Universität des Saarlandes
Fachrichtung 4.7 – Computerlinguistik
Softwareprojekt: Sprechende Roboter mit LEGO MindStorms
Seminarleitung: Alexander Koller und Geert-Jan Kruijff
Wintersemester 2002/2003

Ein sprachgesteuerter pyramidenerkundender Roboter mit LEGO MindStorms

Stefan Girgsdies
stgi@coli.uni-sb.de

Inhaltsverzeichnis

1 Einleitung

Das Ziel des Softwareprojekts *Sprechende Roboter mit LEGO MindStorms* bestand in der Kombination von Robotik und Computerlinguistik. Die Aufgabe der vier Teams war die Konstruktion eines per Sprache kommunizierenden Roboters. Es galt, nach eigenen Ideen mit dem LEGO MindStorms-System einen Roboter zu bauen, diesen mit LeJOS zu programmieren und in ein Dialogsystem samt Spracherkenner und -synthese einzubetten. Unser Team[1] konstruierte einen kleinen kettengetriebenen Roboter zur Erkundung einer selbstgebastelten Pyramide mit verwinkelten Gängen und Hindernissen aus LEGO und Holz. Die Idee dazu stammt von *The UPUAUT Project*[2], in dessen Rahmen in den Neunzigerjahren des letzten Jahrhunderts verschiedene Roboter zur Erkundung der Cheops-Pyramide in Ägypten konstruiert wurden. Wie diese wurde auch unser Roboter mit einer kleinen Kamera ausgerüstet, die es erlaubt, den Weg durch enge, für Menschen unzugängliche Gänge am PC mitzuverfolgen.

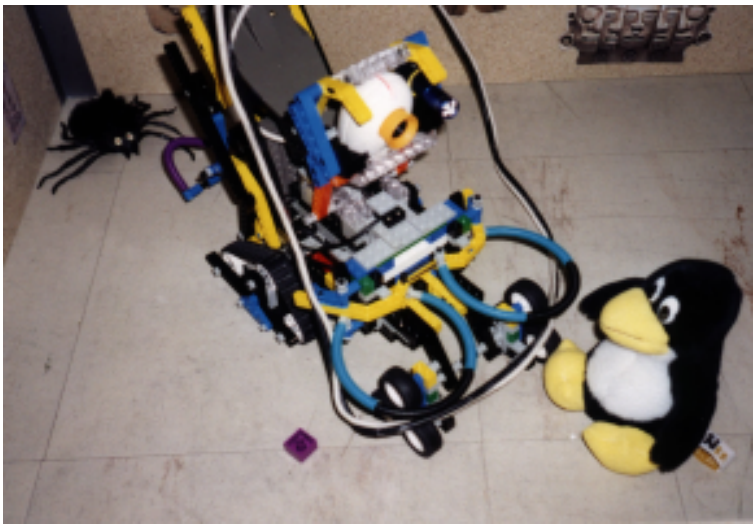


Abbildung 1: Unser pyramidenerkundender Roboter, genannt "Indi"

Unser Roboter ist vorne links, vorne rechts und hinten mit jeweils einem Kollisionsdetektor ausgestattet. Beim Aufprall auf einen großen Gegenstand oder eine Wand wird diese Information über Berührungssensoren an den RCX, den "programmierbaren LEGO-Stein" des LEGO MindStorms-Systems, weitergeleitet. Das Java-Programm, welches auf dem RCX läuft, veranlasst den Roboter, entsprechend zu reagieren, zum Beispiel ein Ausweichmanöver einzuleiten.

Neben den üblichen Steuerungsmöglichkeiten - vorwärts, rückwärts, Drehung links/rechts - kann auch der Arm, an dem die mitgeführte Kamera befestigt ist, nach oben und unten bewegt werden, was eine Veränderung der Perspektive des übermittelten Bildes ermöglicht. Alle diese Bewegungen können durch vordefinierte lautsprachliche Kommandos über ein Mikrophon am PC gestartet und gestoppt werden.

[1] Monica Rodriguez, Ruth Kusterer, Stefan Girgsdies
[2] http://www.cheops.org/

Zur Sprachsteuerung des Roboters laufen auf dem PC verschiedene Programme. Im Zentrum steht dabei ein Dialogsystem der Firma CLT[3]. Für die sprecherinnenunabhängige Spracherkennung sowie die Sprachsynthese sorgen zwei Programme von Lernout & Hauspie, welche als Clients angebunden werden und die Java Speech API[4] nutzen. Ein weiterer Client übernimmt die Kommunikation mit dem RCX. Die Nachrichtenübermittlung zwischen diesem Client und der RCX erfolgt per Infrarotlicht. Ein Mindstorms-Roboter kann sich somit normalerweise einigermaßen frei im Raum bewegen. Da im Fall unseres Roboters jedoch auch die Bilddaten der mitgeführten Kamera zum PC übertragen werden müssen und sich die Infrarot-Signalübertragung innerhalb der selbstgebastelten Pyramide als problematisch erwies, haben wir uns entschieden, auf die kabellose Übertragung zu verzichten. Wie die original UPUAUT-Roboter zieht daher auch unser "Indi" ein Kabelbündel hinter sich her.

Im folgenden werde ich das LEGO MindStorms-System (Kapitel 2) und die Java-Programmierumgebung leJOS (Kapitel 3) vorstellen. Danach werde ich auf zwei Aspekte unseres Roboters näher eingehen. Zum einen auf die Möglichkeit, eigene Sensoren für die Steuerung eines LEGO MindStorms-Roboters zu basteln sowie die Anzahl der Sensoreingänge durch den Einsatz eines Multiplexers zu erhöhen. Beides kam bei unserem Roboter zur Anwendung, was ich in Kapitel 4 darstelle. Schließlich wird in Kapitel 5 das leJOS-Programm beschrieben, welches auf dem RCX unseres Roboters läuft und die Reaktionen auf bestimmte sensorische Ereignisse sowie die Kommunikation mit dem RCX-Client auf dem PC ermöglicht.

2 LEGO MindStorms und der RCX

Die Produktreihe *MindStorms*[5] wurde von LEGO in Zusammenarbeit mit dem Massachusetts Institute of Technology (MIT) in Boston entwickelt. Der Grundbaukasten ist das *Robotics Invention System* (RIS) mit über 700 LEGO-Steinen. Das "Gehirn" des Systems stellt der Mikrocomputer RCX - *Robotics Command System* - dar, ein programmierbarer LEGO-Baustein. Der RCX verfügt über drei Ausgänge für Motoren und drei Eingänge, an die verschiedene Sensoren angeschlossen werden können. Die Kommunikation mit dem PC erfolgt drahtlos über eine Infrarotschnittstelle. Dazu wird PC-seitig ein IR-Tower angeschlossen, über den nun am PC geschriebene Programme oder Daten zum RCX geschickt werden können. Darüberhinaus kann der RCX auch Daten an den PC oder über sogenannte "Messages" an andere RCX-Einheiten schicken.

Weiterhin verfügt der RCX über eine interne Systemuhr und drei Timer für zeitgesteuerte Aktionen, einen Lautsprecher, über den verschiedene Pieptöne ausgegeben werden können und ein einfaches LCD-Display. Für die Stromversorgung von 9 Volt sorgen sechs AA-Batterien. Mit Hilfe eines Datenprotokolls können die Werte der Sensoreingänge, von Variablen, Timern und der Systemuhr auf dem RCX protokolliert und später vom PC aus abgefragt werden. Dies ist zum einen für den Test von komplexen Programmen nützlich, der RCX kann dadurch außerdem als eine Art Messstation benutzt werden. Insgesamt stehen für Programme und das Datenprotokoll etwa sechs Kilobyte Speicherplatz zur

[3] http://www.clt-st.de/
[4] http://java.sun.com/products/java-media/speech/
[5] http://mindstorms.lego.com/

Verfügung.

Das RIS wird inzwischen - nach den Versionen 1.0 und 1.5 - in der Version 2.0 ausgeliefert. Meines Wissens bestehen die Unterschiede zwischen den verschiedenen Versionen im wesentlichen in der verbesserten Programmier-Software, die wir für unser Projekt jedoch gar nicht genutzt haben. Außerdem wird der Infrarot-Tower zur Kommunikation zwischen PC und RCX beim RIS 2.0 über die USB-Schnittstelle angeschlossen, bei den älteren Versionen über die serielle Schnittstelle, wodurch hier noch eine Stromversorgung mit einer zusätzlichen Batterie notwendig ist. Beim RIS 2.0 wird außerdem ein Lichtwellenleiter mitgeliefert, mit dem mensch den *Micro-Scout* - ein nur sehr eingeschränkt programmierbarer Mikrocomputer aus dem *Droid Developer Kit* bzw. *Dark Side Developer Kit* - mit dem IR-Tower verbinden kann. Während der RCX aus dem RIS 1.0 noch über einen Netzteilanschluss verfügte, kann die neuere Version leider nur noch mit Batterien betrieben werden.

Abbildung 2: Der RCX, das "Gehirn" der LEGO MindStorms-Roboter

2.1 Die Motoren

Es gibt drei unterschiedliche LEGO-Technik-Motoren, die an die 9V-Ausgänge angeschlossen werden können: Der *Standard-Motor* ist als Hochgeschwindigkeitsmotor mit niedrigem Drehmoment seit vielen Jahren Bestandteil der Technik-Serie. Die meisten motorisierten Sets enthalten diesen Motor, der während des Normalbetriebs mit ungefähr 100mA läuft. Der *Mikro-Motor* ist ein sehr kompakter, langsamer Motor mit mittlerem Drehmoment. Dieser Motor hat nicht genügend Energie, um als Teil eines Hauptantriebs zu taugen, kann aber für einige Hilfsfunktionen verwendet werden. Der neue *Zahnrad-Power-Motor* hat interne Zahnradübersetzungen und ein Schwungrad. Infolgedessen hat er eine mittlere Geschwindigkeit mit hohem Drehmoment und läuft sehr gleichmäßig bei sehr geringem Stromverbrauch - nur 10mA während des Normalbetriebs. Das RIS enthält standardmäßig zwei dieser Power-Motoren.

Zum Betrieb der Motoren können die drei Ausgänge auf drei verschiedene Modi gesetzt werden: an (on), aus (off) und gleitend (float). Ein Motor, der

auf *float* gesetzt wurde, läßt sich frei drehen. Im Modus *off* wird der Motor hingegen durch einen Kurzschluss gebremst, was ein Weiterrollen des Roboters verhindert. Die Drehrichtung kann frei gesteuert werden, hängt aber auch davon ab, wie der Motor angeschlossen wurde. Zur Steuerung der Drehgeschwindigkeit stehen acht Stufen zur Verfügung, bei denen die Spannung des Ausgangs jedoch konstant bleibt. Hier kommt das Verfahren der *Pulse Width Modulation* (PWM) zur Anwendung: Lediglich bei der höchsten Stufe liegt die Spannung die ganze Zeit über an, für die niedrigeren Geschwindigkeiten wird sie periodisch für mehr oder weniger lange Zeit abgeschaltet.

2.2 Die Sensoren

An die drei Eingänge des RCX können verschiedene Sensoren angeschlossen werden. Mit dem RIS werden zwei Berührungssensoren und ein Lichtsensor mitgeliefert. Darüberhinaus gibt es zum Beispiel noch einen Rotationssensor sowie einen Temperatursensor.

Generell lässt sich zwischen aktiven und passiven Sensoren unterscheiden. Der Berührungssensor ist ein einfacher, passiver Sensor, der keine Stromversorgung benötigt. Mit Hilfe einer Spannung von 5 Volt wird über die Messung des Spannungsabfalls der Widerstand des Sensors gemessen und damit festgestellt, ob und wie stark dieser gedrückt ist - siehe dazu auch Kapitel 4. Auch der Temperatursensor ist ein passiver Sensor, er enthält einen temperaturabhängigen Widerstand.

Bei den Licht- und Rotationssensoren handelt es sich hingegen um aktive Sensoren. So verfügt der Lichtsensor über eine eingebaute LED. Diese wird durch eine periodisch dazugeschaltete Spannung von 8 Volt betrieben. Der RCX versorgt dazu den Sensoreingang 3 Millisekunden lang ständig mit Strom, wobei außerdem ein Kondensator im Sensor aufgeladen wird. Dann wird die Stromzufuhr für 0,1 Millisekunden unterbrochen. In dieser Zeit wird der Sensor von dem nun geladenen Kondensator mit Strom versorgt, während der RCX den Sensorwert ausliest. Auch der Rotationssensor wird im aktiven Modus betrieben, wodurch zwischen Vorwärts- und Rückwärtsrotation unterschieden werden kann[6].

3 RCX-Code und leJOS

Die Programmierumgebung, die zum *Robotics Invention System* mitgeliefert wird, ermöglicht die Programmierung des RCX mit Hilfe des *RCX-Code*. Um auch Menschen ohne Programmiererfahrung die unkomplizierte Nutzung zu ermöglichen, wurde eine Art "Baukastensystem" entwickelt: In einer graphischen Umgebung können Befehlsblöcke für die verschiedensten Aufgaben - Variablenbelegung und -abfrage, Verzweigungen, Schleifen, Motoransteuerung usw. - mit der Maus aus Schubladen entnommen und zu einer Art Flussdiagramm aneinandergehängt werden. Die Programmierung wird dadurch sehr intuitiv, es muss keine spezielle Programmiersprache gelernt, kein für Nichtinformatikerinnen unübersichtliches Programm im Textmodus abgetippt werden. Allerdings lassen

[6] Ein Betrieb des Rotationssensors im passiven Modus ist jedoch auch möglich, siehe dazu Kapitel 5.4.

sich in dieser Programmierumgebung nicht alle Möglichkeiten, die der RCX bietet, ausnutzen, da die einfache Bedienung im Vordergrund steht.

Als gute Alternative zum RCX-Code wurden von verschiedenen Programmiersprachen Versionen entwickelt, die auf dem RCX laufen. Neben NQC, QBasic und FORTH können auch Programme in Java auf dem RCX ausgeführt werden. Für die Programmierung unseres Roboters haben wir die Java-Umgebung leJOS[7] genutzt. Neben der Installation des JDK ab Version 1.1 muss dafür zunächst die mitgelieferte und vorinstallierte LEGO-Firmware des *Robotics Command System* durch eine neue ersetzt werden. Diese ist als OpenSource-Software verfügbar und stellt eine *Java virtual machine* (JVM) bereit, welche es ermöglicht, Java-Byte-Code-Programme, die mit dem leJOS-Compiler erzeugt wurden, auf den RCX zu laden und dort auszuführen.

Das Java, welches auf dem RCX läuft, unterscheidet sich vom Standard-Java durch gewisse Einschränkungen. So sind zum Beispiel keine *switch*-Anweisungen oder *long*-Variablen möglich, und die Größe von Arrays ist auf 511 Objekte beschränkt. Darüberhinaus sollte mensch acht geben, nicht zu viele Speicherplatz füllende Objekte zu erzeugen, da kein Garbagecollector vorgesehen ist, der diesen wieder freischaufelt. Zwar stehen die meisten JDK-APIs nicht zur Verfügung, dafür erlauben die Pakete der leJOS-API[8], die Möglichkeiten des RCX gut auszunutzen.

So bietet das Paket *josx.platform.rcx* Zugriffsmöglichkeiten auf die Motorausgänge und Sensoreingänge des RCX. Auch die internen Timer, das LCD und die Soundausgabe können genutzt werden. Außerdem sind recht komplexe Kontroll- und Steuerungsfunktionen vorgesehen. Zur Überwachung eines Sensoreinganges kann zum Beispiel eine Klasse genutzt werden, die das Interface *SensorListener* implementiert. Wird einem Sensoreingang mit Hilfe der Methode *addSensorListener(SensorListener aListener)* ein solcher Sensor-Listener zugewiesen, so wird ein neuer Thread erzeugt, welcher diesen Sensoreingang überwacht. Ändert sich der Wert, der an diesem Sensoreingang gemessen wird, so wird entsprechend der Vorgaben in der Methode *stateChanged(Sensor aSource, int aOldValue, int aNewValue)* des Sensor-Listeners reagiert. Ein Roboter kann sich somit mit dem Haupt-Thread ganz intensiv einer bestimmten Aufgabe widmen - wie zum Beispiel vorwärts Fahren und dabei ein Liedchen Piepen - und trotzdem auf Unvorhergesehenes - wie eine Kollision mit einer Wand oder einem anderen Roboter - passend reagieren.

Das Paket *josx.robotics* bietet darüberhinaus Klassen und Interfaces, die Methoden für die grundlegende Navigation eines Roboters zur Verfügung stellen. Diese ermöglichen außerdem, ein recht komplexes Verhalten des Roboters anhand bestimmter Zustände und Reaktionen auf Ereignisse - zum Beispiel die Änderung eines Sensorwertes - zu definieren.

4 Erweiterung der Sensoreingänge

Bei der Sammlung von Ideen zu unserem Roboter-Projekt wurde uns ziemlich schnell klar, dass die drei Sensoreingänge für unsere Anforderungen nicht ausreichen würden. Ein mobiler Roboter, der sich vorwärts und rückwärts bewegen sowie nach links und rechts um die eigene Achse drehen kann, sollte mindestens

[7] http://lejos.sourceforge.net/
[8] http://dudley.wellesley.edu/ anderson/robots/lejosdocs/

mit drei wenn nicht sogar vier Kollisionsdetektoren ausgestattet sein. Wir entschieden uns für drei: jeweils einer vorne links und rechts, um während der Vowärtsbewegung und den Drehungen Hindernisse zu registrieren, und einer hinten, um auf Kollisionen während des Rückwärtsfahrens oder bei Ausweichmanövern entsprechend reagieren zu können[9].

Die drei Eingänge des RCX hätten somit ausgereicht, allerdings wollten wir einen der Eingänge für eine präzisere Steuerung des Kameraarms nutzen und uns einen weiteren für die Erweiterung mit einem Farbsensor offenhalten, zu dessen Konstruktion wir letztendlich aber nicht mehr kamen. Für die Kollisionsdetektoren blieb somit nur ein einziger Sensoreingang zur Verfügung. Durch eine Parallelschaltung hätten wir eine ODER-Verknüpfung der drei Berührungssensoren realisieren können. Es erschien uns für eine optimale Reaktion jedoch wichtig, feststellen zu können, *welcher* der drei Kollisionsdetektoren "gefeuert" hat. Zwar wird bei der Vorwärtsbewegung des Roboters die Kollision kaum am hinteren Detektor erfolgen, jedoch ist es zum Beispiel bei schrägem Auffahren auf eine Wand entscheidend, zu erkennen, ob die Kollision vorne links oder vorne rechts stattfand, um ein entsprechendes Ausweichmanöver einzuleiten.

4.1 Der Multiplexer

Auf verschiedenen MindStorms-Bastlerinnen-Seiten im Internet sowie in Knudsen (1999) fand ich zahlreiche Hinweise darauf, wie mehrere unterscheidbare Berührungssensoren an einem Sensoreingang genutzt werden können. Zunächst muss dafür die genaue Funktionsweise passiver Sensoren am RCX betrachtet werden. Wie schon in Kapitel 2.2 erwähnt, ist der passive Sensor-Modus für solche Sensoren gedacht, bei denen lediglich ein Widerstand gemessen werden muss, die aber zum Betrieb keine Stromversorgung benötigen.

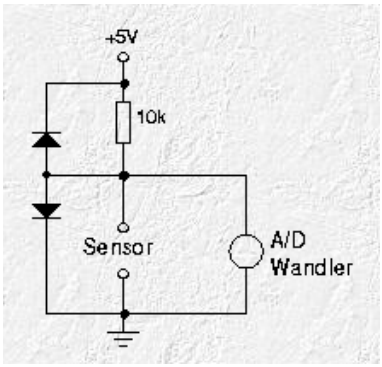


Abbildung 3: Schaltplan eines Sensoreingangs des RCX im passiven Modus

Dies ist zum Beispiel beim Berührungssensor und beim Temperatursensor der Fall. Beide bestehen elektrisch gesehen aus einem Widerstand, an den im RCX über einen Widerstand von 10 Kilo-Ohm eine Spannung von 5 Volt angelegt wird. Ein Analog/Digital-Wandler konvertiert die Spannung, die an dem Sensor abfällt, in eine 10-Bit-Zahl: *0* entspricht 0V, *1023* entspricht 5V Spannungsabfall. Eine Schutzschaltung aus zwei Dioden verhindert, dass eine externe

[9] Die Kollisionsdetektoren sind in Abb. 1 gut zu erkennen.

Spannungsquelle den A/D-Wandler zerstören kann.

Im RCX können die Werte des A/D-Wandlers je nach Art des angeschlossenen Sensors verschieden interpretiert werden. Der Rohwert *raw*, aus dem alle übrigen Werte berechnet werden können, ergibt sich aus dem Spannungsabfall U_{ab} nach folgender Formel: $raw = \frac{U_{ab}}{5V} * 1023$. Der Widerstand R des Sensors kann mit $R = \frac{10k\Omega * U_{ab}}{5V - U_{ab}}$ berechnet werden. Daraus ergibt sich für uns die Möglichkeit, in einem Java-Programm den raw-Wert am RCX auszulesen und R direkt daraus zu berechnen: $R = \frac{10k\Omega * raw}{1023 - raw}$.

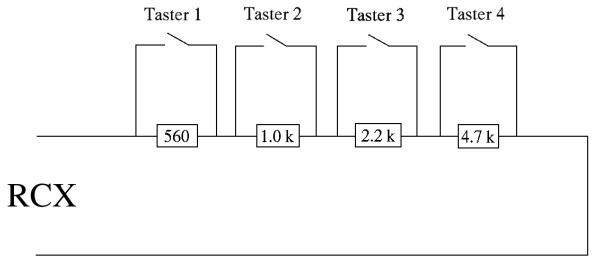


Abbildung 4: Schaltplan des Multiplexers

Der Schaltplan in Abb. 4 verdeutlich, wie ein einzelner Sensoreingang des RCX für den Anschluss von bis zu vier unterscheidbaren Berührungssensoren genutzt werden kann. Dazu werden vier Widerstände in Reihe geschaltet, die jeweils von einem einfachen Drucktaster überbrückt werden. Da jeder der Widerstände jeweils etwa doppelt so groß wie der vorangehende ist, läßt sich aus dem raw-Wert am RCX relativ einfach berechnen, welche der Taster zu einem bestimmten Zeitpunkt gedrückt sind und welche nicht.

Taster1	Taster2	Taster3	Taster4	Widerstand/Ω	raw-Wert
0	0	0	0	8460	469
1	0	0	0	7900	451
0	1	0	0	7460	437
1	1	0	0	6900	418
0	0	1	0	6260	394
1	0	1	0	5700	371
0	1	1	0	5260	353
1	1	1	0	4700	327
...	...	...	...	...	...

Tabelle 1: Ausschnittsweise Darstellung der Widerstände und raw-Werte am Multiplexer bei verschiedenen Kombinationen von gedrückten Tastern

Um den Multiplexer möglichst LEGO-kompatibel zu gestalten, habe ich ihn in einen 4x6 LEGO-Einheiten großen Block der Höhe 2/3 (eine Platte hat 1/3 der Höhe eines normalen LEGO-Steines) integriert. Zunächst habe ich dazu mit einem Teppichmesser jeweils diagonal nebeneinanderliegende Noppen von zwei

2x6-Platten entfernt und durch kleine leitende Zylinder-Schräubchen[10] ersetzt, deren Köpfe exakt auf die Größe der Noppen zurechtgefeilt wurden. Dies ist in Abb. 5 (links) zu sehen. Natürlich können - sofern vorhanden - auch die originalen elektrischen Platten von LEGO verwendet werden. An die Paare aus gegenüberliegenden Metall-Noppen wird später über ein LEGO-Kabel mit 2x2-Platten je ein Berührungssensor angeschlossen. Durch die diagonale Anordnung der Schräubchen muss nicht darauf geachtet werden, in welcher Orientierung das LEGO-Kabel angeschlossen wird. Die beiden Schräubchen in der Mitte der einen 2x6-Platte dienen zum Anschluss an den RCX.

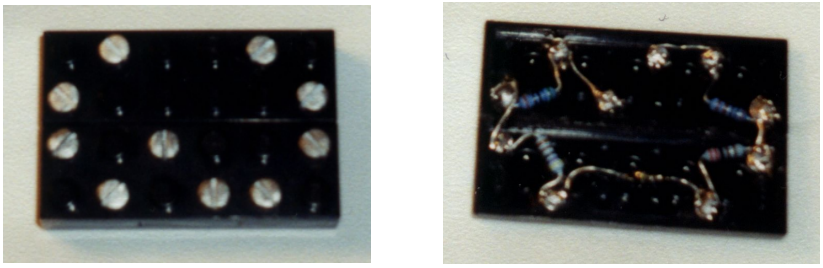


Abbildung 5: Der Multiplexer - links: Draufsicht, rechts: Innenansicht

Zwischen die zwei Schräubchen der vier Paare, an die die Sensoren angeschlossen werden sollen, habe ich dann jeweils einen der Widerstände gelötet. Weiterhin wurde jedes der Paare mit seinem Nachbarpaar direkt verbunden, wofür durch geschicktes Biegen der Widerstandsbeinchen kein zusätzlicher Draht nötig war. Am Ende entstand auf diese Weise die Reihenschaltung der vier Widerstände, mit den zwei Schaltkreisenden am Anschluss für den RCX - in Abb. 5 (rechts) ist dies gut zu erkennen.

Zum Abschluss habe ich drei 2x4-Platten quer unter die 2x6-Platten geklebt. Um die Widerstände nicht zu beschädigen, musste ich auch an diesen Platten einige der Noppen entfernen. Der Multiplexer ist nun recht stabil, die Widerstände zwischen den Platten sind gut geschützt, allzu hoher Druck sollte allerdings vermieden werden, da der Lötzinn an den Schräubchen nicht sehr gut haftet.

4.2 Die Berührungssensoren

Bei den ersten Tests mit Multiplexer, Widerstandsmessgerät und Berührungssensoren wurde klar, dass die original LEGO-Drucksensoren für unseren Zweck nicht geeignet sind: Es handelt sich dabei nicht um einfache "an/aus-Taster" mit unendlichem Widerstand im nichtgedrückten und nahezu keinem Widerstand im gedrückten Zustand. Stattdessen sind beliebige Zwischenzustände möglich, der Widerstand variiert also abhängig von der Stärke des Drucks auf den Sensor. Wir könnten daher zum Beispiel nicht unterscheiden, ob der Taster am 560Ω-Widerstand ganz gedrückt wurde, oder der Taster am 1kΩ-Widerstand nur ein bisschen.

Daher beschloss ich, auch die Berührungssensoren selbst zu basteln. Mit Teppichmesser und Feile wird dazu ein 2x3-Stein ausgehöhlt und, wie schon für

[10] z.B. ZYLINDERSCHRAUBE M3 X 10 von Conrad Electronic (Artikel-Nr.: 814709)

den Multiplexer, zwei diagonal gegenüberliegende Noppen durch kleine zurechtgefeilte Schräubchen ersetzt. Mit einem über einer Kerzenflamme erhitzten Nagel wird dann in die Vorderseite des LEGO-Steins eines kleines Loch geschmolzen und mit einer Rundfeile vergößert. In den LEGO-Stein kann nun ein kleiner Drucktaster geklebt werden, dessen Knopf vorne aus dem Loch herausragt. Ich habe mich für einen Momentkontakt mit sehr kleinem Betätigungsweg (0,25mm) und geringer Andruckstärke (1,0Ncm) entschieden[11].

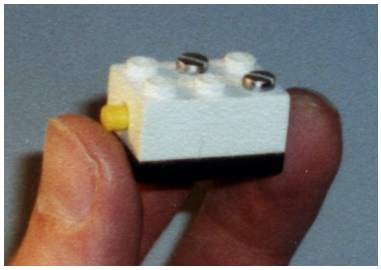
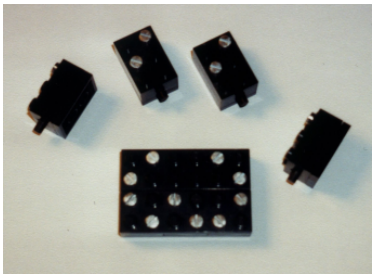


Abbildung 6: selbstgebastelter Berührungssensor (Prototyp)

Nun muss noch jeder der beiden Anschlusskontakte des Drucktasters über ein Stückchen leitenden Drahtes mit einem der Schräubchen verlötet werden. Zum Abschluss wird eine 2x3-Platte von unten an den 2x3-Stein geklebt, um den Berührungssensor sicher zu verschließen. Das Ergebnis ist in Abb. 6 zu sehen. Vier dieser Sensoren können nun über elektrische LEGO-Kabel an den Multiplexer angeschlossen werden, und mit einem Widerstandsmessgerät das gewünschte Verhalten überprüft werden.



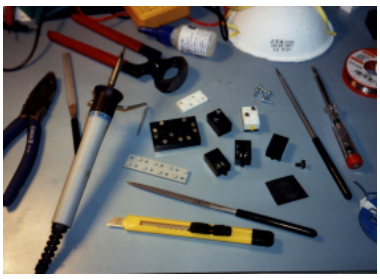


Abbildung 7: Die fertigen LEGO-Elemente und die zugehörige Bastelumgebung

Auch während der Konstruktion von Multiplexer und Berührungssensoren sollte nach jedem Schritt die Leitfähigkeit der Lötverbindungen überprüft werden, da diese zum Beispiel durch die Dämpfe bestimmter Kunstoffkleber beeinträchtigt werden kann, wie ich immer wieder feststellen musste. So ist es mir in der Testphase mehrmals passiert, dass ein fertiger Berührungssensor einwandfrei zu funktionieren schien, nach Abschluss mit der 2x3-Platte und dem Trocknen des Klebers aber bei gedrücktem Taster ein zu hoher Widerstand (um 100Ω, statt den erwarteten 2-10Ω) gemessen wurde.

[11] Geeignet ist z.B. der *DRUCKTASTER T604 = TS695* von *Conrad Electronic* (Artikel-Nr.: 700479) - Technische Daten: Abm. 6mm x 6mm; Schaltspannung max. 50V / 50mA; Betätigungsweg 0,25mm; Andruckstärke 1,0Ncm; Knopf-Ø 3,5mm.; Knopfl. 6,0mm.

5 Das Java-Programm auf dem RCX

Das leJOS-Java-Programm, welches auf dem RCX unseres Roboters läuft, besteht aus den Klassen, die in Anhang A-1 zu finden sind. Ich habe versucht, möglichst selbsterklärende Feld- und Methoden-Namen zu verwenden, sowie den Programm-Code übersichtlich zu gestalten und ausführlich zu kommentieren. Ich werde daher im folgenden nicht auf jede einzelne Methode im Detail eingehen, sondern lediglich die grundlegende Funktionsweise des Programms darstellen.

5.1 Die Hauptklasse: Indiana

Die Hauptklasse mit der main-Methode ist *Indiana* (Programmcode in Anhang A-1.1). Kompiliert wird das ganze Programm mit dem leJOS-Kompiler über den Aufruf `lejosc Indiana.java`. Die Übertragung per Infrarot vom PC auf den RCX erfolgt mit Hilfe von `lejos Indiana`.

Beim Ausführen des Programmes auf dem RCX - Anschalten über die rote On/Off-Taste, Start mit der grünen Run-Taste - wird die Methode `main()` der Klasse *Indiana* aufgerufen. Hier werden zunächst eine Instanz der Klasse *RCXPort* für die Datenübertragung vom und zum Infrarot-Turm am PC sowie eine Instanz `indi` der Klasse *Indiana* selbst erzeugt. Über den Konstruktor dieser Klasse erfolgen nun vor allem verschiedene Initialisierungen. Es werden die beiden Sensoreingänge konfiguriert - sowohl die drei Kollisionssensoren, die über den Multiplexer an einen einzigen Eingang geschaltet sind, als auch der Rotationssensor für die exakte Steuerung des Kameraarms werden als SENSOR_TYPE_TOUCH im Modus SENSOR_MODE_RAW behandelt. Nach der Aktivierung beider Sensoreingänge wird dem Kollisionssensor `collisionSensor` noch ein Sensor-Listener vom Typ *IndianaCollisionListener* zugewiesen. Dazu folgen weiter unten genauere Erläuterungen.

Desweiteren wird ein Objekt der Klasse *Timer* mit einem Timer-Listener vom Typ *IndianaTimerListener* (siehe Anhang A-1.6) kreiert und gestartet. Alle 100 Millisekunden ruft dieser Timer die Methode `timedOut()` des Timer-Listeners auf, welche ihrerseits über die Methode `incrTime()` das Integer-Feld `zeit` von `indi` inkrementiert. `zeit` gibt somit die seit der Instantiierung von `indi` vergangene Zeit in Zehntelsekunden an.

Als weitere Initialisierungsschritte folgen das Einstellen der Motorstärken und der Startwerte der Felder für die aktuelle Bewegungsrichtung des Roboters `direction`: HALT sowie die Kameraposition `camPos`: MIN_CAM_POS (Kameraarm ganz unten) und Kamerabewegung `camDirection`: CAM_HALT. Außerdem werden die drei Felder des boolean Array `collision[]` mit *false* initialisiert. Diese Werte signalisieren, ob und welcher der drei Berührungssensoren (vorne links, vorne rechts und hinten) gedrückt wird, also eine Kollision erfolgte.

Die main-Methode endet mit der Erzeugung und dem Starten von drei verschiedenen Objekten, deren Klassen von *Thread* erben. Zwei der Threads - Instanzen der Klassen *IndianaMoveIndi* (Kapitel 5.3) und *IndianaMoveCamera* (Kapitel 5.4) - sind für die Steuerung der Roboterbewegungen und der Kameraposition zuständig. Der dritte - Klasse *MessageProcessor* (siehe Kapitel 5.2) - ist für die Kommunikation mit dem PC verantwortlich.

5.2 Kommunikation zwischen PC und RCX

Eine der Klassen, welche für die Kommunikation des RCX mit dem Dialogsystem auf dem PC zuständig sind, ist *IndianaMessageHandler* (Programmcode in Anhang A-1.7). Diese implementiert das Interface *MessageHandler*. Ein Objekt der Klasse wird beim Start des RCX-Programmes in der main-Methode von *Indiana* erzeugt und einer neuen Instanz der Klasse *MessageProcessor* (Programmcode in Anhang A-1.8) übergeben, welche dann als selbständiger Thread gestartet wird. Diese Klasse übernimmt die Aufgabe, eine Nachricht `in` über den RCX-Port zu empfangen, die Methode `process(in, out)` des *IndianaMessageHandler*-Objektes aufzurufen und eine Antwort-Nachricht `out` an den PC zurückzuschicken.

Bei den Nachrichten `in` und `out` handelt es sich um Objekte der Klasse *DialogMessage* (Programmcode in Anhang A-1.9). Diese Klasse wird (in zwei Versionen: eine für die RCX- und eine für die PC-Seite) beim Kompilieren automatisch mit Hilfe der Spezifikationen in der Datei *indianamsg.xml* (siehe Anhang A-2.1) erzeugt. Im xml-Format sind dort die Nachrichten-Typen und ihre möglichen Argumente beschrieben. Für unseren Roboter sind dies drei Nachrichten-Typen: *gehe* für die Steuerung der Roboterbewegungen, *bewegearm* für die Kamerasteuerung und *status*, welcher für die Statusrückmeldung nach Ausführung einer Bewegung verwendet wird.

Als Argument *richtung* für die Steuerung der Roboterbewegungen sind verschiedene Integer-Zahlen vorgesehen, deren letzte Ziffer jeweils die Bewegungsrichtung angibt: *0* für halt, *1* für vorwärts, *2* für rückwärts, *3* für Drehung nach links und *4* für Drehung nach rechts. Die Ziffern davor geben an, wie lange die Bewegung ausgeführt werden soll. Als Einheit wird hier das Zehntelsekundenintervall des Timers verwendet. Ist das Argument einstellig, so bewegt sich der Roboter so lange in die entsprechende Richtung, bis ein neues Kommando erfolgt oder andere Umstände, wie die Kollision mit einem Hindernis, die Bewegung unterbrechen. Dies wird in Kapitel 5.3 noch ausführlicher dargestellt.

Für die Steuerung des Kameraarms können dem Integer-Argument *armrichtung* die Werte *1*, *2*, *11* und *12*, zugewiesen werden. Auch hier wird die letzte Ziffer als Bewegungsrichtung (*1* für aufwärts, *2* für abwärts) interpretiert. Ist das Argument einstellig, so erfolgt die Bewegung bis zur entsprechenden maximalen Kameraposition, ansonsten bewegt sich der Kameraarm so lange, bis der angeschlossene Rotationssensor anderthalb Umdrehungen registriert hat, dies entspricht etwa einem Achtel des Winkels zwischen tiefster und höchster Kameraposition. Kapitel 5.4 gibt hierzu nähere Informationen.

Die verschiedenen Kommandos zur Steuerung der Roboter- und Kamerabewegungen werden über einen Spracherkenner mit Hilfe einer kontextfreien Grammatik im Java-Speech-Grammar-Format (JSGF) entgegengenommen[12]. Entsprechend der Spezifikationen in der Datei *indianamsg.xml* werden passende Instanzen der Klasse *DialogMessage* erzeugt und per Infrarot an den RCX geschickt.

Vom dort laufenden Thread der Klasse *MessageProcessor* entgegengenommen, werden sie von der diesem zugewiesenen Instanz der Klasse *IndianaMessageHandler* interpretiert. Zunächst wird über das Feld `in.Type` festgestellt, um

[12] Die verwendete Grammatik und das Dialogsystem für unseren Roboter werden in den Arbeiten meiner beiden Team-Kolleginnen näher beschrieben. Zum JSGF siehe http://java.sun.com/products/java-media/speech/forDevelopers/JSGF/.

welchen Nachrichtentyp es sich handelt. Im Fall eines Kommandos zur Steuerung der Roboterbewegungen (*DialogMessage.*TYPE_GEHE) wird die gewünschte Richtung über den Aufruf `indi.setDirection(in.richtung)` an die laufende Roboter-Instanz übergeben. Für die Bewegung des Roboterarms (`in.Type` == *DialogMessage.*TYPE_BEWEGEARM) geschieht dies über `indi.setCamDirection(in.armrichtung)`. Trat keine Exception auf, so wird das Feld `out.okay` auf *0* gesetzt, andernfalls auf *-1*, und die Nachricht `out` vom Typ *DialogMessage.*TYPE_STATUS über den Message-Processor an den PC zurückgeschickt.

5.3 Robotersteuerung: IndianaMoveIndi

Die Klasse *IndianaMoveIndi* (Programmcode in Anhang A-1.2) erbt von der Klasse *Thread*. In der main-Methode der Klasse *Indiana* wird eine Instanz `moveIndi` dieser Klasse erzeugt und ihr über den Konstruktor `indi`, selbst ein Objekt der Klasse *Indiana*, übergeben. Nach dem Aufruf der Methode `run()` ist `moveIndi` für die Steuerung der Bewegungen des Roboters `indi` zuständig.

In einer Endlosschleife wird zunächst die momentane Bewegungsrichtung des Roboters über die Methode `indi.getDirection()` abgefragt. Im Fall von `direction` == `indi`.HALT werden über die Methode `stop()` die Motoren des Kettenantriebs blockiert. Die Abläufe für die übrigen Bewegungsrichtungen - vorwärts und rückwärts Fahren, links- und rechtherum Drehen - sind einander recht ähnlich: Die Motoren werden in die entsprechende Richtung gedreht, der Thread schläft für 100 Millisekunden (die Motoren laufen währenddessen weiter), dann wird überprüft, ob der Roboter an ein Hindernis gestoßen ist. Dies geschieht mit Hilfe der Methode `indi.getColl(int pos)`, wobei `pos` einen der drei Berührungssensoren bezeichnet: COLL_LEFT, COLL_RIGHT oder COLL_BACK.

Im Falle einer Kollision wird als Reaktion ein Ausweichmanöver eingeleitet. So sind zum Beispiel für den Zustand der Vorwärtsbewegung folgende Manöver vorgesehen: Bei frontalem Aufprall - Berührungssensoren vorne links und vorne rechts gedrückt - fahre ein Stückchen zurück und bleibe stehen. Erfolgt die Kollision einseitig vorne links, fahre ein Stückchen zurück, drehe dich ein wenig nach rechts und versuche erneut, vorwärts zu fahren. Entsprechend erfolgt bei einem Hindernis vorne rechts eine Korrekturbewegung nach links.

Die kurzen Ausweichbewegungen werden durch Aufruf der Methode `go(int direction, int units)` durchgeführt. Diese nutzt über die Methode `indi.getTime()` den in Kapitel 5.1 erwähnten Timer, um den Roboter genau `units` Zehntelsekunden in die gewünschte Richtung zu bewegen. Melden in dieser Phase die Berührungssensoren ein Hindernis, so wird kein weiteres Ausweichmanöver gestartet, stattdessen fährt der Roboter direkt mit der Bewegung in der ursprünglichen Richtung (`indi.getDirection()`) fort.

Die Methode `go(int direction, int units)` wird auch genutzt, um den Roboter direkt eine kurze Strecke in eine bestimmte Richtung fahren zu lassen. Über den *MessageProcessor* und den *IndianaMessageHandler* (Kapitel 5.2) muss hierzu nur eine einzige mehrstellige Integer-Zahl an die Methode `go(int unidir)` übergeben werden. Diese interpretiert die letzte Ziffer von `unidir` (`unidir` % 10) als Richtungsangabe und `unidir` ohne die letzte Ziffer (`unidir` / 10) als Fahrzeit in Zehntelsekunden, mit denen wiederum `go(int direction, int units)` aufgerufen wird.

5.3.1 Kollisionsüberwachung: IndianaCollisionListener

Für die Erkennung und Meldung von Kollisionen ist die Klasse *IndianaCollisionListener* (Programmcode in Anhang A-1.4) zuständig. Eine Instanz dieser Klasse wird im Konstruktor von *Indiana* kreiert und `collisionSensor` zugewiesen. Dem Konstruktor von *IndianaCollisionListener* wird dazu neben `indi` der raw-Wert übergeben, welcher am Sensoreingang gemessen wird, wenn keiner der drei Berührungssensoren gedrückt ist. Laut Tabelle 1 liegt dieser bei 469. Mit Hilfe der Formel `rMax` = 10000 * `rawValue` / (1023 - `rawValue`) wird daraus der maximal auftretende Widerstand berechnet und gespeichert. Es schien mir sinnvoll, diesen nicht fest vorzugeben, da sich der Gesamtwiderstand durch lange Kabel und Lötstellen erhöhen kann und in diesem Fall der tatsächliche raw-Wert im Ausgangszustand nur im Konstruktor von *Indiana* geändert werden muss[13].

Das Interface *SensorListener*, welches von *IndianaCollisionListener* implementiert wird, sorgt dafür, dass bei jeder Änderung des raw-Wertes am Eingang des `collisionSensor` die Methode `stateChanged()` aufgerufen wird. Da kleine Schwankungen dieses raw-Wertes allerdings im Normalbetrieb ständig auftreten, wird hier zusätzlich überprüft, ob der neue Wert um mehr als 10 vom zuletzt aktuellen `rawValue` abweicht. Ist dies der Fall, wird der neue Wert gespeichert und der Gesamtwiderstand `rGes` der momentan aktiven Kollisions-Detektoren berechnet. Dieser entspricht der Differenz aus dem maximal auftretenden Widerstand `rMax` und dem aktuell am Multiplexer anliegenden[14].

Wird dieser Wert nun noch durch 500 geteilt (dies entspricht ungefährt dem kleinsten der Widerstände in Ω), so lässt sich per modulo und Integer-Division recht einfach ausrechnen, welche der Berührungssensoren gedrückt sind und welche nicht. Diese Ergebnisse werden über die Methode `indi.setColl(int pos, boolean coll)` an die Instanz der Hauptklasse weitergereicht. Wie weiter oben beschrieben, können nun über die Methode `indi.getColl(int pos)` während der Bewegung des Roboters Kollisionen erkannt und entsprechend reagiert werden.

Da verschiedene Threads über die `set`- und `get`-Methoden die Felder `direction`, `camPos`, `camDirection`, `zeit` und `collision[]` der Roboter-Instanz `indi` auslesen bzw. verändern, sind diese Methoden als *synchronized* deklariert. Ein Thread muss also zunächst eine Sperre auf das Objekt `indi` erwerben, bevor er eine dieser Methoden aufrufen kann. Während dieser Zeit kann kein anderer Thread eine ebenfalls als *synchronized* deklarierte Methode dieses Objektes aufrufen. Statt dessen muss auf die Freigabe der Sperre durch den ersten Thread gewartet werden.

5.4 Kamerasteuerung: IndianaMoveCamera

Der zweite Thread `moveCamera`, welcher in der main-Methode von *Indiana* gestartet wird, ist eine Instanz der Klasse *IndianaMoveCamera* (Programmcode in Anhang A-1.3). Bei der Instantiierung werden zunächst die Kameraposition `camPos` und aktuelle Bewegungsrichtung des Kameraarms `camDirection` mit Hilfe der Methoden `indi.getCamPos()` und `indi.getCamDirection()` ausge-

[13] Natürlich wäre es auch interessant, "Indi" selbst diesen Ausgangswert ausmessen zu lassen. In diesem Moment dürften die Kollisionsdetektoren allerdings nicht feuern, worauf sich der Roboter ohne menschliche Hilfe aber nicht verlassen kann.

[14] Zum Aufbau und der Funktionsweise des Multiplexers siehe Kapitel 4.1.

lesen. Die Kameraposition wird hier über die Drehung desRotationssensors in Einheiten von 90 Grad gemessen und läuft von `indi.MIN_CAM_POS`=0 (Kamera an tiefster Position) bis `indi.MAX_CAM_POS`=48 (Kamera an höchster Position).

Außerdem wird dem Sensoreingang `indi.camSensor` eine Instanz der Klasse *IndianaCamListener* als Sensor-Listener zugewiesen. Dessen Aufgabe ist es, den Rotationssensor zu überwachen und viermal pro Umdrehung darauf wartende Threads mit Hilfe von `indi.camSensor.notifyAll()` aufzuwecken, wofür die abfallende Flanke des Sensorsignals (der Übergang von raw-Wert >= 1000 zu raw-Wert < 1000) genutzt wird. Hier wäre auch eine noch feinere Abstufung möglich, da sich der raw-Wert am Rotationssensor sechszehnmal pro Umdrehung ändert, was in Abb. 8 verdeutlicht wird. In diesem raw-Modus kann der Rotationssensor als passiver Sensor genutzt werden, eine eigene Stromversorgung ist somit nicht notwendig.

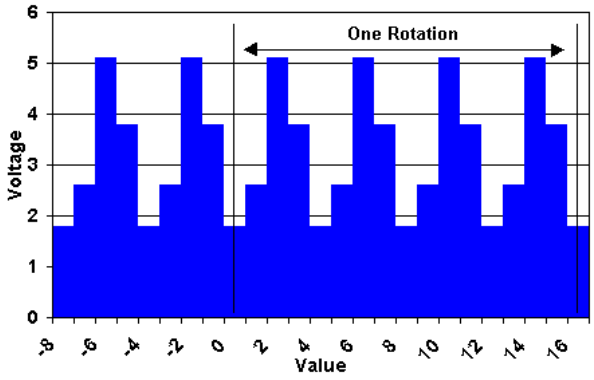


Abbildung 8: Der Spannungsverlauf am Rotationssensor

Nach dem Starten des Threads `moveCamera` wird innerhalb der Methode `run()` eine Endlosschleife durchlaufen. In dieser wird ständig die aktuell gewünschte Bewegung des Kameraarms abgefragt, bei Abweichung von `indi.CAM_HALT` über den Aufruf `moveCam(camDirection)` die entsprechende Bewegung ausgeführt und danach der Kameramotor wieder angehalten.

Die Bewegung des Kameraarms geschieht in zwei Schritten. Zunächst wird in der Methode `moveCam(int updown)` überprüft, ob die Bewegung an die höchste (`updown == indi.CAM_UP`) oder tiefste (`updown == indi.CAM_DOWN`) Kameraposition gewünscht ist. Ist dies der Fall, so wird über den Aufruf der Methode `moveCam(int updown, int units)` eine Bewegung um die nötigen Rotationssensor-Einheiten in die entsprechende Richtung veranlasst. Ansonsten werden die letzte Ziffer von `updown` als Bewegungsrichtung (`indi.CAM_UP` oder `indi.CAM_DOWN`), die übrigen Ziffern als Rotationssensor-Einheiten interpretiert und ebenfalls `moveCam(int updown, int units)` aufgerufen. Zur einfacheren Steuerung wird diese Zahl noch mit sechs multipliziert, die resultierende Einheit ist somit die anderthalbfache Rotationssensorumdrehung: 6 * 90 Grad.

In der Methode `moveCam(int updown, int units)` wird zunächst überprüft, ob die gewünschte Bewegung innerhalb des Bereichs von

`indi.MIN_CAM_POS` bis `indi.MAX_CAM_POS` möglich ist. Ist dies nicht der Fall, so wird `units` entsprechend vermindert. Die eigentliche Bewegung erfolgt dann in Schritten von 90 Grad (siehe Erläuterungen zum Rotationssensor weiter oben). Dazu wird der Motor in die gewünschte Richtung gestartet und `waitSensorUnits(indi.camSensor, units)` aufgerufen. Dadurch wird über `indi.camSensor.wait()` der Sensoreingang genau `units`-mal "schlafen" geschickt und vom Sensor-Listener jeweils nach einer Drehung des Rotationssensors um 90 Grad wieder "aufgeweckt". Ist die gewünschte Position erreicht, wird der `camMotor` angehalten und die neue Kamera-Position über `indi.setCamPos(camPos)` an die Roboter-Instanz übergeben.

6 Schlussbemerkungen

Die Präsentation unseres pyramidenerkundenden Roboters verlief sehr zufriedenstellend. Einige ursprünglich angedachte Punkte konnten jedoch nicht realisiert werden. So war z.B. geplant, Anfragen bezüglich der Breite eines Ganges an den Roboter stellen zu können und mit Hilfe eines Farb- und/oder Entfernungssensors Informationen über Farbe bzw. Entfernung von Gegenständen zu erfragen. Eine andere mögliche Erweiterung wäre die Option, Gegenständen in der Pyramide bei ihrer "Entdeckung" Namen wie *Objekt 1* oder *Sarkophag* zu geben. Der Roboter würde sich diese Bezeichnungen und die Positionen der Objekte merken und könnte später dorthin zurückgeschickt werden. Dies hätte jedoch eine äußerst exakte Navigation innerhalb der Pyramide zur Voraussetzung gehabt, was im Widerspruch zur Anforderung stand, sperrige Hindernisse überwinden zu können. Denn bei solchen Manövern kommt es durchaus vor, dass auch ein kettengetriebener Roboter abrutscht oder sich die eine Antriebskette schneller dreht als die andere. Das Wiederfinden eines bekannten Gegenstandes nach ausschweifenden Erkundungsgängen wäre somit kaum zu realisieren.

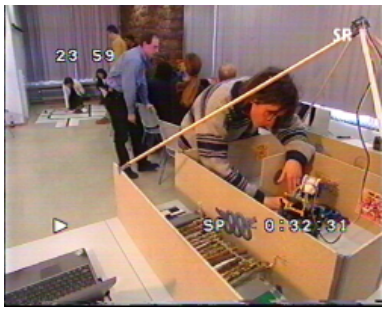


Abbildung 9: Indi kommt ganz groß raus: *Aktueller Bericht* des SR vom 25.02.03

Letztendlich gab es noch die Idee, die mitgeführte Kamera nicht bloß zur Bildübertragung vom Inneren der Pyramide nach draußen zu nutzen, sondern über eine Bilderkennungssoftware auch zur Identifizierung von Objekten, Farben und Bewegungen. Erste Ansätze zur Realisierung dieser Erweiterung wurden unternommen, jedoch waren die Ergebnisse nicht spektakulär genug für die Abschlusspräsentation. Vielleicht beim nächsten Mal...

ANHANG

A-1 Die Java-Klassen auf dem RCX

A-1.1 Indiana.java

```
import josx.platform.rcx.*;
import josx.rcxcomm.RCXPort;
import josx.util.Timer;

public class Indiana {

    // constants for moving indi
    static final int HALT = 0;
    static final int GO_FORWARD = 1;
    static final int GO_BACKWARD = 2;
    static final int GO_LEFT = 3;
    static final int GO_RIGHT = 4;

    // constants for moving camera
    static final int CAM_HALT = 0;
    static final int CAM_UP = 1;
    static final int CAM_DOWN = 2;
    static final int MIN_CAM_POS = 0;
    static final int MAX_CAM_POS = 48;

    // constants for collisions
    static final int COLL_LEFT = 0;
    static final int COLL_RIGHT = 1;
    static final int COLL_BACK = 2;

    // motors: left, right, camera
    static Motor lMotor = Motor.C;
    static Motor rMotor = Motor.A;
    static Motor camMotor = Motor.B;

    // sensors for collisions & camera (rotation sensor)
    static Sensor collisionSensor = Sensor.S1;
    static Sensor camSensor = Sensor.S2;

    private int direction, camPos, camDirection;
    private boolean[] collision;

    // timer for moving some deciseconds
    Timer tim;
    private int zeit;

    /** constructor for Indiana **/
    public Indiana() {
```

```
        // set type and mode of sensors, activate them
        collisionSensor.setTypeAndMode(SensorConstants.SENSOR_TYPE_TOUCH, \
SensorConstants.SENSOR_MODE_RAW);
        collisionSensor.activate();
        collisionSensor.addSensorListener(new \
IndianaCollisionListener(this, 469));
        camSensor.setTypeAndMode(SensorConstants.SENSOR_TYPE_TOUCH, \
SensorConstants.SENSOR_MODE_RAW);
        camSensor.activate();

        // instantiating & starting of timer calling timedOut()
        // of new IndianaTimerListener-Object every decisecond
        // which increments zeit
        zeit = 0;
        tim = new Timer(100, new IndianaTimerListener(this));
        tim.start();

        // power of motors
        lMotor.setPower(5);
        rMotor.setPower(5);
        camMotor.setPower(3);

        // start values: indi not moving,
                         camera on bottom & not moving
        direction = HALT;
        camPos = MIN_CAM_POS;
        camDirection = CAM_HALT;

        // starting with no collisions
        collision = new boolean[3];
        collision[COLL_LEFT] = false;
        collision[COLL_RIGHT] = false;
        collision[COLL_BACK] = false;
    }

    public static void main (String args[]) {

        RCXPort port;
        Indiana indi;
        IndianaMoveIndi moveIndi;
        IndianaMoveCamera moveCamera;
        IndianaMessageHandler imh;
        MessageProcessor mp;

        try {
            port = new RCXPort();
            indi = new Indiana();
            moveIndi = new IndianaMoveIndi(indi);
            moveCamera = new IndianaMoveCamera(indi);
```

```
            imh = new IndianaMessageHandler(indi);
            mp = new MessageProcessor(port, imh);

            // start thread for moving indi
            moveIndi.start();
            // start thread for moving camera
            moveCamera.start();
            // start thread of message processor
            mp.start();
        }
        catch(Exception e) {
        }
    }

    /** returns current movement direction of robot **/
    public synchronized int getDirection() {
        return direction;
    }

    /** sets movement direction of robot **/
    public synchronized void setDirection(int direction) {
        this.direction = direction;
    }

    /** returns current camera position **/
    public synchronized int getCamPos() {
        return camPos;
    }

    /** sets camera position **/
    public synchronized void setCamPos(int camPos) {
        this.camPos = camPos;
    }

    /** returns current movement direction of camera **/
    public synchronized int getCamDirection() {
        return camDirection;
    }

    /** sets movement direction of camera **/
    public synchronized void setCamDirection(int camDirection) {
        this.camDirection = camDirection;
    }

    /** returns time after creating
        this Indiana-Object (in deciseconds) **/
    public synchronized int getTime() {
        return zeit;
    }
```

```
    /** used by IndianaTimerListener for increasing timer **/
    public synchronized void incrTime() {
        zeit = zeit + 1;
    }

    /** returns if there is a collision at the specified position
        (position might be COLL_LEFT, COLL_RIGHT, COLL_BACK) **/
    public synchronized boolean getColl(int pos) {
        return collision[pos];
    }

    /** sets boolean variables for collision at the specified position
        (position might be COLL_LEFT, COLL_RIGHT, COLL_BACK) **/
    public synchronized void setColl(int pos, boolean coll) {
        collision[pos] = coll;
    }
}
```

A-1.2 IndianaMoveIndi.java

```
import josx.platform.rcx.Sensor;

public class IndianaMoveIndi extends Thread {

    private Indiana indi;
    private int direction;

    /** constructor **/
    public IndianaMoveIndi(Indiana indi) {
        this.indi = indi;
        this.direction = indi.getDirection();
    }

    /** thread gets startet by main method of class Indiana **/
    public void run() {
        try {
            while (true) {

                // get current movement direction of robot
                direction = indi.getDirection();

                // stop
                if (direction == indi.HALT) {
                    stop();
                }
                // moving forward
                else if (direction == indi.GO_FORWARD) {
                    indi.rMotor.forward();
                    indi.lMotor.forward();
                    Thread.sleep(100);
                    // collision front left
                    if (indi.getColl(indi.COLL_LEFT)) {
                        // collision front left & front right
                        if  (indi.getColl(indi.COLL_RIGHT)) {
                            // go back and stop
                            go(indi.GO_BACKWARD,3);
                            indi.setDirection(indi.HALT);
                        }
                        // collision only front left
                        else {
                            // go back and turn right
                            go(indi.GO_BACKWARD,2);
                            go(indi.GO_RIGHT,1);
                        }
                    }
                    // collision only front right
                    else if (indi.getColl(indi.COLL_RIGHT)) {
```

```
        // go back and turn left
        go(indi.GO_BACKWARD,2);
        go(indi.GO_LEFT,1);
    }
}
// moving backward
else if (direction == indi.GO_BACKWARD) {
    indi.rMotor.backward();
    indi.lMotor.backward();
    Thread.sleep(100);
    // collision back
    if (indi.getColl(indi.COLL_BACK)) {
        // go forward and stop
        go(indi.GO_FORWARD,2);
        indi.setDirection(indi.HALT);
    }
}
// turning left
else if (direction == indi.GO_LEFT) {
    indi.rMotor.forward();
    indi.lMotor.backward();
    Thread.sleep(100);
    // collision front left
    if (indi.getColl(indi.COLL_LEFT)) {
        // turn right and go backward
        go(indi.GO_RIGHT,1);
        go(indi.GO_BACKWARD,2);
    }
    // collision back
    else if (indi.getColl(indi.COLL_BACK)) {
        // turn right and go foward
        go(indi.GO_RIGHT,1);
        go(indi.GO_FORWARD,2);
    }
}
// turning right
else if (direction == indi.GO_RIGHT) {
    indi.rMotor.backward();
    indi.lMotor.forward();
    Thread.sleep(100);
    // collision front right
    if (indi.getColl(indi.COLL_RIGHT)) {
        // turn left and go backward
        go(indi.GO_LEFT,1);
        go(indi.GO_BACKWARD,2);
    }
    // collision back
    else if (indi.getColl(indi.COLL_BACK)) {
        // turn left and go forward
        go(indi.GO_LEFT,1);
```

```
                        go(indi.GO_FORWARD,2);
                    }
                }
                // for moving some deciseconds
                else {
                    go(direction);
                    indi.setDirection(indi.HALT);
                }
                // stop motors
                stop();
            }
        }
        catch (InterruptedException ie) {
        }
    }

    // methods for moving the robot

    /** stop motors **/
    private void stop() {
        indi.rMotor.stop();
        indi.lMotor.stop();
    }

    /** for moving some deciseconds
        (last digit for direction,
        first ones for number of deciseconds) **/
    private void go(int unidir) throws InterruptedException {
        go((unidir % 10), (unidir / 10));
    }

    /** for moving some deciseconds -
         e.g. for reactions after collision **/
    private void go(int direction, int units) throws InterruptedException {

        int startTime = indi.getTime();
        int aktTime = indi.getTime();

        // moving forward
        if (direction == indi.GO_FORWARD) {
            indi.rMotor.forward();
            indi.lMotor.forward();
            // move until collision left or right or time is over
            while ((aktTime - startTime < units) &&  \
(!indi.getColl(indi.COLL_LEFT)) && (!indi.getColl(indi.COLL_RIGHT))) {
                aktTime = indi.getTime();
            }
        }
        // moving backward
        else if (direction == indi.GO_BACKWARD) {
```

```
            indi.rMotor.backward();
            indi.lMotor.backward();
            // move until collision in the back or time is over
            while ((aktTime - startTime < units) && \
(!indi.getColl(indi.COLL_BACK))) {
                aktTime = indi.getTime();
            }
        }
        // turning left
        else if (direction == indi.GO_LEFT) {
            indi.rMotor.forward();
            indi.lMotor.backward();
            // move until collision left or in the back or time is over
            while ((aktTime - startTime < units) && \
(!indi.getColl(indi.COLL_LEFT)) && (!indi.getColl(indi.COLL_BACK))) {
                aktTime = indi.getTime();
            }
        }
        // turning right
        else if (direction == indi.GO_RIGHT) {
            indi.rMotor.backward();
            indi.lMotor.forward();
            // move until collision right or in the back or time is over
            while ((aktTime - startTime < units) && \
(!indi.getColl(indi.COLL_RIGHT)) && (!indi.getColl(indi.COLL_BACK))) {
                aktTime = indi.getTime();
            }
        }
        // stop motors
        stop();
    }
}
```

A-1.3 IndianaMoveCamera.java

```
import josx.platform.rcx.Sensor;

public class IndianaMoveCamera extends Thread {

    private Indiana indi;
    private int camPos, camDirection;

    /** constructor **/
    public IndianaMoveCamera(Indiana indi) {
        this.indi = indi;
        camPos = indi.getCamPos();
        camDirection = indi.getCamDirection();
        indi.camSensor.addSensorListener(new IndianaCamListener());
    }

    /** thread gets startet by main method of class Indiana **/
    public void run() {
        try {
            while (true) {
                // get current movement direction of camera
                camDirection = indi.getCamDirection();
                // move camera
                if (camDirection != indi.CAM_HALT) {
                    moveCam(camDirection);
                    indi.setCamDirection(indi.CAM_HALT);
                }
                // stop moving camera
                stop();
            }
        }
        catch (InterruptedException ie) {
        }
    }

    /** stop camera motor **/
    private static void stop() {
        Indiana.camMotor.stop();
    }

    /** move camera **/
    private void moveCam(int updown) throws InterruptedException {
        // move camera to top position
        if (updown == indi.CAM_UP) {
            moveCam(updown, indi.MAX_CAM_POS - camPos);
        }
        // move camera to lowest position
        else if (updown == indi.CAM_DOWN) {
```

```
            moveCam(updown, camPos);
        }
        // move camera some rotation sensor units
        else {
            moveCam((updown % 10), (updown / 10) * 6);
        }
    }

    /** move camera some rotation sensor units **/
    private void moveCam(int updown, int units) throws InterruptedException {
        // move camera upwards
        if (updown == indi.CAM_UP) {
            if (camPos + units > indi.MAX_CAM_POS) units = \
indi.MAX_CAM_POS - camPos;
            if (units > 0) {
                indi.camMotor.forward();
                waitSensorUnits(indi.camSensor, units);
                indi.camMotor.stop();
                camPos += units;
                indi.setCamPos(camPos);
            }
        }
        // move camera downwards
        else {
            if (camPos - units < 0) units = camPos;
            if (units > 0) {
                indi.camMotor.backward();
                waitSensorUnits(indi.camSensor, units);
                indi.camMotor.stop();
                camPos -= units;
                indi.setCamPos(camPos);
            }
        }
    }

    /** for counting of rotation sensor units when moving camera **/
    private void waitSensorUnits(Sensor s, int units) \
throws InterruptedException {
        synchronized(s) {
            do {
                try {
                    s.wait();
                }
                catch (Exception e) {
                }
            } while (--units > 0);
        }
    }
}
```

A-1.4 IndianaCollisionListener.java

```
import josx.platform.rcx.Sensor;
import josx.platform.rcx.SensorListener;

class IndianaCollisionListener implements SensorListener {

    private Indiana indi;
    private float rMax ;
    private int rawValue, rGes;
    private boolean[] coll;

    /** constructor (rawValue is initial raw value
                     for no touch sensor pressed) **/
    IndianaCollisionListener(Indiana indi, int rawValue) {
        this.indi = indi;
        // calculate maximum resistance (no touch sensor pressed)
        rMax = 10000 * rawValue / (1023 - rawValue);
        this.rawValue = rawValue;
        // difference between maximum resistance and current one
        rGes = 0;
        coll = new boolean[3];
    }

    /** reaction if difference between old and
        new raw value is bigger then 10 -
        checks if some collision occured,
        saves this information and
        wakes up all threads waiting for a collision **/
    public void stateChanged (Sensor s, int egal, int rawValue) {

        if (Math.abs(rawValue - this.rawValue) > 10) {

            this.rawValue = rawValue;
            // difference between maximum resistence and
               current one (for easy counting: in 500 Ohm)
            rGes = Math.round((rMax - \
(10000 * rawValue / (1023 - rawValue))) / 500);
            if (rGes  < 0) rGes = 0;

            // check in which positions collisions ocurred
            // and save information about collisions
            indi.setColl(indi.COLL_LEFT, ((rGes / 4) == 1));
            indi.setColl(indi.COLL_RIGHT, (((rGes % 4) % 2) == 1));
            indi.setColl(indi.COLL_BACK, (((rGes % 4) / 2) == 1));
        }
    }
}
```

A-1.5 IndianaCamListener.java

```
import josx.platform.rcx.Sensor;
import josx.platform.rcx.SensorListener;

class IndianaCamListener implements SensorListener {

    /** wakes up all threads that are waiting on the rotation sensor
        of the robot's camera every 45 degrees of turning **/
    public void stateChanged (Sensor s, int oldVal, int newVal) {

        if ((oldVal >= 1000) && (newVal < 1000)) {
            synchronized(s) {
                s.notifyAll();
            }
        }
    }
}
```

A-1.6 IndianaTimerListener.java

```
import josx.util.TimerListener;

public class IndianaTimerListener implements TimerListener {

    private Indiana indi;

    /** constructor **/
    IndianaTimerListener(Indiana indi) {
        this.indi = indi;
    }

    /** calls robot's method for increasing timer **/
    public void timedOut() {
        indi.incrTime();
    }
}
```

A-1.7 IndianaMessageHandler.java

```
class IndianaMessageHandler implements MessageHandler {

    private Indiana indi;

    /** constructor **/
    public IndianaMessageHandler(Indiana indi) throws Exception {
        this.indi = indi;
    }

    /** handles an incoming message and constructs a reply
        (incoming messages can be for moving robot or camera) **/
    public void process (DialogMessage in, DialogMessage out) {

        out.Type = DialogMessage.TYPE_ok;

        // message for moving robot
        if (in.Type == DialogMessage.TYPE_gehe ) {
            try {
                // set new movement direction of robot
                indi.setDirection(in.richtung);
                out.status = 0;
            }
            catch (Exception e) {
                // some error occured
                out.status = -1;
            }
        }
        // message for moving camera
        else if (in.Type == DialogMessage.TYPE_bewegearm ) {
            try {
                // set new movement direction of camera
                indi.setCamDirection(in.armrichtung);
                out.status = 0;
            }
            catch (Exception e) {
                // some error occured
                out.status = -1;
            }
        }
    }
}
```

A-1.8 MessageProcessor.java

```
/**
 * @file   MessageProcessor.java
 * @author Alexander Koller
 * @date   Wed Jun 26 17:48:24 2002
 */

import josx.rcxcomm.*;
//import josx.platform.rcx.*;

/** Communication and concurrency details for RCX-side message transport.
 *
 * You still need to define a MessageHandler object to deal with
 * the messages your program receives.
 *
 * After you've created a new instance mp of this class, you need
 * to start it with mp.start(). If you want to terminate the thread
 * it implicitly creates, call mp.interrupt(). This will terminate
 * AFTER THE NEXT MESSAGE HAS BEEN RECEIVED.
 *
 * This class is for the RCX.
 */
class MessageProcessor extends Thread {
    private DialogMessage in, out;
    private RCXPort port;
    private MessageHandler handler;

    MessageProcessor(RCXPort port, MessageHandler handler) {
        this.port = port;
        this.handler = handler;

        // dummy values; all that counts here is that
        // they're properly allocated instances of DialogMessage.
        in = new DialogMessage((byte)0);
        out = new DialogMessage((byte)0);
    }

    /** Receive a message from the port, process it, and
     *  send the result back to the port. */
    public void run() {
        while(!interrupted()) {
            in.receive(port);
            if(!isInterrupted()) handler.process(in,out);
            if(!isInterrupted()) out.send(port);
        }
    }
}
```

A-1.9 DialogMessage.java

```
/**
 * @file   DialogMessage.java
 * @date   07.05.2003 17:32:41
 * @brief  Encapsulates messages exchanged between dialog system and RCX.
 * (This version is intended for use on the RCX.)
 * This is an automatically generated file -- do not edit by hand!
 */

import josx.rcxcomm.*;
import java.io.*;

/** Encapsulates messages exchanged between dialog system and RCX.
 *
 * This class is used both on the RCX and on the PC. It contains
 * the type of message and the arguments it might have, and defines
 * methods for sending and receiving messages through the IR port,
 * and for printing the message for debugging purposes (on the PC).
 *
 * Every message has a certain type, which determines which
 * arguments the message uses. Each argument has a globally defined
 * data type (e.g. int or String). They're read through accessor
 * functions which throw exceptions if the respective argument
 * isn't defined for the message type.
 */
public class DialogMessage {

    // enum for message types
    public static final byte TYPE_gehe = 0;
    public static final byte TYPE_bewegearm = 1;
    public static final byte TYPE_status = 2;

    // internal data fields
    public byte Type;           /**< The message type. */
    public int richtung;        /**< Bewegungsrichtung des Roboters */
    public int armrichtung;     /**< Bewegungsrichtung des Kameraarms */
    public int okay;            /**< Statusargument */

    /** Instantiate message by reading data from the IR port.
     * @param port The port from which to read the data. */
    DialogMessage(RCXPort port) {
        receive(port);
    }

    /** Generic constructor that doesn't specify any arguments (yet).
     * Note that if the message requires further arguments,
     * the message is invalid until you set the argument values
     * explicitly, but you have no way to tell. Therefore usage
```

```
 * of this constructor should be avoided, unless the message
 * has no arguments.
 * @param type Type of the message.
 */
DialogMessage(byte type) {
    this.Type = type;
}

// Communication functions

/** Send message to the IR port.
 * @param port The port to which to send the data.
 * @return true iff the communication was successful
 */
boolean send(RCXPort port) {
    OutputStream os = port.getOutputStream();
    try {
        os.write(Type);
        if( Type == TYPE_gehe  ) {
            write_int(os, richtung);
        }
        else if( Type == TYPE_bewegearm  ) {
            write_int(os, armrichtung);
        }
        else if( Type == TYPE_status  ) {
            write_int(os, okay);
        }
        else {
        }
        return true;
    } catch(IOException e) {
        return false;
    }
}

/** Receive message from the IR port.
 * This is similar to the constructor that builds a message
 * object from data received through the port, but doesn't
 * create a new object.
 *
 * This method, like the set_* accessor functions,
 * is a bit hacky, but necessary because Lejos on the RCX
 * doesn't implement garbage collection yet. For this reason,
 * we have to be able to reuse the same message object
 * for different messages.
 *
 * @param port The port to which to send the data.
 * @return true iff communication was successful
 */
boolean receive(RCXPort port) {
```

```
    InputStream is = port.getInputStream();
    try {
        Type = (byte)is.read();
        if( Type == TYPE_gehe  ) {
            richtung = read_int(is);
        }
        else if( Type == TYPE_bewegearm  ) {
            armrichtung = read_int(is);
        }
        else if( Type == TYPE_status  ) {
            okay = read_int(is);
        }
        else {
        }
        return true;
    } catch(IOException e) {
        return false;
    }
}

// low-level communication functions

/** Send an atom to an output stream. */
private void write_atoms(OutputStream os, int a) throws IOException {
    os.write(a);
}

/** Send an int to an output stream.
 *
 * This method sends one int to an OutputStream. It
 * is defined only for symmetry with the write_* functions
 * for the other data types.
 *
 * @param os The OutputStream to which to write.
 * @param b The byte to write to the OutputStream.
 */
private void write_int(OutputStream os, int b) throws IOException {
    os.write(b);
}

/** Send a string to an output stream.
 *
 * This method sends a string to an OutputStream, character by character.
 *
 * @param os The OutputStream to which to write.
 * @param s The string to write to the OutputStream.
 */
private void write_string(OutputStream os, String s) throws IOException {
    os.write(s.length());
    for(int i = 0; i < s.length(); i++)
```

```
            os.write(s.charAt(i));
    }

    /** Read a string from an input stream.
     * This method reads a string from an InputStream, character
     * by character.
     *
     * @param is The InputStream from which to read.
     * @return The received string, or null if an error occurred.
     */
    private String read_string(InputStream is) {
        int len;
        char[] bytes;
        try {
            len = is.read();
            bytes = new char[len];
            for(int i = 0; i < len; i++) bytes[i] = (char) is.read();
            return new String(bytes, 0, len);
        } catch (IOException e) {
            return null;
        }
    }

    /** Read an int from an input stream.
     * This method reads one int from an InputStream. It
     * is defined only for symmetry with the read_* functions
     * for the other data types.
     *
     * @param is The OutputStream from which to read the int.
     * @return The int that was received.
     * @todo Exception probably not treated well.
     */
    private int read_int(InputStream is) {
        try {
            return is.read();
        } catch(IOException e) {
            return -1;
        }
    }

    private int read_atoms(InputStream is) {
        try {
            return is.read();
        } catch(IOException e) {
            return -1;
        }
    }
}
```

A-2 Die Nachrichtenspezifikation

A-2.1 indianamsg.xml

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE messages SYSTEM "message.dtd">

<messages>
    <argdecl>
        <name>richtung</name>
        <datatype><int/></datatype>
        <description> Bewegungsrichtung des Roboters:
            0 halt, 1 vor, 11 vor1, 2 zurueck, 12 zurueck1,
            3 links, 13 links15, 63 links90, 123 links180,
            4 rechts, 14 rechts15, 64 rechts90, 124 rechts180
        </description>
    </argdecl>

    <argdecl>
        <name>armrichtung</name>
        <datatype><int/></datatype>
        <description> Bewegungsrichtung des Kameraarms:
            1 hoch, 2 runter, 11 hoch1, 12 runter1
        </description>
    </argdecl>

    <argdecl>
        <name>okay</name>
        <datatype><int/></datatype>
        <description>Statusargument</description>
    </argdecl>

    <message>
        <name>gehe</name>
        <description>Steuerung des Roboters</description>
        <argument>richtung</argument>
    </message>

    <message>
        <name>bewegearm</name>
        <description>Steuerung des Kameraarms</description>
        <argument>armrichtung</argument>
    </message>

    <message>
        <name>status</name>
        <description>Statusnachricht</description>
        <argument>okay</argument>
    </message>
</messages>
```

Literatur

Jonathan Knudsen. *The Unofficial Guide to LEGO MINDSTORMS Robots.* O'Reilly & Associates, Inc., 1999.