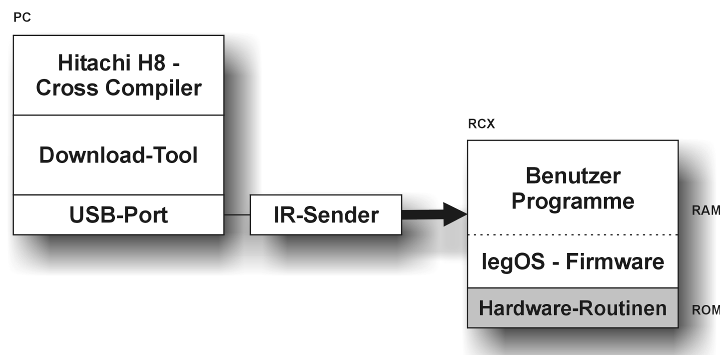


Abbildung 10: brickOS-Softwarearchitektur

Die folgende Beschreibung der Funktionalitäten von brickOS ist sehr kurz gehalten und konzentriert sich auf die wesentlichen Funktionen zur Programmierung des RCX. Weiterführende und detailliertere Angaben zu den brickOS-Bibliotheken finden sich auf den Internetseiten von [SF03] und [Ch03]. Unter dem früheren Namen legOS findet sich auch bei Knudsen [Kn99] eine gute Einführung in brickOS, die jedoch keine Beschreibung der neueren Bibliotheken, wie z. B. der LinkNetworkProtokoll-Bibliothek

bung der neueren Bibliotheken, wie z. B. der LinkNetworkProtokoll-Bibliothek (s.u.), enthält.

5.2 IO-Funktionen

Bibliotheken

brickOS stellt dem Programmierer eine Reihe von Bibliotheken zur Programmierung des RCX zur Verfügung. Die wichtigsten Bibliotheken sind in der Tabelle 21 aufgelistet. Eine genauere Beschreibung der wichtigsten Bibliotheken und deren Funktionen findet sich in den folgenden Abschnitten.

Bibliothek	Funktion	Klasse
`dmotor.h`	Motorsteuerung	Ausgabe
`dlcd.h` `conio.h`	Displaysteuerung	
`dsound.h`	Töne erzeugen	
`dsensor.h`	Auslesen der Sensoren	Eingabe
`dbutton.h` `dkey.h`	Auslesen der Knöpfe	
`battery.h`	Batteriestatus auslesen	
`lnp/lnp.h` `lnp/lnp-logical.h`	Kommunikation über IR-Schnittstelle	Sonstiges
`unistd.h`	Task-Management	
`semaphores.h`		
`persistent.h`	persistente Speicherung von Daten	
`stdlib.h`	Speicherverwaltung/ Zufallszahlen	
`rom/system.h`	Ein / Aus / Stand-by	
`time.h`	Systemzeit	
`string.h`	String-Operationen	

Tabelle 21: brickOS-Bibliotheken

Ausgabe

Motoren

Für die Motorsteuerung werden durch die Bibliothek „`dmotor.h`“ die Funktionen „`motor_*_dir`“ und „`motor_*_speed`“ bereitgestellt. Mit „`motor_*_dir`“ lassen sich die vier Modi *vorwärts, rückwärts, bremsen* und *aus* für jeden Motor einstellen. Die Leistung der einzelnen Motoren kann mit der Funktion „`motor_*_speed`“ eingestellt werden (vlg. Tabelle 22)

Funktion	Beschreibung
void **motor_a_dir**(MotorDirection dir)	Setzt die Richtung des Motors dir ∈ (fwd, rev, brake, off)
void **motor_b_dir**(MotorDirection dir)	
void **motor_c_dir**(MotorDirection dir)	
void **motor_a_speed**(unsigned char speed)	Setzt die Geschwindigkeit des Motors speed ∈ (0..255)
void **motor_b_speed**(unsigned char speed)	
void **motor_c_speed**(unsigned char speed)	

Tabelle 22: brickOS-Funktionen für die Motorsteuerung

Display

Die Displaysteuerung wird durch die Bibliotheken „conio.h“ und „lcd.h“ mit einer Vielzahl von Funktionen, die eine komfortable Programmierung des Displays ermöglichen, unterstützt. Während „conio.h“ die Darstellung ganzer Zeichenketten und Zahlen unterstützt, erlaubt „lcd.h“ das Ansprechen der einzelnen Segmente des Displays. Die folgende Tabelle zeigt einige ausgewählte Display-Funktionen.

Funktion	Beschreibung
void **cls**()	Löscht die Anzeige
void **lcd_number**(int I, number_style ns, comma_style cs)	Integer I wird im gewünschten Number/Komma-Style angezeigt
void **lcd_int**(int i)	Integer wird angezeigt (ohne Pos. 0)
void **lcd_unsigned**(int u)	Unsigned Integer wird angezeigt
void **lcd_clock**(int t)	T wird im Zeitformat XX:XX angezeigt
void **lcd_digit**(int d)	Integer wird an Pos. 0 angezeigt
void **lcd_show**(lcd_segment segment)	Einschlaten eines best. Segments
void **lcd_hide**(lcd_segment segment)	Ausschalten eines best. Segments
void **cputs**(char *s)	Ascii-String wird dargestellt
void **cputw**(unsigned word)	Zeigt eine HEX-Zahl an

Tabelle 23: brickOS-Funktionen für die Displaysteuerung

Sound

Die Sound-Bibliothek „sound.h” unterstützt das Abspielen eines Arrays von Noten mit wählbarer Tonart und Länge. Voreingestellte Töne sind „PITCH_A0...PITCH_A8“ (8 Oktaven), „PITCH_PAUSE“ steht für eine Pause und „PITCH_END“ muss am Ende des Arrays stehen. Zusätzlich ist es nach dem Abspielen der Notensequenz möglich einen Thread zu aktivieren, wodurch das Abspielen von Noten parallel zu weiteren Funktionen ausgeführt werden kann.

Funktion	Beschreibung
void **dsound_play**(const note_t *notes)	Spielt eine Notensequenz
wakeup_t **dsound_finished**(wakeup_t data)	Aktiviert einen Thread, sobald Noten-Wiedergabe beendet.

Tabelle 24: brickOS-Funktionen für das Spielen von Tönen

Eingabe

Sensoren

Die für die für das Einstellen und Auslesen der Sensoren benötigten Funktionen und Makros sind in der Bibliothek „dsensor.h“ enthalten. Die Einstellung der Sensoren (aktiv/passiv und Rotation) wird durch die Funktionen aus Tabelle 25 unterstützt. Zum Auslesen der Sensorwerte werden die in Tabelle 26 aufgelisteten Makros verwendet. Je nach Sensorart kann dabei auf unterschiedliche Makros zurückgegriffen werden.

Funktion	Beschreibung
void **ds_active**(unsigned* const sensor)	Sensoren werden auf aktiv oder passiv gesetzt
void **ds_passive**(unsigned* const sensor)	
void **ds_rotation_on**(unsigned* const sensor)	Das Zählen der Umdrehungen wird ein-/ausgeschaltet
void **ds_rotation_off**(unsigned* const sensor)	
void **ds_rotation_set**(unsigned* const sensor, int i)	Umdrehungszähler wird auf den Wert i gesetzt.

Tabelle 25: brickOS-Funktionen zum Einstellen von Sensoren

Makros	Beschreibung
int SENSOR_1, SENSOR_2, SENSOR_3	gibt den RAW-Wert des Sensors aus
boolean TOUCH_1, TOUCH_2, TOUCH_3	zeigt an, ob ein Sensor gedrückt ist
int ROTATION_1, ROTATION_2, ROTATION_3	gibt die Anzahl der Rotationen aus
int LIGHT_1, LIGHT_2, LIGHT_3	gibt den Wert eines Lichtsensors aus

Tabelle 26: brickOS-Makros zum Auslesen der Sensoren

Knöpfe

Die vier Knöpfe des Bricks können mit Hilfe der Bibliothek „dbutton.h“ angesprochen werden. Das Auslesen der Knöpfe des RCX geschieht mit den Makros aus Tabelle 28. Wahlweise kann überprüft werden, ob bestimmte Knöpfe gedrückt oder nicht gedrückt sind. Die Makros benötigen als „state“-Parameter das Ergebnis der Funktion dbutton() aus Tabelle 27 und als „button“-Parameter eine einzelne oder eine Kombination (oder-Verknüpfung) der Knopf-Konstanten BUTTON_RUN, BUTTON_PROGRAM, BUTTON_VIEW und BUTTON_ONOFF. Der Ausdruck

```
PRESSED (dbutton(), BUTTON_RUN | BUTTON_VIEW)
```

liefert das Ergebnis „`true`", falls beide Knöpfe gedrückt sind.

Funktion	Beschreibung
`int dbutton()`	Gibt den Status der Knöpfe zurück

Tabelle 27: brickOS-Funktion zum Initialisieren der Knöpfe

Makros	Beschreibung
`PRESSED(state, button)`	true, wenn alle angegebenen Knöpfe gedrückt sind
`RELEASED(state, button)`	true, wenn mindestens einer der angegebenen Knöpfe nicht gedrückt wird.

Tabelle 28: brickOS-Funktionen zum Auslesen der Knöpfe

Sonstiges

Infrarot-Port

Die Kommunikation über die IR-Schnittstelle wird durch das Link Network Protokoll (LNP) unterstützt. Das Protokoll definiert zwei Schichten, die Integrity-Layer und die Addressing-Layer. Während die Integrity-Layer nur sicherstellt, dass empfangene Pakete auch fehlerfrei sind, ermöglicht die Addressing-Layer zusätzlich noch die Adressierung der Pakete durch Host- und Portadressen. Beide Schichten garantieren jedoch nicht, dass versandte Pakete auch ankommen. Das Eintreffen neuer Pakete wird mit Handler-Funktionen überwacht, die zuvor registriert werden müssen. Einige wichtige Funktionen aus den LNP-Bibliothek („`lnp.h`") sind in der folgenden Tabelle aufgeführt.

Funktion	Beschreibung
`void lnp_set_hostaddr (unsigned char host)`	Definiert die Adresse des RCX
`void lnp_integrity_set_handler (lnp_integrity_handler_t handler)`	Definiert einen Integrity-Layer-Handler für ankommende Pakete
`void lnp_addressing_set_handler (unsigned char port, lnp_addressing_handler_t handler)`	Definiert einen Addressing-Layer-Handler für Pakete, die an einem best. Port ankommen
`int lnp_integrity_write (const unsigned char *data, unsigned char length)`	Sendet length Bytes aus dem data-Puffer über die Integrity-Layer
`int lnp_addressing_write (const unsigned char *data, unsigned char length, unsigned char dest, unsigned char srcport)`	Sendet length Bytes aus dem data-Puffer an den Port srcport des Hosts dest über die Addressing-Layer

Tabelle 29: brickOS-Funktionen Timer

Multitasking

Die Bibliothek „unistd.h" enthält Funktionen, die preemptives Multitasking unterstützen. Neben dem Starten und Löschen von Tasks gibt es die Möglichkeit Tasks für einen bestimmten Zeitraum oder bis zu einem gewünschten Ereignis zu unterbrechen. Einige wichtige Funktionen des Taskmanagements sind in der Tabelle 30 dargestellt. Zusätzlich besteht die Möglichkeit mit Hilfe der Bibliothek „semaphores.h" die Zugriffe mehrerer Tasks auf Datenelemente zu synchronisieren. Auf eine Beschreibung der Semaphoren-Bibliothek wird an dieser Stelle jedoch verzichtet. Weitere Angaben hierzu finden sich auf der brickOS-Homepage [SF03].

Funktion	**Beschreibung**
`tid_t` **`execi`**`(int (*code_start)(int, char**), int argc, char **argv, priority_t priority, size_t stack_size)`	registriert die übergebene Task. argc= Anzahl Parameter, argv=Parameter, priority=Priorität des Tasks (1..20), stack_size=ben. freier Stack
`void` **`tm_start`**`()`	Startet den Taskmanager
`void` **`kill`**`(tid_t tid)`	Löscht die übergebene Task
`unsigned int` **`sleep`**`(unsigned int sec)`	Ausführung eines Tasks wird für die übergebene Zeit angehalten.
`unsigned int` **`msleep`**`(unsigned int msec)`	
`wakeup_t` **`wait_event`**`(wakeup_t(*wakeup) (wakeup_t), wakeup_t data)`	Unterbricht die Ausführung bis die wakeup-Funktion einen Wert >0 zurückgibt.

Tabelle 30: brickOS-Funktionen für das Task-Management

Memory

Um größere Mengen von Daten zu speichern stellt die Bibliothek „stdlib.h" Funktionen zur Verfügung (vgl. Tabelle 31), mit denen größere Speicherblöcke alloziert und freigegeben werden können. Die Bibliothek „persistent.h" erlaubt darüber hinaus die Definition persistenter Variablen, deren Werte über mehrere Programmdurchläufe im Speicher erhalten bleiben. Beispiel:

```
int counter __persistent = 0;
```

Funktion	**Beschreibung**
`void` **`*calloc`**`(size_t nmemb, size_t size)`	Alloziert Speicher für ein Array mit `nmemb` Elementen der Größe `size`
`void` **`*malloc`**`(size_t size)`	Alloziert einen Speichetblock der größe `size`
`void` **`free`**`(void *ptr)`	Löscht die übergebene Task

Tabelle 31: brickOS-Funktionen zum Allozieren von Speicherblöcken

Weitere

Funktionen, wie das Erzeugen von Zufallszahlen, das Power-Management sowie Zeit- und Stringoperationen die durch weitere Bibliotheken (vgl. Tabelle 21) bereitgestellt werden, sollen an dieser Stelle nicht näher behandelt werden.

5.3 Beispiel

Das brickOS Beispiel steuert den LegoBot mit der Hilfe von Tasks, die parallel zueinander laufen. Im ersten Block werden die benötigten Bibliotheken importiert. Der zweite Block definiert benötigte Konstanten, sowie ein Array für das Taskmanagement. Für die Bewegung des Bots sind die Tasks der Blöcke drei und vier zuständig. Jede der beiden Tasks schreibt die gewünschte Bewegungsrichtung in eine zugehörige globale Variable (`cruise_command` und `wall_command`). Die Arbitrator-Task (Schiedsrichter-Task) überwacht beide Variablen und leitet die Kommandos abhängig von der Priorität (`wall`- vor `cruise_comman`) der Bewegungstasks an die Funktion motor_command (8) weiter. motor_command wandelt die Kommandos in Motorsteuerungen um. Block 6 und 7 enthalten Hilfsfunktionen zum registrieren und stoppen der Tasks. Im Hauptprogramm werden die Tasks registriert und anschließend das Programm durch das Starten des Taskmanagers gestartet.

Das hier verwendete Programmkonzept, welches die LegoBot-Verhaltensweise in einzelne logische Einheiten unterteilt und diese jeweils durch eine Task realisiert, zeichnet sich durch gute Manipulierbarkeit und Veränderbarkeit aus und lässt sich leicht auf beliebige Robotersteuerungen übertragen.

```
#include "dmotor.h"                                  (1)
#include "dsensor.h"
#include "unistd.h"
#include "sys/tm.h"

#define MAX_TASKS 32                                 (2)
tid_t tid[MAX_TASKS];
int task_index;

#define BACK_TIME 500
#define TURN_TIME 800

// Motor commands
#define COMMAND_NONE -1
#define COMMAND_FORWARD 1
#define COMMAND_REVERSE 2
#define COMMAND_LEFT 3
#define COMMAND_RIGHT 4
#define COMMAND_STOP 5

int cruise_command;                                  (3)
int cruise(int argc, char **argv) {
  cruise_command = COMMAND_FORWARD;
  while (1) sleep(1);
  return 0;
}

int wall_command;                                    (4)
int wall(int argc, char **argv) {
  wall_command = COMMAND_NONE;
  while(1) {
    if (SENSOR_1 < 0xf000) {
      wall_command = COMMAND_REVERSE;
      msleep(BACK_TIME);
      wall_command = COMMAND_RIGHT;
      msleep(TURN_TIME);
      wall_command = COMMAND_NONE;
    }
    if (SENSOR_3 < 0xf000) {
      wall_command = COMMAND_REVERSE;
      msleep(BACK_TIME);
      wall_command = COMMAND_LEFT;
      msleep(TURN_TIME);
      wall_command = COMMAND_NONE;
    }
  }
  return 0;
}

int arbitrate(int argc, char **argv) {               (5)
  while(1) {
    if (cruise_command != COMMAND_NONE)
      motor_command = cruise_command;
    if (wall_command != COMMAND_NONE)
      motor_command = wall_command;
    motor_control();
  }
}

int stop_task(int argc, char **argv) {               (6)
  int i;
    msleep(200);
  while (!PRESSED(dbutton(), BUTTON_RUN))
    ;
  for (i=0; i < task_index; i++)
    kill(tid[i]);
  return 0;
}

void exec_helper(int                                 (7)
(*code_start)(int,char**)) {
  tid[task_index++] = execi(code_start,
0, NULL, 0, DEFAULT_STACK_SIZE);
}

int motor_command;                                   (8)
void motor_control() {
  motor_a_speed(MAX_SPEED);
  motor_c_speed(MAX_SPEED);

  switch (motor_command) {
    case COMMAND_FORWARD:
      motor_a_dir(fwd);
      motor_b_dir(fwd);
      break;
    case COMMAND_REVERSE:
      motor_a_dir(rev);
      motor_b_dir(rev);
      break;
    case COMMAND_LEFT:
      motor_a_dir(rev);
      motor_b_dir(fwd);
      break;
    case COMMAND_RIGHT:
      motor_a_dir(fwd);
      motor_b_dir(rev);
      break;
    case COMMAND_STOP:
      motor_a_dir(brake);
      motor_b_dir(brake);
      break;
  }
}

int main() {                                         (9)
  task_index = 0;

  execi_helper(&avoid);
  execi_helper(&cruise);
  execi_helper(&arbitrate);
  execi(&stop_task, 0,NULL, 0,
        DEFAULT_STACK_SIZE);

  tm_start();
  return 0;
}
```

Abbildung 11: legOS LegoBot-Beispielprogramm

6 pbFORTH

6.1 Einführung

Die Programmiersprache pbFORTH (Programable Brick FORTH) basiert auf der in den 60er Jahren von Charles Moore entwickelte Sprache FORTH. FORTH wurde zunächst

ausschließlich zur Steuerung von Teleskopen genutzt. Aufgrund der geringen Speicheranforderungen und der relativ einfachen Struktur der Sprache ist sie besonders für den Einsatz auf kleineren Rechnersystemen geeignet.

Zentraler Bestandteil von FORTH ist der Datenstack, über den sämtliche Daten zwischen Benutzer und Funktionen und zwischen Funktionen ausgetauscht werden. Eine weitere Besonderheit, die pbFORTH von den anderen hier vorgestellten Sprachen unterscheidet, ist die Tatsache, dass FORTH eine Interpreter-Sprache ist. Die Befehle werden also nicht als vollständiges Programm kompiliert und anschließend auf den Brick übertragen, sondern einzeln mit einem Terminalprogramm an den pbFORTH-Interpreter, der auf dem RCX installiert ist, gesendet. Der Interpreter verarbeitet alle empfangenen Befehle sofort und sendet das Ergebnis der Bearbeitung an das Terminalprogramm zurück. Der Entwicklungsprozess unterscheidet sich daher deutlich von den Compiler-Sprachen und ist besonders für Einsteiger gewöhnungsbedürftig.

Die wesentlichen Bestandteile der Softwarearchitektur von pbFORTH sind ein Terminalprogramm auf dem PC und der pbFORTH-Interpreter, der die Lego-Firmware ersetzt (vgl. Abbildung 12).

Eine vollständige Beschreibung der Sprache pbFORTH mit Beispielprogrammen befindet sich auf den Internetseiten von Hempel, dem Entwicklers von pbFORTH (vgl. [He03]). Für Einsteiger eignet sich das Buch [Kn99] von Knudsen.

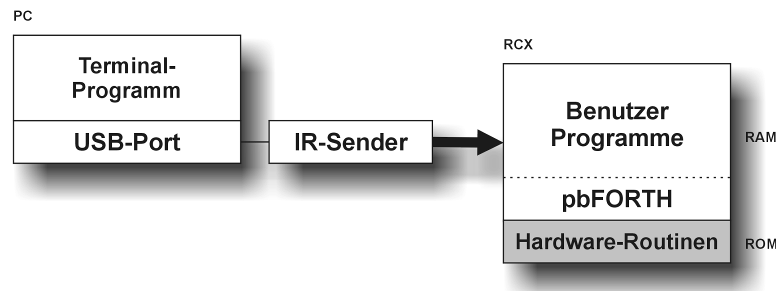


Abbildung 12: Die Softwarearchitektur von pbFORTH

6.2 Grundlegende Elemente

Dieser Abschnitt gibt einen Überblick über die grundlegenden Elemente der Programmiersprache FORTH. Dabei wird auf die Kommunikation mit dem pbFORTH-Interpreter, die Konzepte des Datenstacks und des Wörterbuchs, den Umgang mit Variablen und Konstanten sowie auf die vorhandenen Kontrollstrukturen eingegangen.

Kommunikation mit pbFORTH

Wie bereits erwähnt, handelt es sich bei pbFORTH um eine Interpretersprache. Für die Programmierung wird deshalb ein Terminalprogramm benötigt, das über die USB-Schnittstelle und den angeschlossenen IR-Tower mit dem pbFORTH-Interpreter, der auf dem RCX installiert ist, kommuniziert. Jeder empfangene Befehl wird, falls er korrekt ist, sofort ausgeführt und anschießend mit der Ausgabe „ok" bestätigt. Um beispielsweise den Motor 0 zu starten sendet man „`7 2 0 MOTOR_SET`" an den Interpreter. Nach dem Drücken der Enter-Taste wird der Motor gestartet und „`ok`" als Bestätigung ausgegeben. Zur Erleichterung der Kommunikation mit pbFORTH bieten die gängigen Terminalprogramme auch die Möglichkeit komplette Programm-Skripte (Dateien) an den Interpreter zu übertragen.

Der Datenstack

Der von pbFORTH für den Datenaustausch verwendete Datenstack besitzt die Funktionalität eines herkömmlichen Stacks. Er unterstützt zwei Funktionen, die Push-Operation, um ein Element an oberster Stelle auf den Stack zu legen, und die Pop-Operation, um das oberste Element aus dem Stack zu entnehmen. Jede Eingabe über das Terminalprogramm stellt eine Pusch-Operation dar. Nach dem Drücken der Enter-Taste werden die eingegebenen Zeichen in Elemente zerlegt, die der Reihenfolge nach auf den Stack gelegt werden. Elemente sind durch Leerzeichen getrennte Zeichenketten, die keine Befehle darstellen. Die Pop-Operation wird durch den Befehl „`.`" ausgeführt. Die Eingabe

```
27 55 13
ok
```

legt die Elemente 27, 55 und 13 auf den Stack. Jetzt können Pop-Operationen durchgeführt werden:

```
.
13 ok
..
27 55 ok
```

Da Befehle und Operatoren ebenfalls für die Ein- und Ausgabe auf den Datenstack zugreifen, verwendet pbFORTH die Postfix-Notation. Die für einen Befehl benötigten Daten müssen vorher auf den Stack gelegt werden. Das Ergebnis wird ebenfalls auf den Stack gelegt.

Wörter und das Wörterbuch

In pbFORTH entspricht ein Wort (word) einem Programm mit einem Namen. Wörter und ihre Bedeutung werden im Wörterbuch (Dictionary) gespeichert. FORTH besitzt eine Reihe von voreingestellten Wörtern, z.B. zur Manipulation des Datenstacks (vgl. Tabelle 32) und für arithmetische Operationen (vgl. Tabelle 33).

Wort	Bedeutung	Stack vorher	Stack nachher
DUP	kopiert 1. Element des Stacks	x	x x
OVER	kopiert 2. Element des Stacks	x2 x1	x2 x1 x2
PICK	kopiert n. Element des Stacks	xn ... n	xn ... xn
SWAP	vertauscht 1. u. 2. Element des Stacks	x2 x1	x1 x2
ROT	schiebt 3. Element an 1. Position	x3 x2 x1	x2 x1 x3
DROP	entfernt 1. Element vom Stack	x	

Tabelle 32: pdFORTH-Wörter für Stack-Operationen

Wort	Bedeutung	Stack vorher	Stack nachher
+	addiert 1. und 2. Element	x2 x1	<x2 + x1>
-	subtrahiert 1. und 2. Element	x2 x1	<x2 - x1>
*	multipliziert 1. und 2. Element	x2 x1	<x2 * x1>
/	dividiert 1. und 2. Element	x2 x1	<x2 / x1>
AND	bitweises AND von 1. und 2. Element	x2 x1	<x2 AND x1>
OR	bitweises OR von 1. und 2. Element	x2 x1	<x2 OR x1>
XOR	bitweises XOR von 1. und 2. Element	x2 x1	<x2 XOR x1>

Tabelle 33: pbFORTH-Wörter für arithmetische Operationen

Neue Wörter definieren:

Es besteht auch die Möglichkeit, das Wörterbuch um neue Wörter zu erweitern. Neue Wörter können auf bereits definierte Wörter zurückgreifen, wodurch sich ausgehend von den vordefinierten Wörtern zunehmend komplexere Wörter erstellen lassen. Die Definition eines neuen Wortes geschieht folgendermaßen:

```
: <neues Wort> (<definiertes Wort> | <Element>)*;
```

Beispiel:

```
: dreimal DUP DUP + + ;
ok
7 dreimal .
21 ok
```

Konstanten und Variablen

Die Definition von Konstanten und Variablem wird in der folgenden Tabelle veranschaulicht. Das Auslesen einer Variable geschieht in zwei Schritten, zunächst wird mit

dem „@“-Operator der Wert der Variablen auf den Stack gelegt und anschießend mit dem Pop-Operator („.“) der oberste Wer des Stacks ausgegeben.

Ausdruck	Bedeutung	Beispiel
<Wert> **CONSTANT** <Name>	Definiert die Konstante <Name> mir Wert <Wert>	7 **CONSTANT** MAX_SPEED MAX_SPEED 2 0 **MOTOR_SET**
VARIABLE <Name>	Deklariert die Variable <Name>	VARIABLE z
<Wert> <Var-Name> **!**	<Wert> wird Variable <Var-Name> zugewiesen	12 z **!**
<Var-Name> **@** .	Variable <Var-Name> wird ausgelesen	z **@** . 12 ok

Tabelle 34: pbFORTH Definition von Konstanten und Variablen

Kontrollstrukturen

Zur Steuerung des Programmablaufs stellt pbFORTH neben der if-Bedingung die Loop- und die Repeat-Schleife, zur Verfügung. Die für die Kontrollstrukturen benötigten relationalen Operatoren können der Tabelle 35 entnommen werden.

IF-Bedingung

```
<Bool-Wert> IF <True-Teil> THEN
<Bool-Wert> IF <True-Teil> ELSE <False-Teil> THEN
```

Der oberste wert des Stacks wird als Bool-Wert interpretiert und der entsprechende Teil der Bedingung, falls vorhanden, ausgeführt.

LOOP-Schleife

```
<Ende> <Anfang> DO <Schleifen-Teil> LOOP
<Ende> <Anfang> DO <Wörter> <Inkrement> +LOOP
```

Der Schleifen-Teil wird sooft durchlaufen, bis der der interne Zähler I, der zu Beginn auf <Anfang> gesetzt wird und bei jedem Durchlauf um eins erhöht wird, $\geq$ <Ende> ist. Die 2. Variante der Schleife erhöht den internen Zähler nicht um 1, sondern um <Inkrement>. Auf I kann innerhalb der Schleife zugegriffen werden. Beispiel:

```
: einsBisZehn 11 1 DO I . LOOP;
ok
einsBisZehn
1 2 3 4 5 6 7 8 9 10 ok
```

Schleife mit Abbruchbedingung

```
BEGIN <Schleifen-Teil> UNTIL
```

Der Schleifen-Teil wird durchlaufen, bis oberster Wert des Stacks „true“ ist. Beispiel:

```
: warteAufButton2
  BEGIN
    RCX_BUTTON DUP
    BUTTON_GET @ 2 AND
  UNTIL;
```

In der Warteschleife wird die Variable RCX_BUTTON auf den Stack gelegt und verdoppelt. BUTTON_GET speichert den Status der Buttons in der Variable RCX_BUTTON (der Stack enthält jetzt nur noch einmal RCX_BUTTON) und anschließend ließt „@“den Wert von RCX_BUTTON aus. Nun wird „2“ auf den Stack gelegt und bitweise mit dem Wert von RCX-BUTTON verknüpft (and). Falls der Knopf 2 gedrückt ist, liefert die Verknüpfung einen Wert größer 0, der als true interpretiert wird und die Schleife wird beendet.

Wort	Beispiel
<	1. und 2. Element des Stacks ersetzt durch true, falls 2.< 1., sonst false
=	1. und 2. Element des Stacks ersetzt durch true, falls 2.= 1., sonst false
>	1. und 2. Element des Stacks ersetzt durch true, falls 2.> 1., sonst false
`0<`	1. Element des Stacks ersetzt durch true, falls 1. < 0, sonst false
`0=`	1. Element des Stacks ersetzt durch true, falls 1. = 0, sonst false

Tabelle 35: pbFORTH relationale Operatoren

6.3 IO-Funktionen

Ausgabe

Motoren

Die Funktion MOTOR_SET zur Steuerung der Motoren entnimmt dem Stack den „modus“ (0 ~ forward, 1 ~ reverse, 2 ~ brake, 3 ~ float), die „power“ (1 ~ minimale … 7 ~ maximale Leistung) und die Nummer des Motors („index“ 0..2), um den entsprechenden Motor einzustellen.

Wort	Bedeutung	Stack vorher	Stack nachher
`MOTOR_SET`	aktiviert den Motor <index> mit Leistung <power> und Modus <modus>	`<power> <modus> <index>`	

Tabelle 36: pbFORTH-Wort für die Motorsteuerung

Display

pbFORTH ermöglicht sowohl das darstellen von Zahlen, als auch die Steuerung einzelnen Segmente des Displays (vgl. Tabelle 37)

Wort	Bedeutung	Stack vorher	Stack nachher
`LCD_REFRESH`	Änderungen am Display werden sichtbar		
`LCD_NUMBER`	Stellt eine Dezimalzahl auf dem Display dar.	`<komma> <wert> <sign>`	
`LCD_SHOW`	aktiviert das Segment <segment>	`<segment>`	
`LCD_HIDE`	deaktiviert das Segment <segment>	`<segment>`	
`LCD_CLEAR`	deaktiviert alle Segmente		

Tabelle 37: pbFORTH-Wörter für die Displaysteuerung

Sounds

Die Soundausgabe unterstützt das Abspielen von System-Tönen und das Erzeugen von Tönen beliebiger Dauer und Frequenz.

Wort	Bedeutung	Stack vorher	Stack nachher
`SOUND_PLAY`	Spielt einen System-Sound (0..6)	`<code>`	
`SOUND_TONE`	Spielt einen Ton mit der Dauer <time> (1/100 sec.) und der Frequenz <freq>	`<time> <freq>`	

Tabelle 38: pbFORTH Wörter für das Abspielen von Sounds

Eingabe

Sensoren

Mit den Wörtern SENSOR_ACTIVE, SENSOR_PASSIVE, SENSOR_TYPE und SENSOR_MODE können die Sensoren Konfiguriert werden. Die möglichen Typen und Modi können der Tabelle 40 und der Tabelle 41 entnommen werden. Das Auslesen der Sensoren geschieht immer in zwei Schritten. Zunächst wird der Wert des Sensors mit SENSOR_READ aktualisiert und anschließend mit SENSOR_VALUE im entsprechenden Modus ausgegeben. Die Funktionen SENSOR_RAW und SENSOR_BOOL geben den Wert unabhängig von Modus aus.

Wort	Bedeutung	Stack vorher	Stack nachher
SENSOR_ACTIVE	Sensor **<index>** wird mit Strom vesorgt.	<index>	
SENSOR_PASSIVE	Sensor **<index>** wird **nicht** mit Strom vesorgt	<index>	
SENSOR_TYPE	Definiert den Typ **<typ>** des Sensors **<index>**	<typ> <index>	
SENSOR_MODE	Definiert den Typ **<modus>** des Sensors **<index>**	<modus> <index>	
SENSOR_READ	Sensors **<index>** wird ausge-lesen	<index>	<status>
SENSOR_VALUE	Wert des Sensors **<index>** ausgegeben (abh. vom Modus)	<index>	<wert>
SENSOR_RAW	Wert des Sensors **<index>** ausgegeben (im RAW-Modus)	<index>	<wert>
SENSOR_BOOL	Wert des Sensors **<index>** ausgegeben (im BOOL-Modus)	<index>	<wert>

Tabelle 39: pbFORTH-Worte zum Einstellen und Auslesen der Sensoren

<typ>	Beschreibung
0	Raw
1	Touch
2	Temperature
3	Light
4	Rotation

Tabelle 40: pbFORTH Sensortypen

<modus>	Beschreibung
0	Raw
20	Boolean
40	Edge counting
60	Pulse counting
80	Percent
A0	Celsius
C0	Fahrenheit
E0	Rotation

Tabelle 41: pbFORTH Sensormodi

Knöpfe

Die Funktion BUTTON_GET schreibt den Knopf-Status als bitweise OR-Verknüpfung der Knopfwerte (vgl. Tabelle 43) in die auf dem Stack liegende Variable. Sind z. B. die Knöpfe „Run“ und „Prgm“ gedrückt wird der Wert 5 ausgelesen. Der Status des „On/Off“-Knopfes wird mit `POWER_GET` abgefragt (s.u.).

Wort	Bedeutung	Stack vorher	Stack nachher
BUTTON_GET	schreibt Knopf-Status in Variable <var>	<var>	

Tabelle 42: pbFORTH-Wort für das Initialisieren und Auslesen der Knöpfe

Knopf	Wert
Run	1
View	2
Prgm	4

Tabelle 43: pbFORTH Werte der Knöpfe

Batterie

Der Batteriestatus kann mit POWER_GET in einer Variablen (es steht die Variable RCX_POWER zur Verfügung) gespeichert werden. Es besteht zudem die Möglichkeit, den RCX auszuschalten.

Wort	Bedeutung	Stack vorher	Stack nachher
RCX_POWER	Variable zur Speicherung des POWER_GET-Resultats		<var>
POWER_GET	schreibt Batterie-Status (<code>= 0x4001) oder OnOff-Knopfstatus (<code>=0x4000) in die Variable <var>	<var> <code>	
POWER_OFF	Schaltet RCX aus, bis zum Drücken des OnOff-Knopfes		

Tabelle 44: pbFORTH-Wort für das Auslesen des Batteiestatus

Sonstiges

Infrarot-Port

Für die Datenübertragung über den Infrarotport stellt pbFORTH Funktionen zur Verfügung, die sowohl das Senden, als auch das Empfangen von einzelnen Bytes unterstützen (vgl. Tabelle 45). Im Vergleich zu anderen Sprachen, die komplexe Protokolle für die IR-Kommunikation bereitstellen, ist der Funktionsumfang von pbFORTH an dieser Stelle sehr eingeschränkt. Protokolle für den Datenaustausch auf einer höheren Abstraktionsebene müssen vom Programmierer selbst implementiert werden.

Wort	Bedeutung	Stack vorher	Stack nachher
EKEY	liest Byte-Wert vom IR-Puffer, blockiert bis erfolgreich gelesen		<Wert>
EKEY?	gibt true zurück, falls CHAR-Wert im IR-Puffer, sonst false		
EMIT	sendet Byte-Wert	<Wert>	

Tabelle 45: pbFORTH-Wörter für die IR-Kommunikation

Timer

pbFORTH stellt für beide Timer-Typen, die zehntel-Sekunden-Timer und die hundertstel-Sekunden-Timer, Wörter zum Einstellen und Auslesen bereit. die zehntel-Sekunden-Timer erhöhen ihren Wert jede zehntel Sekunde um 1 (bis 32767), während die hundertstel Sekunden Timer ihren Wert jede hundertstel um 1 reduzieren bis 0.

Wort	Bedeutung	Stack vorher	Stack nachher
TIMER_SET	setzt den 1/10 sek.-Timer <index> (0-3) auf <value>	<value> <index>	<Wert>
timer_SET	setzt den 1/100 sek. - Timer <index> (0-9) auf <value>		
TIMER_GET	liest den Wert des 1/10 sek. -Timers <index> (0-3) aus	<Wert>	
timer_GET	liest den Wert des 1/100 sek. -Timers <index> (0-3) aus		

Tabelle 46: pbFORTH-Wörter für die Timer-Kontrolle

Multitasking

Es ist zwar möglich in pbFORTH kooperatives Multitasking zu implementieren, hierfür werden aber in der aktuellen Version keine Wörter bereitgestellt.

6.4 Beispiel

Das Legobot-Beispiel in pbFORTH lässt sich in sieben Blöcke unterteilen, deren Funktionalität im Folgenden kurz erläutert wird. Im ersten Block wird der Hex-Modus für alle Zahlenwerte eingestellt und benötigte Konstanten und Variablen für Motoren, Sensoren, Knöpfe und Timer werden definiert. Der zweite Block definiert das Wort `initialize` zum Einstellen der beiden Berührungs-Sensoren. Anschließend werden im Block vier die Wörter für die fünf Bewegungsrichtungen definiert. Das Wort `wait` des fünften Blocks unterbricht die Ausführung für die Dauer von „obersten Stackwertes in hundertstel Sekunden“. Das Wort move (fünfter Block) bewegt den LegoBot. Solange keiner der Sensoren gedrückt ist, bewegt sich der LegoBot nach vorne. Sollte einer der Sensoren gedrückt werden, wird ein entsprechendes Ausweichmanöver gestartet. Der sechste Block enthält Wörter zum Auslesen des Run-Knopfes. Das Hauptprogramm bzw. das Wort start im letzten Block initialisiert die Sensoren, wartet bis der Run-Knopf gedrückt und wieder losgelassen wird und ruft anschließend solange das Wort move auf, bis der Run-Knopf abermals gedrückt wurde. Das Beispiel soll verdeutlichen, dass

das Programmieren mit pbFORTH im Prinzip darauf beruht, dass aus einfachen Wörtern (Block 3) immer komplexere Wörter (Block 7) definiert werden.

```
(1)
HEX
\ Constanten
\ Sensoren
0 CONSTANT S1
1 CONSTANT S2
2 CONSTANT S3

1 CONSTANT TOUCH
20 CONSTANT BOOL

\ Motoren
0 CONSTANT M1
1 CONSTANT M2
2 CONSTANT M3
1 CONSTANT FWD
2 CONSTANT REW
4 CONSTANT FLW
7 CONSTANT MAX_SPEED

A CONSTANT REW_TIME
5 CONSTANT TURN_TIME

\ Buttons
1 CONSTANT RUN

\ Variablen
VARIABLE timer
```

```
(2)
: initialize
  TOUCH S1 SENSOR_TYPE
  BOOL S1 SENSOR_MODE

  TOUCH S3 SENSOR_TYPE
  BOOL S3 SENSOR_MODE
;
```

```
(3)
: forward
  MAX_SPEED FWD M1 MOTOR_SET
  MAX_SPEED FWD M3 MOTOR_SET
;

: rewind
  MAX_SPEED REW M1 MOTOR_SET
  MAX_SPEED REW M3 MOTOR_SET
;

: stop
  0 FLW M1 MOTOR_SET
  0 FLW M3 MOTOR_SET
;

: turnRight
  MAX_SPEED FWD M1 MOTOR_SET
  MAX_SPEED REW M3 MOTOR_SET
;
```

```
(3)
: turnLeft
  MAX_SPEED REW M1 MOTOR_SET
  MAX_SPEED FWD M3 MOTOR_SET
;
```

```
(4)
: wait
  timer !
  0 0 TIMER_SET
  BEGIN
    0 TIMER_GET timer @ =
  UNTIL
;
```

```
(5)
: move
  forward
  S1 SENSOR_READ DROP
  S1 SENSOR_VALUE
  1 = IF
    rewind
    REW_TIME wait
    turnRight
    TURN_TIME wait
  THEN

  S3 SENSOR_READ DROP
  S3 SENSOR_VALUE
  1 = IF
    rewind
    REW_TIME wait
    turnLeft
    TURN_TIME wait
  THEN
;
```

```
(6)
: buttonState
  RCX_BUTTON DUP BUTTON_GET @
;
: isRunButtonPressed
  buttonState RUN AND
;
: isRunButtonNotPressed
  buttonState RUN XOR
;
```

```
(7)
: start
  initialize
  BEGIN
    isRunButtonPressed
  UNTIL
  BEGIN
    isRunButtonNotPressed
  UNTIL
  BEGIN
    move
    isRunButtonPressed
  UNTIL
  stop
;
```

Abbildung 13: pbFORTH LegoBot-Beispielprogramm

7 Industrieroboter

7.1 Einführung

Ziel dieses Kapitels ist es, dem Leser einen Einblick in die Programmierung moderner Industrieroboter zu geben. Dies ist besonders im Hinblick auf die gravierenden Unterschiede interessant. Zuvor soll aber die Frage, was ein Industrieroboter eigentlich ist, eingegangen werden. Die VDI-Richtlinie 2860 bietet eine gute Definition:

„Industrieroboter sind universell einsetzbare Bewegungsautomaten mit mehreren Achsen, deren Bewegung hinsichtlich Bewegungsfolge und Wegen bzw. Winkeln frei programmierbar (d.h. ohne mechanischen Eingriff vorzugeben bzw. änderbar) und gegebenenfalls sensorgeführt sind. Sie sind mit Greifern, Werkzeugen oder anderen Fertigungsmitteln ausrüstbar.“ (vgl. [VD90])

Diese Definition ist recht allgemein gehalten, verdeutlich aber trotzdem die beiden wichtigsten Aspekte. Zum einen, dass die Bewegungen eines Roboters in den konstruktionsbedingten Grenzen frei programmierbar sind und, dass er mit einem Effektor (Werkzeug, Greifer etc.) ausgestattet ist. Diese Informationen reichen aber noch nicht aus, um einen konkreten Roboter zu beschreiben, es fehlen Informationen über die Freiheitsgrade, die Positionierungsgenauigkeit, die Wiederholgenauigkeit, die Bahntreue, die Payload und die Verfahrengeschwindigkeit. Eine Erörterung der genannten Fachbegriffe folgt.

Freiheitsgrade

Unter dem Begriff „Freiheitsgrade“ wird verstanden, wie ein Roboter seinen Effektor im dreidimensionalen Raum positionieren kann. Kann das Werkzeug nur auf einer Achse, z.B. der X-Achse verschoben werden, hat der Roboter den Freiheitsgrad eins. Für jede weitere Achse die für Verschiebungen zur Verfügung steht, erhöht sich der Freiheitsgrad um eins. Der Wert wird für jede Achse, um die der Effektor rotieren kann ebenfalls erhöht. Es sind auf diese Weise bis zu sechs Freiheitsgrade möglich. Nur Roboter mit sechs Freiheitsgeraden können auch wirklich jeden Punkt im Raum, von allen Seiten erreichen. Ein Großteil der Aufgaben lässt sich aber auch mit Freiheitsgraden von drei bis vier erledigen. (vgl. [Sc02])

Positionierungsgenauigkeit

Die nächste charakteristische Eigenschaft eines Roboters ist die Positionierungsgenauigkeit Sie beschreibt, wie genau der Roboter einen Punkt ansteuern kann. Bei modernen Maschinen liegt sie im Bereich von zehntel Millimetern.

Wiederholgenauigkeit

Die Wiederholgenauigkeit spezifiziert den Bereich um einen Punkt, der bei mehrmaliger Wiederholung garantiert getroffen wird. Dieser Wert liegt normalerweise in derselben Größenordnung wie die Positionierungsgenauigkeit.

Payload

Das maximale Gewicht, das ein Effektor haben darf, ohne dass die Wiederholgenauigkeit und die Positionierungsgenauigkeit beeinträchtigt werden, wird als Payload bezeichnet.

Bahntreue

Je genauer der Roboter den Effektor entlang einer vorprogrammierten Bahn führen kann, desto besser ist der Wert für die Bahntreue. Sie wird durch die Positionierungsgenauigkeit und die Wiederholungsgenauigkeit stark beeinflusst.

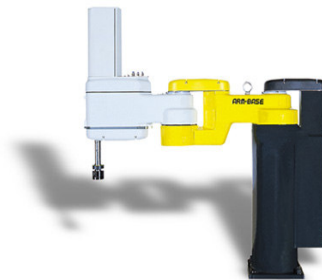
Verfahrensgeschwindigkeit

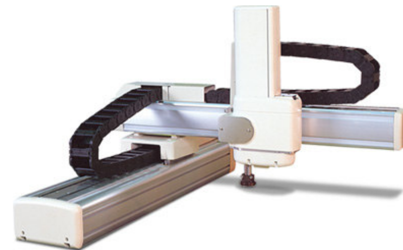
Die Geschwindigkeit mit welcher die Roboter arbeiten, ist im industriellen Einsatz ein wichtiges Kriterium, sie wird durch den Parameter Verfahrensgeschwindigkeit angegeben.

Robotertypen

Für verschiedene Tätigkeiten müssen dementsprechend passend parametrisierte Roboter gewählt werden, ein Roboter zum Schrauben sortieren braucht z.B. keine Payload von drei Tonnen, andererseits braucht ein Hochregalroboter keinen Freiheitsgrad von sechs. All diese Überlegungen haben in der Industrie zu drei Standardkonstruktionen geführt. Dazu gehören die so genannten „Knickarmroboter“, die sich durch viele Freiheitsgerade auszeichnen und besonders für komplexe Aufgaben geeignet sind. Die meisten Menschen assoziieren mit dem Begriff Industrieroboter Knickarmroboter. Weit verbreitet sind ebenfalls Schwenkarm- bzw. SCARA-Roboter. Sie besitzen meist nur einen Freiheitsgrad von 3,5 und finden bei einfachen Aufgaben, wie dem Bestücken einer Platine Verwendung. Das letzte verbreitete Design sind die Linearroboter. Sie besitzen gleich-

falls einen Freiheitsgrad von 3,5 und werden in den gleichen Szenarien wie SCARA-Roboter eingesetzt. Die konstruktiven Unterschiede verdeutlichen folgende Grafiken:





Knickarmroboter Schwenkarmroboter Linearroboter

7.2 Onlineprogrammierung

Kommen wir zurück zum eigentlichen Thema dieser Ausarbeitung, der Programmierung eines Roboters. Im Fall von Industrierobotern wird dabei zwischen Online- und Offlineprogrammierung unterschieden. Onlineprogrammierung hat aber nicht zu bedeuten, dass die Programmierung über ein Netzwerk erfolgt, sondern dass die Eingabe des Programms direkt am Roboter erfolgt. Es wird zwischen der „Teach-in“-Methode und der Master-Slave-Methode unterschieden. Beim „Teach-in“ erfolgt die Eingabe durch eine so genannte „Teachbox“, ein kompliziertes Eingabegerät, das eine genaue Steuerung des Roboters ermöglicht. Hat der Effektor die gewünschte Koordinate erreicht, wird sie als Raumpunkt gespeichert. Dieser Vorgang wird solange wiederholt, bis ein vollständiges Bewegungsmuster entstanden ist. Bei Bedarf kann das so erstellte Programm noch per Hand nachbearbeitet werden. Im Betrieb fährt der Roboter die zuvor programmierten Punkte ab. Daher handelt es sich hierbei um eine so genannte Punk-zu-Punkt Steuerung. Im Gegensatz hierzu erzeugt die Master-Slave-Programmierung eine Bahnsteuerung bei der nicht nur nicht nur einzelne Punkte, sondern die Bewegungen als Ganzes gespeichert werden. Die Eingabe erfolgt indem ein Mensch den Effektor entlang der gewünschten Bahn bewegt. Der Roboter zeichnet dabei die abgefahrenen Koordinaten und die dazugehörigen Beschleunigungen auf, so dass später eine exakte Wiederholung möglich ist. Besonders bei der Master-Slave-Programmierung kann das Wissen erfahrener Mitarbeiter einfließen. Als Beispiel sei hier ein Lackierer genannt.

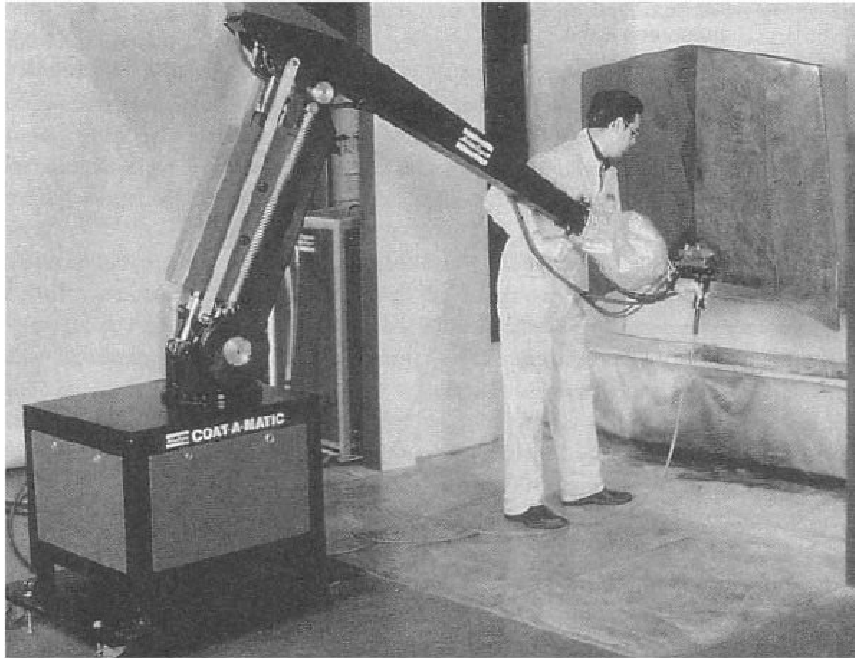


Abbildung 14: Onlineprogrammierung Teach-in-Methode

Der große Nachteil der Onlineprogrammierung ist, dass zur Programmierung eine Arbeitsstation genutzt werden muss. Weiterhin ist keine Reaktion auf Sensordaten möglich, und somit auch keine komplexen Programme.

7.3 Offlineprogrammierung

Die Mängel der Onlineprogrammierung werden von der Offlineprogrammierung behoben. Im Gegensatz zur gerade vorgestellten Methode wird das Programm nicht mit Hilfe des Roboters erzeugt, sondern an einer Workstation. Es lassen sich dabei drei verschiedene Ansätze erkennen, erstens die textuelle Programmierung, zweitens die visuelle Programmierung und drittens die Simulationsprogrammierung.

Die textuelle Programmierung weist die typischen Elemente einer Programmiersprache auf, z.B. Kontrollstrukturen, Schleifen, Variablen und arithmetische Funktionen. Zusätzlich existiert eine umfangreiche API, die Methoden zur Steuerung und Positionierung des Roboters sowie zur Interaktion mit den Sensoren bereitstellt. Unterschiede existieren zwischen den einzelnen Sprachen hinsichtlich der Methodik. Wenn in einem Programm nur eine Folge von kartesischen Koordinaten, bzw. exakte Bahnen angegeben sind, spricht man von expliziten Verfahren, wird dagegen eine Reihe von Aufgaben beschrieben handelt es sich um ein implizites Verfahren. Damit eine Aufgabe der Art „Schweiße Tür" richtig erledigt werden kann, ist vorher eine exakte Beschreibung der

Arbeitsumgebung nötig. Letztendlich muss zur textuellen Programmerstellung noch gesagt werden, dass es eine Vielzahl von möglichen Sprachen gibt, die jedoch herstellerspezifisch und somit auf eine Teilmenge aller Roboter beschränkt sind. Zu nennen wären unter anderem die „Industrial Robot Language“ (IRL), die „Kuka Robot Language“ (KRL) oder Bosch Automatisierungs-Programmiersprache (BAPS) (vgl. [Bo95]).

Mit dem Aufkommen leistungsstarker CAD-Anwendungen kam der Wunsch in vielen Betrieben auf, die Roboter durch das CAD-System zu programmieren zu lassen. Im CAD-Programm neu erstellte Teile sollten nun nicht mehr manuell in die Schweißroboter eingegeben werden, sondern die Software sollte diese Aufgabe autonom erledigen. Heutzutage ist die Kopplung soweit fortgeschritten, dass die Programmierung entweder direkt durch die Software erfolgt, oder durch Interaktion eines Benutzers mit einer drei dimensionalen Entwicklungsumgebung. Der Benutzer ist in der Lage die Position des 3D-Robotermodells am Rechner zu verändern und dessen Werkzeuge zu benutzen. Aus den aufgezeichneten Daten erstellt das Programm schließlich den eigentlichen Programmcode. Die meisten Programme unterstützen eine Vielzahl von Robotern und deren Sprachen. Das folgende Beispiel zeigt einen Screenshot einer solchen Umgebung.

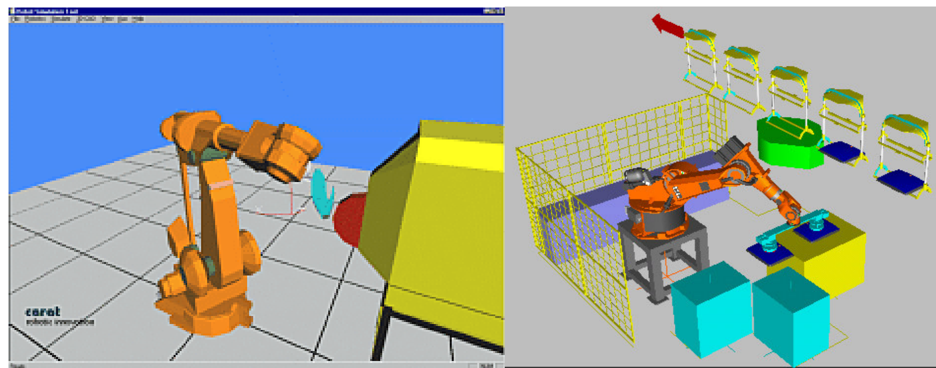


Abbildung 15: EasyRob und KukaSim (Quelle: www.easyrob.com, www.kuka.be)

Der letzte und auch aktuellste Ansatz ist eine konsequente Weiterentwicklung der visuellen Programmierung. Statt die Codegenerierung nur mit einem CAD-System zu koppeln, wird in den neusten Ansätzen die Programmierung von einem Simulationssystem übernommen. Dieses System wird zuvor mit den nötigen Daten gespeist bis eine Simulation des Fertigungsprozesses möglich ist. Auf diese Weise können global optimale Programme entwickelt werden, da die Simulationsumgebung nicht nur die einzelne Arbeitsstation, sondern ganze Produktionsstrassen simulieren kann.

Abschließend lässt sich sagen, dass die Offline- der Onlineprogrammierung überlegen ist. Es lassen sich leichter komplexere Programme erstellen, da der Programmierungsprozess im Bereich der Offlineprogrammierung besonders durch die Visuelle- bzw. Simulationsprogrammierung entscheidend vereinfacht worden ist.

8 Zusammenfassung

In dieser Arbeit wurden mehrere Programmiersprachen zur Programmierung des Lego-Bricks vorgestellt. Die Sprachen weisen deutliche Unterschiede in den verwendeten Programmierkonzepten auf. Für den Brick wurden graphische und textuelle sowie prozedurale und objektorientierte Sprachen beschrieben. Die Leistungsfähigkeit der Sprachen ist in besonderem Maße von der verwendeten Firmware abhängig. Diejenigen Sprachen, die die Lego-Firmware ersetzen schränken die Möglichkeiten der Programmierung deutlich weniger ein.

Im Anschluss an die Brick-Programmiesprachen wurde in einem weitern Kapitel auf die Programmierung von Industrierobotern eingegangen. Beim Vergleich der professionellen Roboterprogrammierung mit der Programmierung des RCX-Bricks von Lego fällt auf, dass diese nicht auf derselben Abstraktionsebene arbeiten. So befinden sich die LegoBrick-Sprachen auf einer deutlich niedrigeren Ebene als etwa die Industrieroboter-sprachen. Erstgenannte unterstützen nur einfache Outputoperationen, z.B. in der Form „drehe Motor A vorwärts“. Industrierobotersprachen arbeiten dagegen auf einer höheren Ebene, es muss nur noch die abzufahrende Bahn programmiert werden. Wie die einzelnen Motoren dabei angesteuert werden, ist für den Programmierer unwichtig und wird durch die Steuerung des Roboters automatisch erledigt. Diese Art der Programmierung, ist auf Grund der Abstraktion von der eigentlichen Hardware wesentlich komfortabler und weniger fehleranfällig, da sich der Programmierer auf die Aufgaben des Roboters konzentrieren kann. Die Systeme zur visuellen Programmierung von Industrierobotern arbeiten noch eine Ebene höher. Zwar kann in der visuellen Entwicklungsumgebung eine Bewegungsfolge des Roboters auch per Hand eingegeben werden, doch die Kopplung mit CAD-Systemen vereinfacht die Arbeit deutlich. Die Aufgabe des Programmierers beschränkt sich nun nur noch darauf, die Arbeitsumgebung und die Aufgaben des Roboters im Computer abzubilden, die einzelnen Bewegungen werden autonom berechnet. Eine Ausnahme bildet die Onlineprogrammierung (besonders die Teach-in-Methode). Der Benutzer muss sich zwar nicht mit den Hardwaredetails befassen, aller-

dings ist das Erstellen komplexer Programme mit dieser Methode recht schwierig, und auch die Tatsache, dass ein Arbeitsplatz belegt wird, ist in der Praxis oft problematisch. Zusammenfassend lässt sich sagen, dass sich die Programmierung im Laufe der Zeit zunehmend vereinfacht hat. Dies ist darauf zurück zu führen, dass die Konstrukteure der Roboter viele Low-Level-Operationen, wie etwa die Motorsteuerung, bereits implementieren und so die Abstraktion von der eigentlichen Maschine vorantreiben.

Literaturverzeichnis

[Ba03] David Baum, *NQC Programmer´s Guide*, http://www.baumfamily.org/nqc/ Abruf 5.9.2003, 2003

[Bi98] Judy M. Bischop, *Java Gently*, Addision Wesley, 1998

[Bo95] Ulrich Borgolte, IRL - *Die deutsche Norm für explizite Roboterprogrammierung*, http://prt.fernuni-hagen.de/forschung/BORGOLTE/abs-b93.html, Abruf 10.9.2003, 1995

[Ch03] Eugene Charniac, *Building Intelligent Robots*, http://www.cs.brown.edu/courses/cs148/2003/brickOS.html, Abruf 16.12.2003, 2003

[FL02] Giulio Ferrari, Dario Laverede, *Programming Lego Mindstorms with Java, Syngress*, 2002.

[He03] Ralph Hempel, *pbFORTH Homepage*, http://www.hempeldesigngroup.com/lego/pbForth/homePage.html, Abruf 17.9.2003, 2003

[Kn99] Jonathan Knudsen, *The unofficial Guide to Lego® Mindstoms*, O'Reilly & Associates, 1999.

[Lo98] Dirk Louis, *C/C++-Kompendium*, Markt und Technik, 1998.

[Sc02] Henner Schneider, *Grundlagen der Robotik*, http://www.fbi.fh-darmstadt.de/~schneider/ba-wp-ro/skriptum.pdf, Abruf 11.9.2003, 2002.

[SF03] SourceForge, *brickOS at SourceForge*, http://brickos.sourceforge.net, Abruf 14.12.2003, 2003

[VD90] Verein Deutscher Ingeneure VDI, *2860 Montage- und Handhabungstechnik; Handhabungsfunktionen, Handhabungseinrichtungen; Begriffe, Definitionen, Symbole*, http://www.vdi.de/vdi/vrp/richtliniendetails/index.php?ID=2372581, Abruf 6.9.2003, 1990-