Inhaltsverzeichnis

1 Vorwort

Warum dieses Thema?

Die Grundidee meiner Maturarbeit ist es, herauszufinden, wie ein Roboter programmiert wird. Warum? Ich wollte lernen, ein Computerprogramm zu erstellen, ich finde es aber sehr interessant, wenn man nebst aufflimmernden Pixels, wie das bei den meisten PC-Programmen der Fall ist, ein „materielles“ Ergebnis hat.
Roboter haben mich nebenbei schon von klein an fasziniert, sei es jetzt der Industrieroboter, der eine monotone Arbeit ausführt, oder ein humanoider Roboter, der versucht, dem Menschen möglichst ähnlich zu sein.
Für meine Maturarbeit stellte ich dann eine passende Aufgabe:

Ich konstruiere einen sinnvollen Roboter und programmiere ihn mit Hilfe einer gängigen Programmiersprache. Dabei stelle ich mir verschiedene Teilaufgaben, die es zu lösen gilt.

2 Theoretischer Teil

2.1 Roboter, was ist das?

Den Begriff Roboter zu definieren ist nicht einfach, denn eine einheitliche Definition gibt es nicht. Das hängt vielleicht damit zusammen, dass der Begriff Roboter in sehr vielen verschiedenen Arten auftaucht und für verschiedene Zwecken verwendet wird. Ein Definitionsversuch könnte so lauten:

Ein mechanisches System, dessen Bewegungsfunktionen denen lebender Organismen entsprechen oder das die Bewegungsfunktion mit intelligenten Funktionen kombiniert und dem Willen des Menschen entsprechend handelt.[1]

Der weltweit am meisten verwendete Roboter ist der Industrieroboter. In der europäischen Norm EN775 wird der Industrieroboter wie folgt definiert:

Ein Roboter ist ein automatisch gesteuertes, wiederprogrammierbares, vielfach einsetzbares Handhabungsgerät mit mehreren Freiheitsgraden, das entweder ortsfest oder beweglich in automatisierten Fertigungssystemen eingesetzt wird.[2]

Nach VDI[3] Richtlinie 2860 wird er allgemein so definiert:

"Ein Roboter ist ein frei und wieder programmierbarer, multifunktionaler Manipulator mit mindestens drei unabhängigen Achsen, um Materialien, Teile, Werkzeuge oder spezielle Geräte auf programmierten, variablen Bahnen zu bewegen zur Erfüllung der verschiedensten Aufgaben." (VDI, 1990)[4]

Das Wort selbst stammt vom tschechischen Wort Robota, was soviel heisst wie Fronarbeit, oder vom russischen Wort Robota, das für Arbeit steht.

2.1.1 Geschichte

Schon in der Antike wurden roboterähnliche Ideen geäussert, es wurden beispielsweise Geschichten geschrieben, in denen aus Lehm gebaute Menschen lebendig wurden. Auch gab es schon früh Kinderspielzeuge, die roboterähnliche Züge hatten. 1921 wurden das erste Mal Roboter in der unserer Vorstellung entsprechenden Art und Weise in einem Theaterstück von Karel _apec namens RUR vorgestellt.

Der erste kommerziell relevante Industrieroboter war Unimates von der Firma Unimation, der durch eine magnetische Speichertrommel gesteuert wurde.

2.1.2 Aktuelles

Die Robotertechnik hat in den letzten Jahren grosse Fortschritte gemacht. Durch die Leistungssteigerung in der Mikrochiptechnologie und der Verkleinerung von Akkumulatoren sind immer intelligentere und kleinere Roboter realisierbar. Der Industrieroboter macht zwar heute noch immer die Mehrheit aller Roboter aus, doch ergeben sich für die Zukunft immer neue Perspektiven. Als besonders interessant gelten z. B. das Gebiet der Nanorobotik, das auch eine wichtige Rolle in der Medizin spielen wird, oder Roboter im Weltraum, die wegen ihren geringen Lebensbedürfnissen das Weltall erforschen werden.

---

[1] http://www.fh-landshut.de/~gschied/robotic/slide0006.html, 22.10.2004
[2] http://www.reisrobotics.de/robotech/default.htm, 22.10.2004
[3] Verein Deutscher Ingenieure
[4] http://www.robowelt.de/1tech/1wasist/1wasist.htm, 22.10.2004

2.1.3 Ethik

Dass Menschen durch Roboter sterben, ist nicht nur Sciencefiction. Durch ihre rasanten Bewegungen und ihre Anziehungskraft auf Menschen kommt es immer wieder zu Konflikten zwischen Mensch und Maschine, z. B. dann, wenn jemand den Arbeitsbereich eines Roboters betritt und verletzt wird. Schon die Einführung des vorhin erwähnten Unimates kostete mehreren Menschen das Leben. Der Robotiker John Kreifeldt formulierte daher die folgenden Regeln:

1. Kehre niemals einem Roboter den Rücken zu.

2. Wenn du dich einem Roboter näherst, dann schalte ihn aus, bevor er dich ausschaltet.

Vor über 45 Jahren formulierte der Sciencefiction-Autor Isaac Asimov drei Gesetze der Roboterethik, die bis heute unangefochten bestehen:

1. Ein Roboter darf kein menschliches Wesen verletzen und auch nicht durch Untätigkeit zulassen, dass ein menschliches Wesen verletzt wird.

2. Ein Roboter muss den vom Menschen gegebenen Befehlen gehorchen, es sei denn, sie kommen mit dem ersten Gesetz in Konflikt.

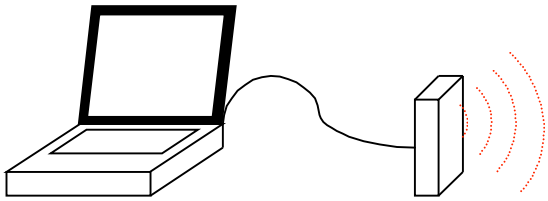
3. Ein Roboter muss seine eigene Existenz schützen, solange er dabei nicht mit dem ersten oder zweiten Gesetz in Konflikt gerät.[5]

Ob ein Roboter Arbeitsplätze vernichtet, ist zweifelhaft und man kann sich fragen, ob Arbeiten, die ein Roboter durchführt, überhaupt menschenwürdig sind. In Japan ist trotz geringer Arbeitslosigkeit die Roboterdichte achtmal höher als in den USA. Die Erfindung des Rades oder des Webstuhls kostete vielleicht auch einigen Arbeitern ihren Job, sie ermöglichte aber jedem einzelnen, mehr zu produzieren, was sich positiv auf die Wirtschaftsentwicklung ausübte.

[5] http://www.zeit.de/archiv/1997/03/roboter.txt.19970110.xml, 22.10.2004, aus: Gero von Randow, "Roboter. Unsere nächsten Verwandten"

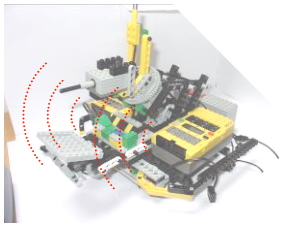
2.2 *Lego RCX 1.0 mit NQC Umgebung*

In meinem Roboterprojekt werde ich mit zwei Systemen arbeiten. Das erste System ist ein Lego RCX Baustein, den ich mit der von Dave Baum erfundenen Programmiersprache NQC (Not Quite C) programmieren werde. Er befolgt die Befehle des zweiten Systems, das ihm diese über Infrarot zuspielt. Das zweite System besteht aus einem PC und einem Steuerungsprogramm. Die Infrarotübertragung erfolgt über einen an einer seriellen Schnittstelle angeschlossenen Infrarot-Tower, kurz IR-Tower.



Laptop mit IR-Tower

Auf ihm läuft System 2.
Es sendet dem Roboter einen Auftrag und stellt, nachdem er diesen ausgefüllt hat, die erhaltenen Werte dar.

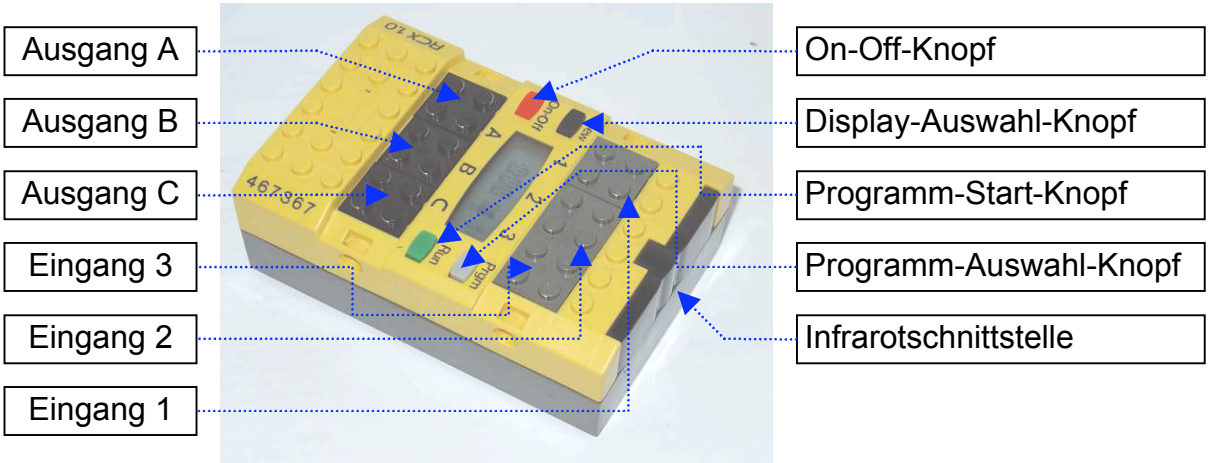


Roboter

Auf ihm läuft System 1.
Es versucht, den erhaltenen Auftrag auszuführen und die dabei ermittelten Werte dem PC zu senden.

Abb. 1

2.2.1 RCX Baustein



Kern des mobilen Roboters ist der Lego RCX Baustein. Seine Entwicklung geht auf den am MIT[6] entwickelten "MIT Programmable Brick" zurück. Sein Herz ist ein Renesas H8/300 Microcontroller als CPU. Er hat 32 KB Speicherplatz, die für das Betriebssystem BrickOS und bis zu fünf eigenen Programmen zur Verfügung stehen. Der RCX verfügt über 3 Ein- und 3 Ausgänge, an denen verschiedene Sensoren und Motoren angeschlossen werden können und über eine IR Schnittstelle, mit der er mit anderen RCX's oder mit einem PC kommunizieren kann. Er hat ein Display, über das er Informationen anzeigen, und einen Summer, über den er Töne abspielen kann. Die Energie hat er aus 6 AA-Batterien.

[6] Massachusetts Institut of Technologie

2.2.2 Motoren

Motoren werden an den drei Ausgängen A, B, C angeschlossen. Der RCX kann sie in acht verschiedenen Geschwindigkeiten in beiden Richtungen ansteuern. Neuere Varianten der Motoren verfügen über ein Getriebe.

2.2.3 Sensoren

Sensoren werden an den drei Eingängen 1, 2, 3 angeschlossen. Sie liefern dem RCX Zahlenwerte über seine Umgebung, so genannte Rohwerte im Bereich zwischen 0 und 1023. Von Lego werden folgende Sensoren angeboten:

- Lichtsensor
- Rotationssensor
- Berührungssensor (zwei verschiedene Ausführungen)
- Temperatursensor

Im Eigenbau können natürlich auch weitere Sensoren kreiert werden, zum Beispiel ein Distanzmesssensor.[7]

2.2.3.1 Lichtsensor

Lichtsensoren messen die Lichtintensität. Sie werden meistens im Modus *Percent* betrieben, wo die gemessene Helligkeit in Prozent ausgedrückt wird. Auch die Rohwerte werden manchmal verwendet, sie bieten die Möglichkeit einer genaueren Auswertung. Für eine schwarze Oberfläche liegen die Rohwerte bei ca. 800, für eine weisse bei 650. Die Werte verändern sich je nach Umgebungslicht.

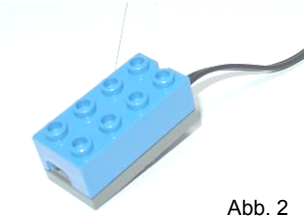


Abb. 2

2.2.3.2 Berührungssensor

Der Berührungssensor reagiert auf Druck. Mit ihm kann man z. B. Hindernisse feststellen. Standardmässig wird er im Modus *Bool* betrieben, d. h., ist er gedrückt, übermittelt er eine 1, sonst eine 0. Aber auch er liefert genauere Werte: der Rohwert beträgt 1023 im entspannten Zustand und sinkt mit steigendem Druck auf ca. 50.

Abb. 3

2.2.3.3 Temperatursensor

Der Temperatursensor ist ein Zusatzartikel, der nicht im Grundbaukasten enthalten ist. Er misst die Umgebungstemperatur. Meist verwendet man den Modus *Celsius*, wo die Temperatur in Grad Celsius angezeigt wird. Auch die Anzeige in Fahrenheit ist möglich.

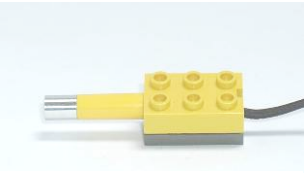


Abb. 4

2.2.3.4 Rotationssensor

Wie der Temperatursensor ist auch der Rotationssensor ein Zusatzartikel. Er ist sehr praktisch, wenn eine von einem Motor angetriebene Welle eine exakt bestimmte Anzahl Umdrehungen erreichen soll. Pro gedrehte 22.5° zählt der Rotationswert eins nach oben oder unten, je nach Drehrichtung.

Abb. 5

[7] http://users.informatik.haw-hamburg.de/~kvl/dahlmann/diplom.pdf, 22.10.2004, Diplomarbeit Oliver Dahlmann, Entwicklung eines kostengünstigen Distanzmesssystems für mobile Roboter.

2.2.3.5 Zwei Sensoren an einem Eingang

Der RCX hat bekanntlich nur drei Eingänge. Diese Einschränkung lässt sich mit einem Trick etwas vermindern. Es ist nämlich möglich, zwei Sensoren an einem Eingang anzuschliessen. Falls man dafür sorgt, dass nicht beide gleichzeitig verwendet werden, ist das noch nichts Besonders. Wie sieht es aber aus, wenn zwei Sensoren am selben Eingang gleichzeitig in Betreib sein sollten, wie z. B. ein Lichtsensor und ein Berührungssensor? Diese beiden lassen sich gut an einem Eingang anschliessen, nur muss einfach der Sensortyp auf Lichtsensor eingestellt werden (der Lichtsensor benötigt Strom, der Berührungssensor nicht) und der Interpretationsmodus *Raw* sein. Wie oben beschrieben, bewegen sich die Lichtsensorwerte zwischen sechshundert und achthundert, maximal zwischen fünfhundert und neunhundert. Die Werte des Berührungssensors liegen schon bei leichter Berührung unter 200. Mit einer if-Anweisung lassen sich die zwei Sensoren am selben Eingang unterscheiden. Da der Berührungssensorwert auch bei starkem Druck nicht unter 50 fällt, mit zwei Berührungssensoren am selben Eingang dies aber durchaus der Fall ist, lässt sich zusätzlich auch noch ein zweiter Berührungssensor anschliessen.

2.2.4 NQC

Mein Roboter werde ich in der Programmiersprache NQC programmieren. NQC steht für Not Quite C. Sie ist stark mit der Programmiersprache C verwandt; Es ist ein C, das auf die Steuerung des RCX zugeschnitten ist. Die Sprache wurde von Dave Baum, einem RCX-Kenner, geschrieben. Da sie auf den RCX abgestimmt ist, hat sie etliche Einschränkungen gegenüber C. Obwohl NQC speziell für das RCX geschaffen wurde, gibt es doch eine logische Trennung zwischen der NQC-Sprache und der RCX-API[8], die das RCX steuert.
Da eine NQC-Dokumentation zu lang wäre, sind hier die für diese Arbeit wichtigsten Teile der Sprache erklärt.

2.2.4.1 Schlüsselwörter

Eine Reihe von Wörtern können nicht für die Benennung von Variablen, Funktionen und Konstanten verwendet werden. Es sind Schlüsselwörter die der NQC Sprache vorbehalten sind: *__sensor, abs, asm, break, const, continue, do, else, false, if, inline, int, repeat, return, sign, start, stop, sub, task, true, void, while*

2.2.4.2 Konstanten

Werden dieselben Werte im Programmcode immer wieder verwendet, empfiehlt es sich, sie am Anfang des Codes als Konstante abzuspeichern. Um eine Konstante mit dem Namen Turntime und dem Wert 200 zu definieren, wäre folgender Code nötig:

```
#define TURNTIME 200
```

[8] API = Application Programming Interface; auf Deutsch Schnittstelle zur Anwendungsprogrammierung. Siehe auch2.2.4.8 Die RCX-API.

2.2.4.3 Variablen

NQC kann 32 Variablen verwenden. Diese sind entweder global oder lokal im jeweiligen Programm vereinbart. Es sind ganze Zahlen bis zu einer Grösse von 16 Bit möglich. Variablen werden durch *int* vereinbart. Durch Zuweisungsoperatoren wie (=, +=, ||=) werden die Werte der Variablen verändert. Der Zuweisungsoperator ||= weist beispielsweise der Variablen den Absolutwert des Ausdrucks zu.

Beispiel:

```
int example;                // Variable mit dem Namen example wird definiert

example = 2;                // Die Variable example ist 2.
example -= 8;               // Der Variable example wird 8 subtrahiert, sie ist jetzt -6.
example ||= example;        // Die Variable example erhält den Absolutwert des Ausdrucks, sie ist
                            // jetzt 6.
```

2.2.4.4 Programmstruktur/Programmblöcke

Ein NQC Programm besteht aus globalen Variablen und Programmblöcken. Es gibt drei verschieden Typen von Programmblöcken mit unterschiedlichen Eigenschaften:

1. **Der Task**: Der RCX unterstützt Multitasking; Bis zu 10 Tasks können parallel nebeneinander laufen. Ein Task hat den Namen *main*, er wird beim Programmstart aufgerufen. Tasks können einander gegenseitig mit den Befehlen *start* und *stop* starten und beenden.
2. **Das Unterprogramm**: Wird ein Programmablauf an verschiedenen Stellen benötigt, ist ein Unterprogramm hilfreich. Es wird nur einmal im RCX gespeichert und kann bei Bedarf aufgerufen werden. Bis zu acht Unterprogramme können gespeichert werden. Dem Unterprogramm können aber weder Werte mitgegeben werden, noch kann es Werte zurückliefern. Ein Unterprogramm kann auch nicht ein anderes Unterprogramm aufrufen, es kann aber selbst mehrfach aufgerufen werden. Ein gleichzeitiges Ablaufen führt aber zu Funktionsstörungen.
3. **Die Inline-Funktion:** Im Gegensatz zum Unterprogramm werden Inline-Funktionen im RCX direkt an all die Stellen kopiert, wo sie benötigt werden. Das bringt den Vorteil mit sich, das sie mehrfach aufgerufen werden können. Der Nachteil ist, dass bei jeder Kopie Speicherplatz benötigt wird. Inline-Funktionen haben weiter den Vorteil, dass man ihnen mehrere Argumente oder Werte übergeben kann.

Unterprogramme sowie Inline-Funktionen werden über ihren Namen und anschliessende Klammern mit eventuellen Werten aufgerufen.

Beispiel:

```
void beispiel (int example)     // Definition einer Inline-Funktion beispiel mit dem Argument example
{
  Code;
}
```

2.2.4.5 Wichtige Anweisungen

Als wichtigste, durch NQC unterstützte Anweisungen gelten die *If*-Anweisung, die *while*-Anweisung und deren Verwandte, die *do-while*-Anweisung und die *repeat*-Anweisung. NQC unterstützt zusätzlich den Befehl *until*, der eine Alternative zur while-Anweisung darstellt. Es ist eine *while*-Anweisung, die etwas anders definiert ist. Die verwendete Definition lautet:

```
#define  until (Bedingung)  while (! (Bedingung) )
```

Die Schleife wird also so lange fortgesetzt, bis die Bedingung erfüllt ist. Als Bedingung wird meist ein Sensor abgefragt und die Schleife ist leer, d. h. das Programm wartet, bis ein Sensor einen gewissen Wert liefert.
Mit den Anweisungen *break*, *continue* und *return* kann der Ablauf in einer Schleife verändert werden.

Beispiel:

```
// Programm das 9 mal den Ton „DOUBLE_BEEP“ abspielt
task main ()                                  // der Maintask
{
  beep (9);                                   // rufe die Inline-Funktion „beep“ auf und übergibt ihr
}                                             // das Argument 9
void beep (int n)
{
  repeat (n)                                  // wiederhole “n”-mal, in unserem Fall 9-mal
  {
    PlaySound (SOUND_DOUBLE_BEEP);            // spiel den Ton “DOUBLE_BEEP”
    Wait (10);                                // warte 10 Hundertstelsekunden
  }
}
```

2.2.4.6 Bedingungen

Der Programmablauf enthält manchmal Bedingungen. Zwei Bedingungen können mit logischen *und*- und *oder*-Opereatoren verknüpft werden. In dieser Tabelle [9]sind sämtliche Bedingungen aufgelistet:

Bedingung	Bedeutung
true	immer wahr
false	immer falsch
ausdruck1 == ausdruck2	wahr, wenn ausdruck1 gleich ausdruck2 ist
ausdruck1 != ausdruck2	wahr, wenn ausdruck1 nicht gleich ausdruck2 ist
ausdruck1 < ausdruck2	wahr, wenn ausdruck1 kleiner ausdruck2 ist
ausdruck1 <= ausdruck2	wahr, wenn ausdruck1 kleiner oder gleich ausdruck2 ist
ausdruck1 > ausdruck2	wahr, wenn ausdruck1 grösser ausdruck2 ist
ausdruck1 >= ausdruck2	wahr, wenn ausdruck1 grösser oder gleich ausdruck2 ist
! Bedingung	logische Negation einer Bedingung - wahr, wenn die Bedingung falsch ist
Bedingung1 && Bedingung2	logisches UND (nur wahr, wenn beide Bedingungen wahr sind)
Bedingung1 || Bedingung2	logisches ODER (nur wahr, wenn mindestens eine Bedingungen wahr ist)

[9] Aus http://lug.mfh-iserlohn.de/lego/NQC_2.0r2.de.html, siehe auch 5 Verzeichnisse

Tabelle 1

Beispiel:

```
// Programm das den Ton „DOUBLE_BEEP“ solange abspielt, wie Sensor 1 den Wert 1 liefert
task main ()                                  // der Maintask
{
  SetSensor (SENSOR_1, SENSOR_TOUCH);   // SENSOR_1 ist ein Berührungssensor
  until (SENSOR_1 == 1)                       // bis SENSOR_1 den Wert 1 liefert
  {
    PlaySound (SOUND_DOUBLE_BEEP);        // spiel den Ton “DOUBLE_BEEP”
    Wait (10);                                // warte 10 Hundertstelsekunden
  }
}
```

2.2.4.7 Operatoren

Um Werte miteinander zu verknüpfen, benötigt man Operatoren, wie zum Beispiel +, -, abs, u. s. w.. Der Operator ++ vor einer Variable erhöht diese um den Wert eins, der Operator -- vermindert sie um eins.

Beispiel:

```
y = 3          // die Variable y ist 3 (Zuweisungsoperator)
x = y + 2      // die Variable x ist gleich der Variable y + 2
++x            // erhöhe den Wert der Variablen x um 1, sie hat jetzt den Wert 6
```

2.2.4.8 Die RCX-API

Das RCX-API legt eine Reihe von Konstanten, Funktionen und Werten fest, welche Zugriff auf Einrichtungen des RCX, wie etwa Sensoren und Ausgänge, erlauben.
In der folgenden Aufzählung sind die von mir häufig verwendeten Funktionen und Werte kurz erklärt:

- Die Namen *SENSOR_1*, *SENSOR_2* und *SENSOR_3* stehen für die Sensoreingänge des RCX. Es kann der Sensortyp und die Art, wie die Sensordaten ausgewertet werden, festgelegt werden. Bei einigen Sensoren kann man mit dem Befehl *ClearSensor()* den Sensorwert auf 0 zurückstellen. Es folgt eine Tabelle[10] mit den verschiedenen Sensorwerten.
-

Sensormodus	Bedeutung
SENSOR_MODE_RAW	Rohwert von 0 bis 1023
SENSOR_MODE_BOOL	boolescher Wert (0 oder 1)
SENSOR_MODE_EDGE	Zählwert für Impulsflanken (0-1- und 1-0-Wechsel
SENSOR_MODE_PULSE	Zählwert für Impulse
SENSOR_MODE_PERCENT	Prozentwert
SENSOR_MODE_FAHRENHEIT	Grad Fahrenheit
SENSOR_MODE_CELSIUS	Grad Celsius

Tabelle 2

- Die drei Ausgänge des RCX werden mit den Namen *OUT_A*, *OUT_B* und *OUT_C* bezeichnet. Jeder Ausgang hat die drei Eigenschaften Modus, Richtung und Leistung. Der Modus kann Ein, Aus oder Leerlauf sein. Alle Befehle für Motoren können gleichzeitig auf mehrere Ausgänge angewendet werden. Um einen Befehl an mehrere Ausgänge abzugeben, werden deren Namen einfach über ein Pluszeichen verbunden.

[10] Aus http://lug.mfh-iserlohn.de/lego/NQC_2.0r2.de.html, siehe auch 5 Verzeichnisse

- Mit dem Befehl *Wait* kann in einem Task eine bestimmte Wartezeit eingefügt werden, der Task wird „eingefroren“. Die Angabe der Wartezeit erfolgt in Hundertstelssekunden.
- Der RCX besitzt einen Infrarotsender/Empfänger. Er kann bis zu 256 verschiedene Nachrichten senden. Sendet ein RCX eine solche Nachricht, wird diese von allen sich im Empfangskreis befindenden Empfängern in einem Nachrichtenspeicher gespeichert. Mit dem Befehl *ClearMessage* kann dieser auf 0 zurückgesetzt werden. Mit *SendMessage (*0-255*)* werden die Nachrichten verschickt. Durch *Message*() erhält man den Wert der zuletzt erhaltenen Infrarotnachricht.

Beispiel:

```
task main ()
{
  SetSensor (SENSOR_1, SENSOR_LIGHT);          // der Sensor am 1. Eingang ist ein
                                               Lichtsensor
  SetSensorMode (SENSOR_1, SENSOR_MODE_RAW);   // interpretiere seine Werte im Modus
                                               raw
  if (SENSOR_1 < 800 && SENSOR_1 > 750)        // wenn der Lichtsensorwert zwischen
                                               750 und 800 ist
  {
    SendMessage (7);                           // sende Nachricht 7
    Wait (10);                                 // warte 10 Hundertstelsekunden
    SendMessage (7);                           // sende Nachricht 7
  }
}
```

3 Praktischer Teil

3.1 Anforderungen

Ich stelle mir die Aufgabe, einen Roboter zu programmieren, der eine Aufgabe erhält, diese ausführt und die Ergebnisse an den Auftragsgeber zurücksendet. Diese Idee wurde durch die Marsmissionen inspiriert, welche in diesen Jahren ablaufen.
Es wäre sicher einfacher, wenn der PC die gesamte Steuerung des Roboters übernehmen würde. Das wäre durchaus realisierbar. Doch das Übermitteln einer Information auf den Mars benötigt jedes Mal viel Zeit, so dass der Roboter nur mit einer sehr langen Verzögerung steuerbar wäre. Deshalb ist es auch meine Aufgabe, die Kommunikation zwischen PC und Roboter möglichst auf einem Minimum zu halten. Auf meinem Testgelände sollte der Roboter folgende Aufgabe erfüllen können:

3.1.1 Hauptaufgabe: search and examine

Der User gibt am PC die Koordinaten eines Untersuchungsobjekts sowie einige Wegpunkte ein. Der PC sendet dem Roboter diese Koordinaten. Anschliessend fährt der Roboter zum Untersuchungsobjekt, bohrt mit einem Bohrer ein Loch hinein und misst im Loch die Temperatur. Das Gestein, das beim Bohren herunterfällt, fängt der Roboter auf, befördert es in ein Untersuchungsbecken und untersucht es auf seine Helligkeit. Anschliessend fährt der Roboter zurück und bringt so das Untersuchungsmaterial zur Basisstation. Dort sendet er die gemessenen Werte an den PC.

3.1.2 Teilaufgaben

1. Der Roboter muss sich orientieren können. Für diesen Zweck bringe ich am Boden Markierungen an, die ein Lichtsensor wahrnimmt. Die erhaltenen Werte werden durch den Roboter verarbeitet. ->3.3.2 Die Teilfunktion coordinate
2. Zusätzlich muss der Roboter immer wissen, wie er im Raum steht. Dies geschieht mit Hilfe eines Kompasses. ->3.3.4 Die Teilfunktion orient
3. Der Roboter muss kontrolliert in eine Richtung fahren und sich wenden können. -> 3.3.5 Die Teilfunktion move

Der Roboter muss das Objekt untersuchen können. -> 0

4. Die Teilfunktion examine

Der Roboter muss anhand von Wegpunkten einen Weg zum Ziel fahren. -> 0

5. Die Teilfunktion drive_to_aim
6. Der Roboter muss die Koordinaten des Untersuchungsobjekts und der Wegpunkte empfangen können. -> 3.3.73.3.6.2 Die Teilfunktion get_job
7. Der Roboter muss die über das Untersuchungsobjekt gefundenen Informationen an den PC zurücksenden können. -> 3.3.8 Die Teilfunktion transfer_data

Alle Programme auf dem PC sind nicht Bestandteil meiner Maturaarbeit.

3.2 *Hardware meines Roboters*

3.2.1 Das Fahrwerk

Das Fahrwerk ist ein Hauptbestandteil meines Roboters. Da es aus Lego gebaut ist und eine Last von ca. 2 kg tragen muss, liegt die Hauptherausforderung in der Konstruktion eines stabilen Fahrwerks. Nebst der Stabilität sind mir folgende Aspekte wichtig:

- Der Roboter muss sich auf der Stelle wenden können.
- Trotz hohem Gewicht des Roboters sollte er sich mit einer angemessenen Geschwindigkeit fortbewegen können, d. h. ich muss eine angemessene Übersetzung wählen, um die Motoren nicht zu überlasten.
- Es sollte ein möglichst hoher Wirkungsgrad erzielt werden, was die Verwendung von wenigen Zahnrädern und die Vermeidung von Reibungsflächen bedingt.

Indem ich ein Fahrwerk konstruierte, welches auf beiden Seiten der Räder auf der Achse abstützt, konnte ich eine angemessene Stabilität des Roboters erreichen. Diese Konstruktion bringt Nachteile in der Geländetauglichkeit meines Roboters, dem Gewinn an Stabilität messe ich aber mehr Bedeutung zu. Dank eines Fahrwerks mit drei Rädern lässt sich der Roboter auf der Stelle wenden. Die zwei Haupträder werden über je einen Motor (M1 + M2) und über ein Getriebe mit einer Übersetzung von 5:1 angetrieben. Das dritte Rad ist ein schwenkbares Stützrad. Bereits im Fahrwerk integriert ist ein Lichtsensor (L1).

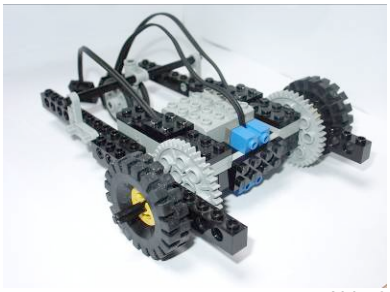


Abb. 6

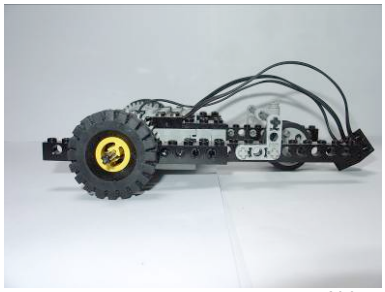


Abb. 7

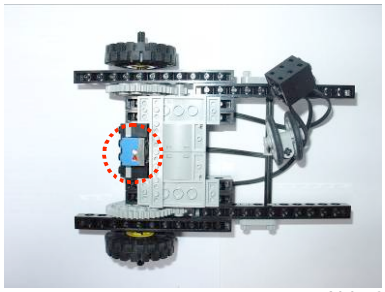


Abb. 8

In Abb. 6-8 sieht man das Grundfahrwerk. Abb. 8 zeigt, wie L1 exakt auf der Achse der Haupträder platziert wurde, so dass er sich bei einer Drehung des Roboters um die eigene Achse nicht von der Stelle bewegt.

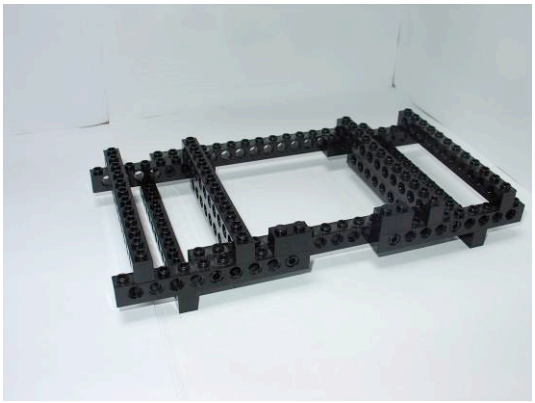


Abb. 9

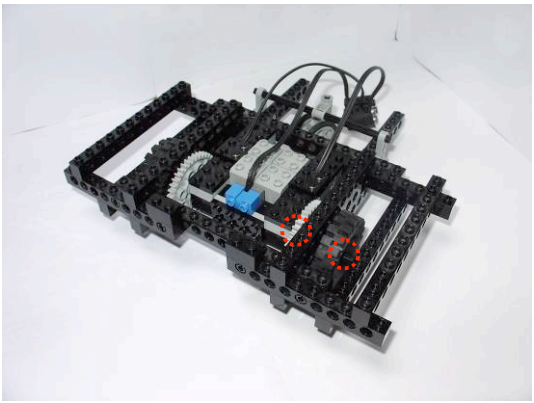


Abb. 10

In einem zweiten Schritt setzte ich auf das Fahrgestell ein Trägergestell (Abb. 9), das später die RCX's und sämtliche Sensoren tragen wird. In Abb. 10 sieht man die

Doppelabstützung der Räder, gerade oberhalb werden später RCX_M[11] und RCX_S[12] platziert sein.

3.2.1.1 Funktionstabelle Fahrwerk

Bezeichnung	Art	Beschreibung
M1	Motor	M1 und M2 sorgen für die Fortbewegung des Roboters. Mit Hilfe dieser Motoren kann der Roboter sich in x- und y-Richtung bewegen und sich um die z-Achse drehen.
M2	Motor	
L1	Lichtsensor	L1 nimmt Veränderungen der Helligkeit des Bodens unter dem Roboter wahr. Mit Hilfe von L1 wird der Roboter bestimmen können, wo er sich gerade im x-y-Koordinatensystem befindet.

Tabelle 3

3.2.1.2 In- und Outputs Fahrwerk

	Anzahl	Typ
Inputs	1	Lichtsensor
Outputs	2	Motor

Tabelle 4

3.2.2 Berührungssensorarm

Der Roboter sollte auf dem Weg zu seinem Arbeitsgebiet Hindernisse erkennen und diesen ausweichen können. Dies geschieht mit Hilfe zweier Berührungssensoren. Die Hauptanforderung an diese Berührungssensoren ist, eine möglichst grosse Seitenfläche des Roboters abzudecken, so dass möglichst alle Hindernisse wahrgenommen werden. Zudem ist es ein Vorteil, wenn man Angaben über den Ort des Hindernisses machen kann. Um diese beiden Ziele zu verwirklichen, habe ich den Berührungssensorarm folgendermassen aufgebaut:

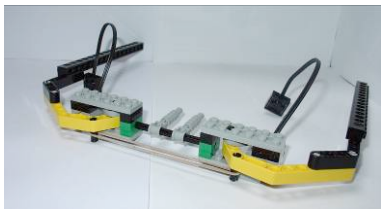


Abb. 11

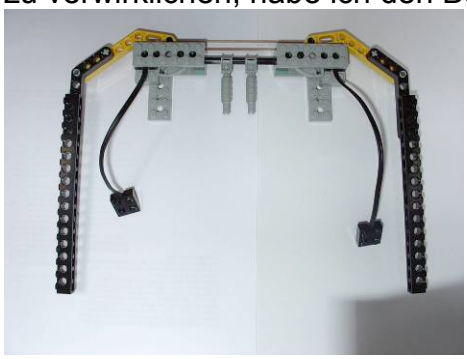


Abb. 12

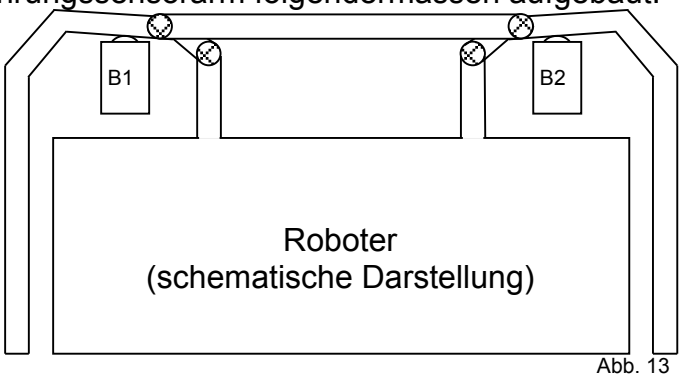


Abb. 13

[11] Master
[12] Slave

Zwei gekrümmte Legoarme sind an je einem Punkt beweglich am Roboter befestigt. Werden sie nach innen gedrückt, berühren sie die Sensoren. Sie werden durch einen Gummizug nach aussen gezogen.
Wenn ein Objekt von links an den Roboter stösst reagiert B1, von rechts B2.
Wenn der Roboter frontal in ein Objekt fährt, reagieren B1 und B2.

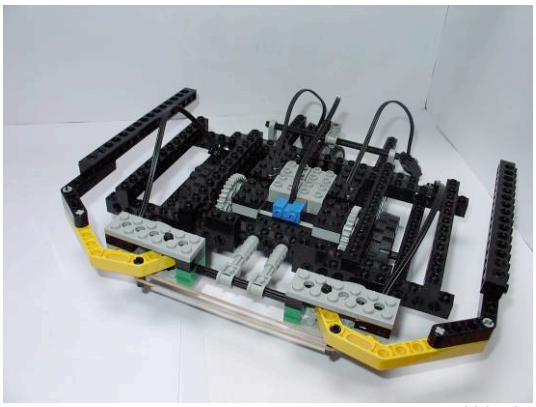


Abb. 14

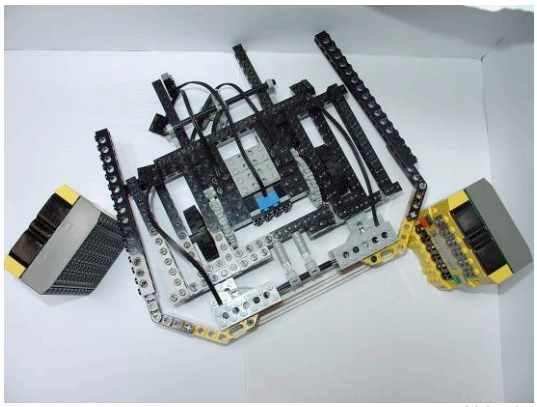


Abb. 15

Der ganze Berührungssensorarm wird am Roboter montiert. (Abb. 14)
In Abb. 15 sieht man, wie der Roboter in zwei Testhindernisse stösst, eins auf der rechten Seite und eins vorne links.

3.2.2.1 Funktionstabelle Berührungssensorarm

Bezeichnung	Art	Beschreibung
B1	Berührungssensor	Mit B1 und B2 kann der Roboter feststellen, ob er frontal, rechts oder links in ein Hindernis gestossen ist.
B2	Berührungssensor	

Tabelle 5

3.2.2.2 In- und Outputs Berührungssensorarm

	Anzahl	Typ
Inputs	2	Berührungssensor

Tabelle 6

3.2.3 Arbeitsarm

Eine der Hauptaufgaben des Roboters ist, Informationen über seine Umgebung aufzunehmen. Mein Roboter sollte fähig sein, mit Hilfe eines Arms einen Stein anzubohren und dann Informationen über dessen Beschaffenheit zu gewinnen. Die Hauptaufgabe des Arbeitsarms ist also, einen Arbeitskopf in die Nähe des Arbeitsraums zu bringen. Zusätzlich kann er:

- ein Materialauffangbecken ausfahren und anschliessend das eingefangene Material, welches beim Bohren herunterfällt, auf seine Helligkeit untersuchen.
- herausfinden, ob sich der Roboter überhaupt in einer geeigneten Position zur Untersuchung eines Untersuchungsobjekts befindet.

Die Mechanik des Arms ist einfach gestaltet. Der Arm kann sich nur in der x-Achse bewegen. Dieses Ausfahren wird vom Motor M4 angetrieben. Da die Legomotoren zu

schnell für so ein Ausfahren sind, wird die Kraft über ein Schneckengetriebe auf den Arm übertragen (Abb. 16).

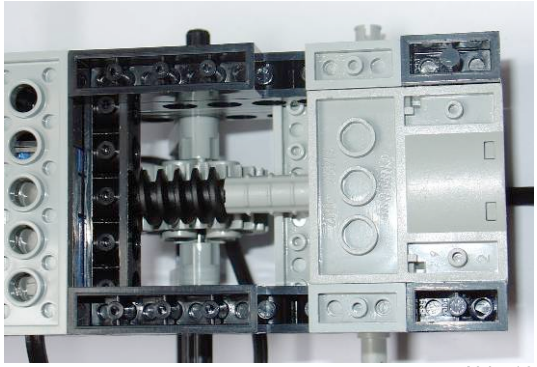


Abb. 16

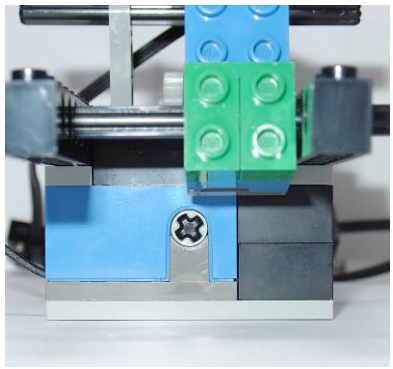


Abb. 17

Die Umdrehungen des Motors werden durch einen Rotationssensor R2 überwacht, das Ausfahren des Arms muss präzise erfolgen. Mit Hilfe des Rotationssensors weiss der RCX_S immer, wo sich der Arbeitskopf gerade befindet. (Abb. 17)
Das Materialauffangbecken wird mit dem Arm gekoppelt; bewegt sich der Arm nach vorne, bringt sich das Becken in Auffangposition, fährt der Arm zurück, kippt das Becken das Untersuchungsmaterial in den Untersuchungsbehälter. In diesem Untersuchungsbehälter untersucht ein Lichtsensor L3 die Helligkeit des Materials. Das Untersuchungsobjekt wird mit Hilfe eines Berührungssensors B3 ausgemacht.

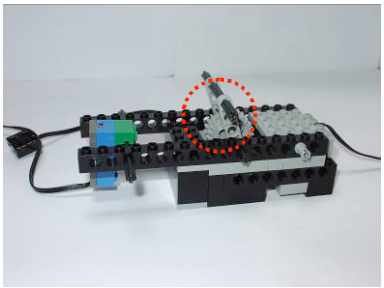


Abb. 18

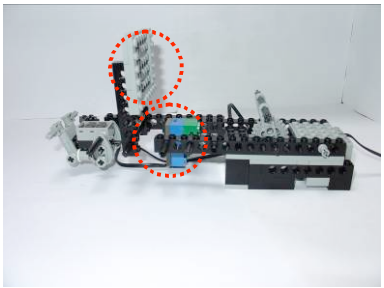


Abb. 19

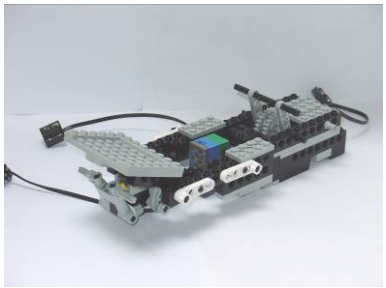


Abb. 20

Abbildung 18-20 stellen den Aufbau des Arbeitsarms dar. In Abb. 18 sieht man die Montierung, an die der Arbeitskopf befestigt wird. In Abb. 19 zeigt den Berührungssensor an der Spitze des Arms, das Auffangbecken in hochgeklappter Stellung und den Lichtsensor, der das Material untersuchen wird. In Abb. 20 ist das Auffangbecken ausgeklappt.

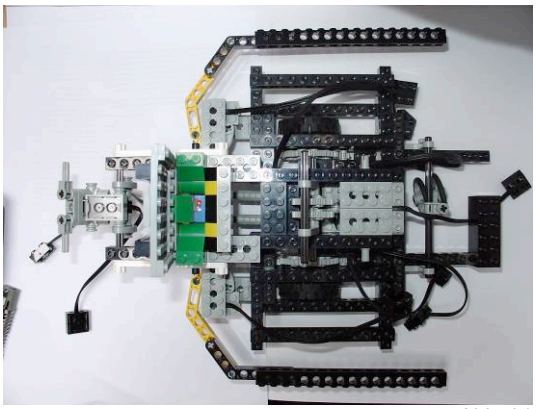


Abb. 21

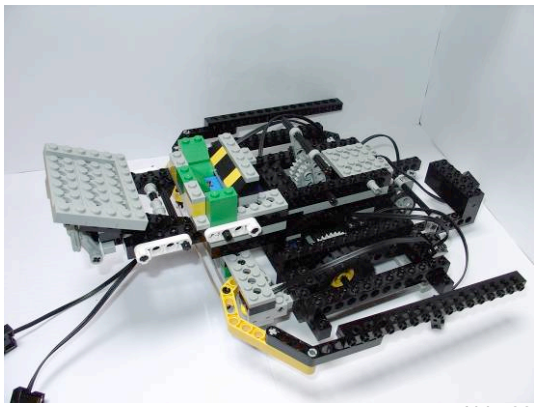


Abb. 22

Am Schluss montierte ich den Arbeitsarm auf den Roboter (Abb. 21 + 22). Er befindet sich in der Mitte des Roboters, genau über den beiden Antriebsmotoren M1 und M2 und stützt sich auf den Berührungssensorarm auf. In Abb. 22 sieht man das über dem Lichtsensor L3 montierte Sammelbecken.

3.2.3.1 Funktionstabelle Arbeitsarm

Bezeichnung	Art	Beschreibung
M3	Motor	M3 verschiebt den Arbeitskopf in der x-Achse und bewegt über eine Schnur das Auffangbecken.
R2	Rotationssensor	R2 überprüft, wie weit sich der Arbeitskopf in der x-Achse verschoben hat.
L3	Lichtsensor	L3 untersucht das Untersuchungsmaterial auf seine Helligkeit
B3	Berührungssensor	Mit B3 stellt der Roboter fest, ob er sich gegenüber einem zu Untersuchenden Objekt befindet.

Tabelle 7

3.2.3.2 In- und Outputs Arbeitsarm

	Anzahl	Typ
Inputs	1	Rotationssensor
	1	Lichtsensor
	1	Berührungssensor
Outputs	1	Motor

Tabelle 8

3.2.4 Arbeitskopf

Der Arbeitskopf hat zwei Aufgaben:

1. Er muss ein Objekt in ein Loch bohren.
2. Er muss in diesem Loch die Temperatur messen.

Zusätzlich sollte das Aushubmaterial via Auffangbecken in das Untersuchungsbecken gelangen, wo es L3 auf seine Helligkeit untersucht.
Um zwei verschiedene Arbeitsgeräte (Bohrer und Temperatursensor) an denselben Punkt zu bringen, habe ich einen drehbaren Arbeitskopf gebaut. Mit Hilfe des Motors M5 dreht sich der Arbeitskopf um die eigene Achse.

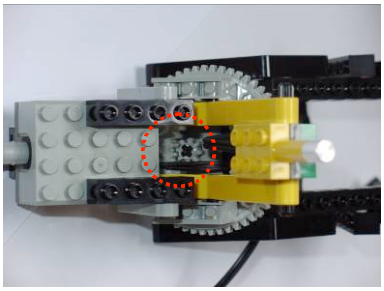


Abb. 23

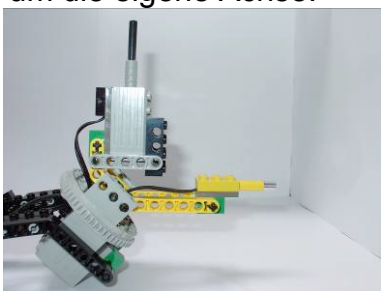


Abb. 24

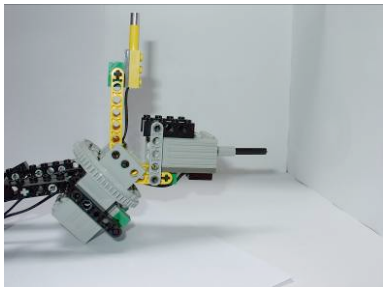


Abb. 25

Der Kopf darf nicht zu schnell drehen, deshalb liegt eine 3:1 Übersetzung vor (Abb. 23). In Abb. 24 ist der Temperatursensor T1 in Position und Abb. 25 der Bohrer, angetrieben vom Motor M6.

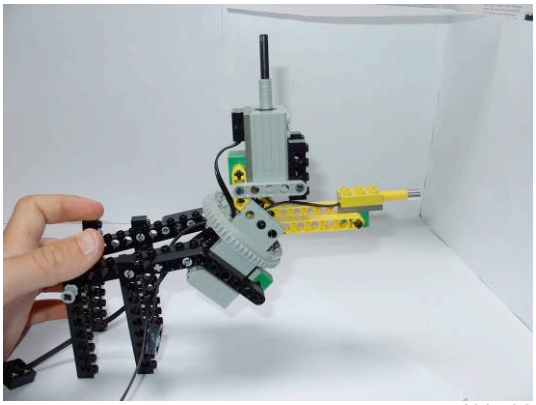


Abb. 26

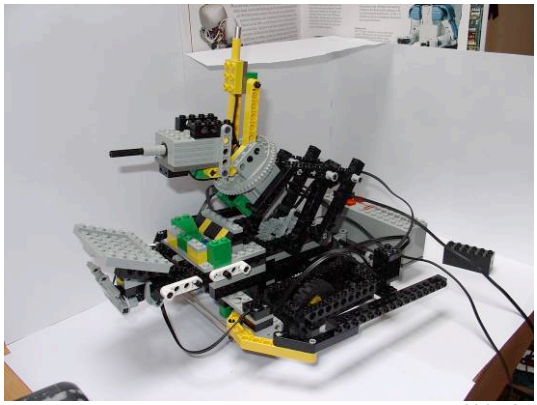


Abb. 27

Hinten am Bohrkopf montierte ich ein bewegliches Gestell, welches am Arbeitsarm befestigt wird (Abb. 26). Zum Schluss installierte ich den Arbeitskopf auf den Roboter (Abb. 27). Auf diesem Bild befindet sich der Arbeitskopf in eingefahrenem Zustand. Das Verbindungsseil zum Auffangbecken fehlt auf Abbildung 27 noch. Der Temperatursensor T1 kann in dieser Position auch die Temperatur der Umgebungsluft messen.

3.2.4.1 Funktionstabelle Arbeitskopf

Bez.	Art	Beschreibung
M5	Motor	M5 dreht den Arbeitskopf um die eigene Achse. Mit seiner Hilfe kann der Roboter auswählen, ob er ein Loch bohren, oder darin die Temperatur messen will.
M6	Motor	M6 dreht den Bohrer.
T1	Temperatursensor	Die Temperatur im Bohrloch sowie die Temperatur der Umgebungsluft lassen sich mit T1 bestimmen.

Tabelle 9

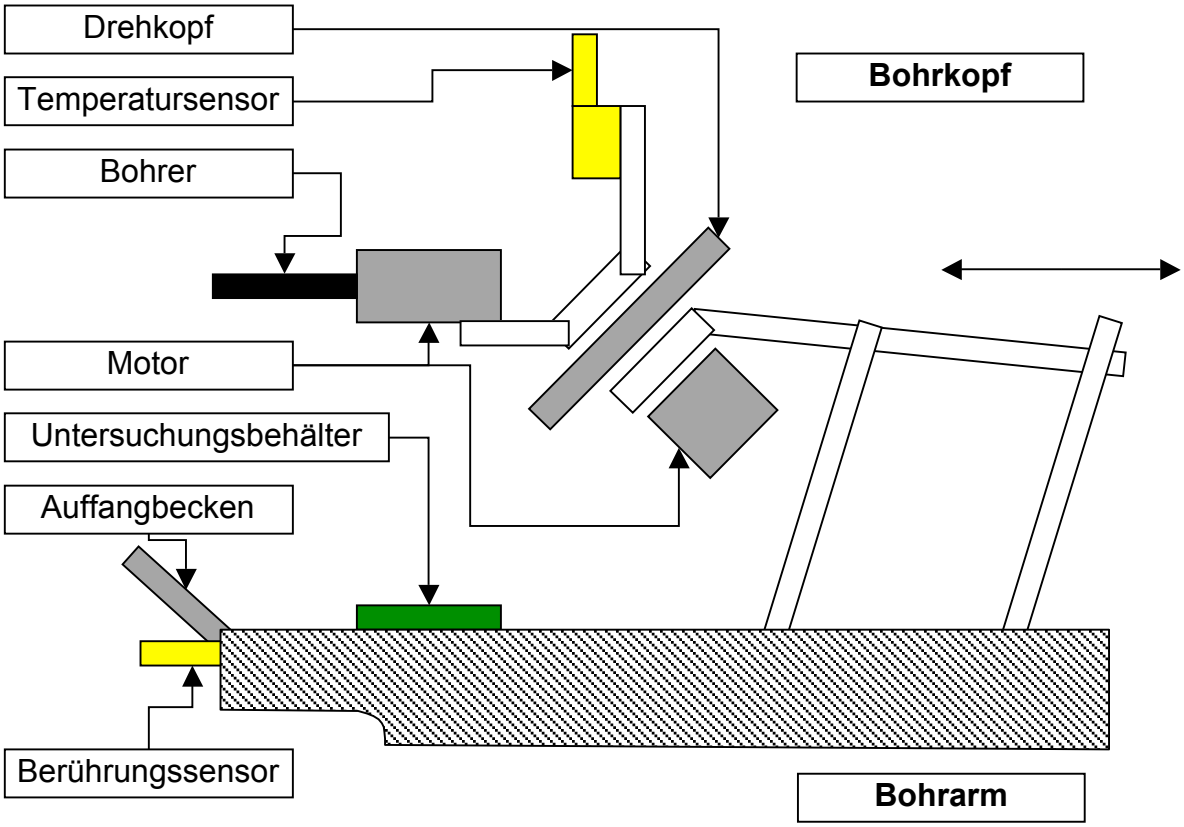
3.2.4.2 In- und Outputs Arbeitkopf

	Anzahl	Typ
Inputs	1	Temperatursensor
Outputs	2	Motor

Tabelle 10

3.2.5 Bohrarm mit Bohrkopf

Der Bohrarm und der Bohrkopf bilden zusammen eine Arbeitseinheit. In Grafik 1 wird ihre Funktionsweise dargestellt.



Grafik 1

3.2.6 Kompass

Damit der Roboter immer weiss, wie er im Raum steht, muss er einen Kompass ablesen können. Dazu kann der Roboter einen Lichtsensor über dem Kompass kreisen lassen. Sobald der Lichtsensor einen helleren Wert als gewöhnlich liefert, befindet er sich über der Kompassnadel. Der bis dahin zurückgelegte Drehweg wird von einem Rotationssensor gemessen. Anhand dieses Drehweges berechnet der RCX die aktuelle Ausrichtung in der Z-Achse.

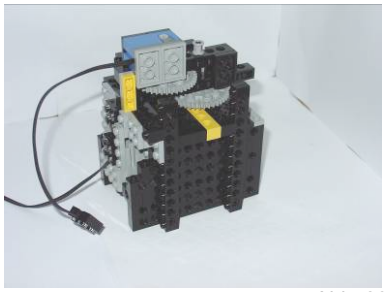


Abb. 28

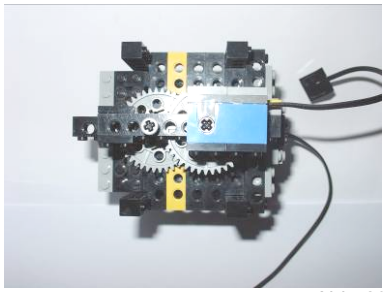


Abb. 29

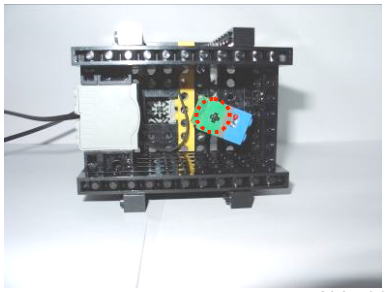


Abb. 30

Zuerst konstruierte ich das Gehäuse, in dem der Lichtsensor drehen kann. Da es nicht viel Licht durchlässt, wird der Lichtsensor wenig von äusseren Einflüssen gestört.

- Abb. 29: Durch vier Zahnräder wird die Geschwindigkeit des Motors um den Faktor 1/25 verkleinert.
- Abb. 30 zeigt, wie der Lichtsensor um die rot markierte Achse drehen wird.


Abb. 31

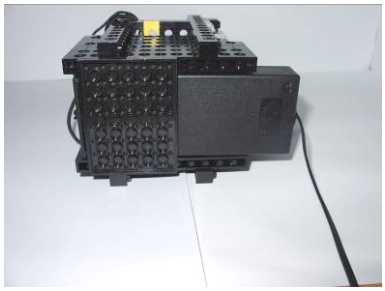


Abb. 32

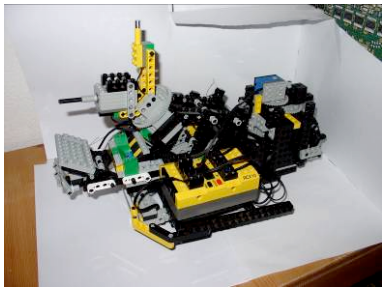


Abb. 33

In einem zweiten Schritt montierte ich einen Kompass unten an das schwarze Gehäuse.

- Der Kompass (Abb. 31) wird montiert (Abb. 32).
- In Abb. 33 wird das ganze Gebilde auf den Roboter gesetzt.

3.2.6.1 Funktionstabelle Kompass

Bezeichnung	Art	Beschreibung
M3	Motor	M5 dreht den Lichtsensor über dem Kompass.
R1	Rotationssensor	R1 misst, wie weit sich der Lichtsensor über dem Kompass dreht.
L2	Lichtsensor	L2 misst die Helligkeit über dem Kompass. Anhand seiner Werte kann die Kompassnadel ausgemacht werden.

Tabelle 11

3.2.6.2 In- und Outputs Kompass

	Anzahl	Typ
Inputs	1	Lichtsensor
	1	Rotationssensor
Outputs	1	Motor

Tabelle 12

3.2.7 2 RCX's

Am Schluss setzte ich noch die beiden RCX's auf den Roboter.

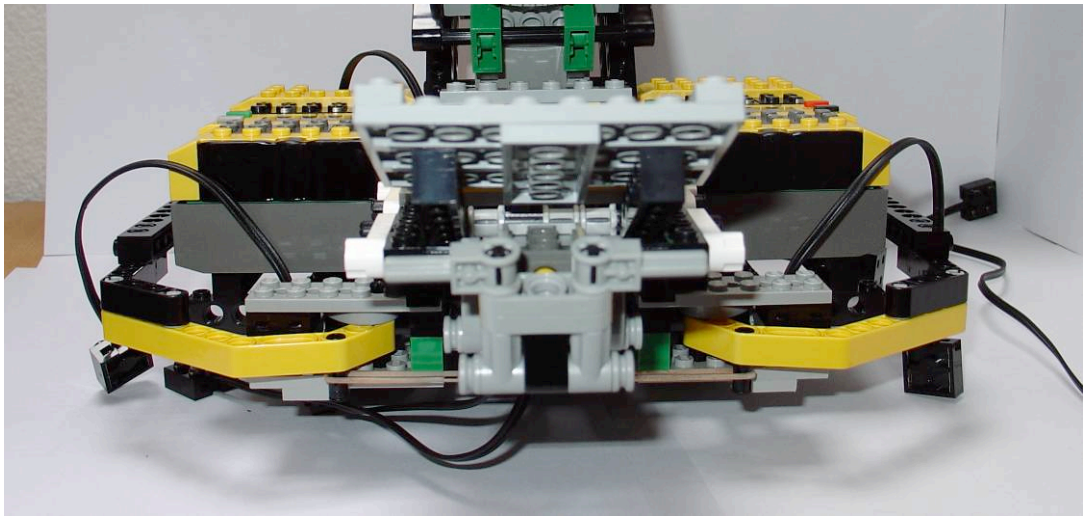


Abb. 34

Abb. 34: Rechts ist der Master, der für die Fortbewegung des Roboters verantwortlich ist, und links der Slave, der den Arbeitsvorgang steuert.

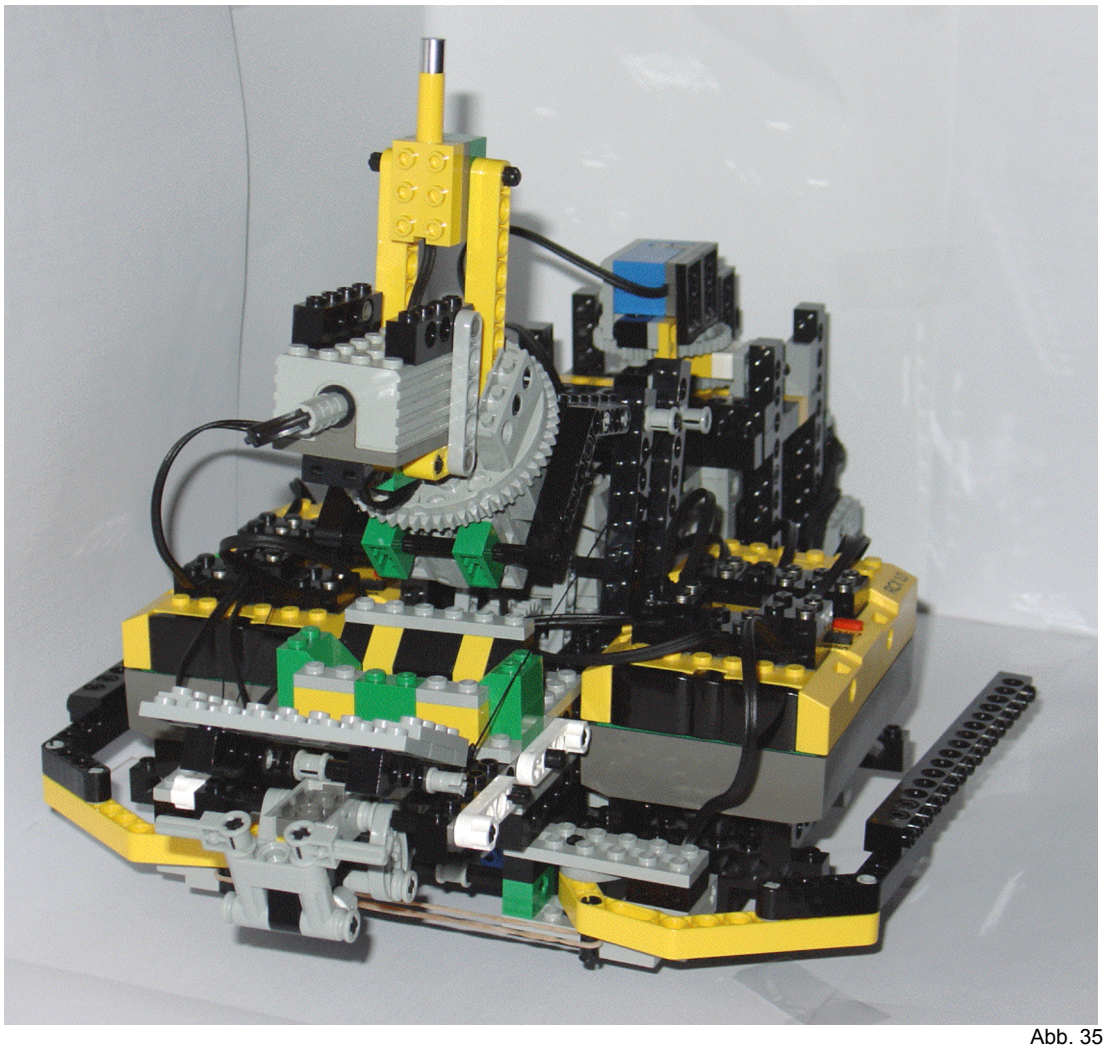
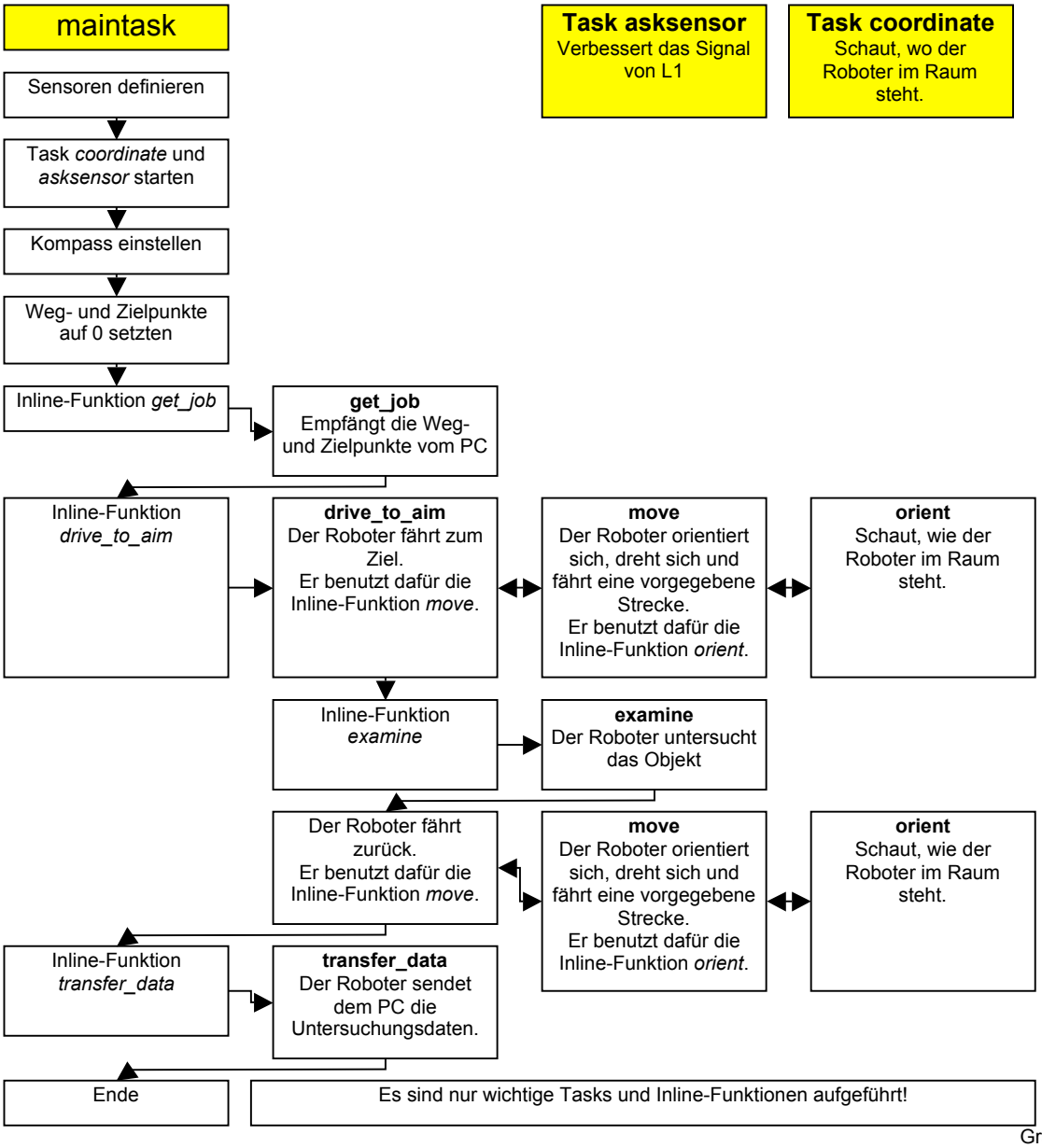


Abb. 35

3.3 Software meines Roboters

3.3.1 Allgemeines Konzept

Die Grundidee ist, meine Software aus verschiedenen Teilfunktionen aufzubauen. Der Code dieser Teilfunktionen besteht meist aus Inline-Funktionen oder Tasks. Durch diese Art des Aufbaus gewinnt mein Programm an Übersichtlichkeit; Fehler können schneller korrigiert werden.
Der Maintask macht nicht besonders viel. Zuerst definiert er die Sensoren, dann stellt er R1 nach der Kompassnadel ein und setzt die Weg- und Zielpunkte auf null. Er startet den Task *coordinate* und den Task *asksensor*. Anschliessend führt er die Inline-Funktionen *get_job*, *drive_to_aim* und *transfer_data* nacheinander aus. Der ganze Ablauf als Schema[13]:

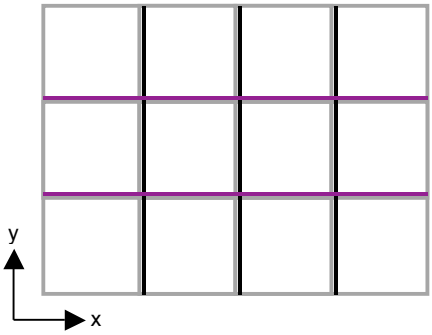


Grafik 2

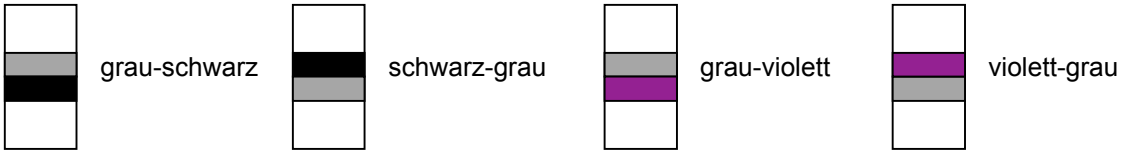
[13] Eine Zeichenerklärung zu Schemas befindet sich im Anhang. Für das Verständnis der Schemas kann der Code im Anhang sehr hilfreich sein.

3.3.2 Die Teilfunktion coordinate

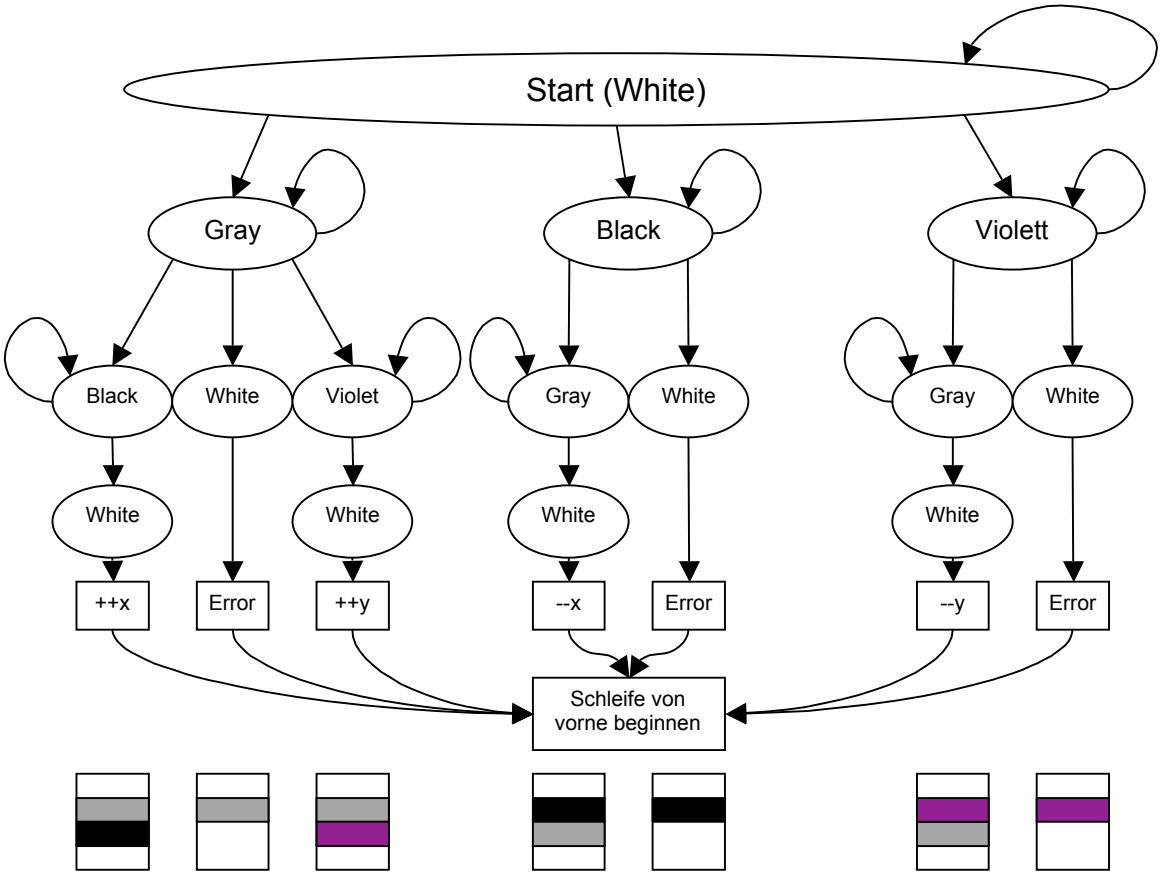
Der Roboter sollte immer wissen, wo er sich im Raum befindet. Da dies eine Aufgabe ist, die der Roboter permanent ausführen muss, egal was sonst passiert, wird sie von einem separaten Task *coordinate* ausgeführt.
Die Informationen über seinen Standort erhält der Roboter über den Lichtsensor L1, welcher zu diesem Zweck unten am Roboter angebracht ist. Am Boden sind im Abstand von 30 cm verschiedenfarbige Linien angebracht, die es dem Roboter ermöglichen, sich in diesem Koordinatensystem zurechtzufinden. Der Übergang grau-schwarz ist in x-Richtung, der Übergang grau-violett in y-Richtung. Die Schwierigkeit bestand darin, den Roboter so zu programmieren, dass er vier verschiedene Übergangstypen unterscheiden kann.



Grafik 3



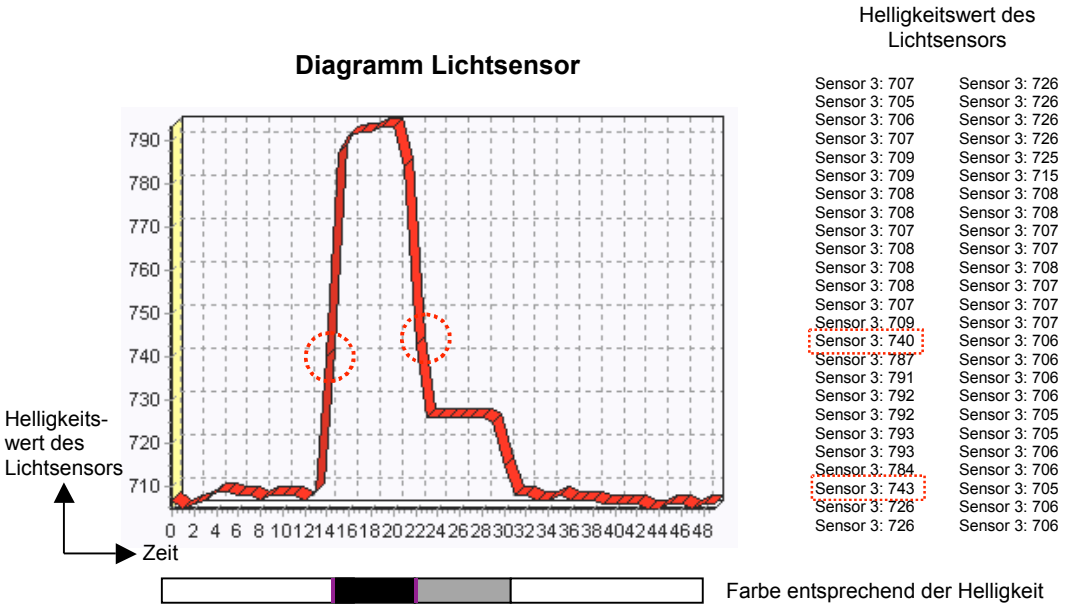
Sobald der Roboter eine dieser Linien überschreitet, sollte ein Task eine der beiden Variablen x, y um eins erhöhen oder senken. Das Schema dieses Tasks sieht folgendermassen aus:



Grafik 4

Nach mehrmaligem Umformen und Ausprobieren ist es mir schliesslich auch gelungen, dieses Schema in NQC funktionstüchtig umzusetzen. Faktoren wie

Genauigkeit und Geschwindigkeit mussten berücksichtigt werden. Das Berechnen der aktuellen Position war jedoch noch immer sehr ungenau, wurde doch z. B. der Übergang schwarz-grau als grau-violett oder grau-schwarz interpretiert. Das Problem ist, dass der Lichtsensor beim Übergang von weiss zu schwarz oder schwarz zu weiss Zwischenwerte wie grau oder violett wahrnimmt.

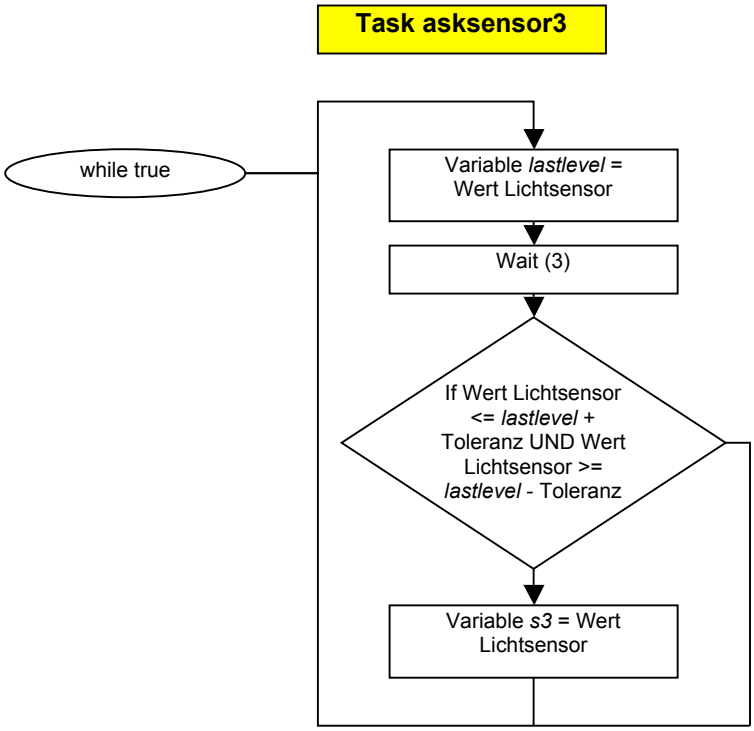


Grafik 5

Auf dem Diagramm sieht man, dass der Lichtsensor beim Überfahren einer Grenze schwarz/grau oder schwarz/weiss kurzzeitig die Werte 740 oder 743 misst. Der Task *coordinate* interpretiert diese Werte als violett, obwohl der Roboter nie eine violette Linie überfahren hat. Deshalb konstruierte ich den Task *coordinate* so, dass er nicht direkt die Werte des Lichtsensors, sondern eine Variable auswertet, die einem aufbereiteten Lichtsensorwert entspricht.
Diese Variable s3 bereite ich im Task *asksensor3* auf.

3.3.2.1 Task asksensor3

Der Task *asksensor3* hat die Aufgabe, den Wert des Lichtsensors aufzubessern, d. h. kleine Schwankungen zu ignorieren. Das erreicht er dadurch, indem er erst Werte weitergibt, die sich während einer bestimmten Zeit nur innerhalb einer vorgegebenen Toleranz verändert haben. Als Schema sieht das so aus:

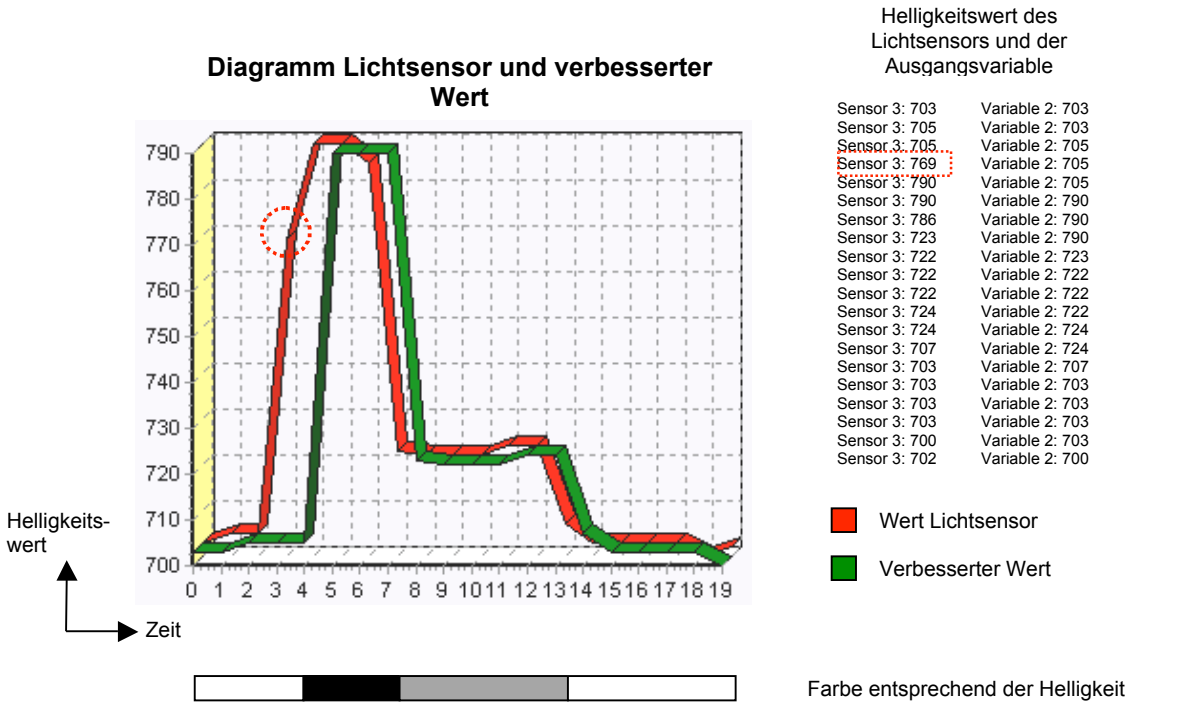


Grafik 6

Ich konstruierte zwei verschiedene Systeme. Das erste System hat den Nachteil, dass es immer nur nach einer Wartezeit überprüft, ob sich der Sensorwert nicht stark verändert hat. Im zweiten System wird das die ganze Zeit überprüft, und wenn der Sensorwert sich nach einer bestimmten Zeit nicht verändert hat, wird er weitergegeben. Der Vorteil des zweiten Systems liegt darin, dass eine rasche Veränderung des Sensorwertes sofort, und nicht erst, nachdem die Wartezeit abgelaufen ist, festgestellt wird. In NQC sieht dieses zweite System so aus:

```
task asksensor3 () // L1 Verbesserungstask
{
  // Die While-Schleife beginnt.
  while (true)
  {
    lastlevel = SENSOR 3;
    // Die Zählvariable auf 0 setzten.
    ns3 = 0;
    // Solange der Sensorwert nicht zuviel abweicht, läuft die Schleife weiter.
    while ((SENSOR_3 <= lastlevel + 1) && (SENSOR_3 >= lastlevel - 1))
    {
      // Zählvariable um 1 erhöhen.
      ns3 = ns3 + 1;
      // Wenn die Schleife 5 mal durch ist...
      if (ns3 == 5)
      {
        // Gib den lastlevel-Wert in die Ausgabevariable s3.
        s3 = lastlevel;
        // Beende die Schleife.
        break;
      }
      // Warte 1/100 Sekunde.
      Wait (1);
    }
  }
}
```

Das zweite System erzielt in der Praxis aber nicht viel bessere Resultate und benötigt eine Variable mehr, deshalb entschied ich mich für das erste System. Anschliessend musste ich ausprobieren, welcher der optimale Wert für die Abweichungstoleranz ist und über wie lange Zeit der Sensor in dieser Toleranz bleiben muss, um das beste Resultat zu erzielen. Aus den Daten, die der Sensor während der Überfahrt eines schwarz-grau-Übergangs liefert, und den dazugehörigen, vom System verbesserten Werten, zeichnete ich ein Diagramm:



Grafik 7

Mit dem Task *asksensor* werden also nur Werte weitergegeben, welche sich während 3/100 Sekunden um maximal 3 Lichtstärkeeinheiten verändern. Kurze Werte wie im Diagramm der Wert 769 werden einfach ignoriert. Der Nachteil ist, dass die Werte erst mit einer gewissen Verzögerung zum Anwender gelangen.

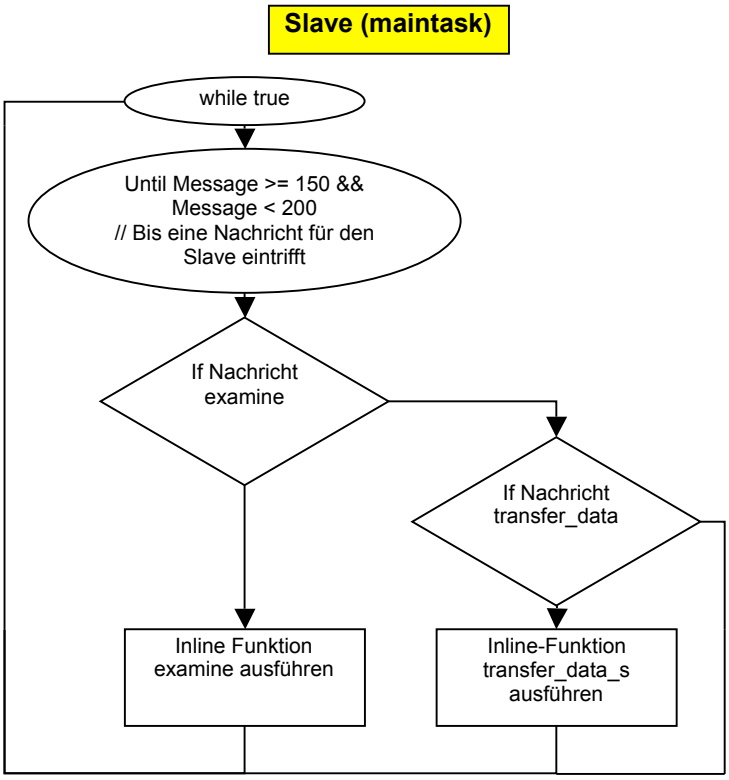
3.3.2.2 Detailtabelle der Teilfunktion coordinate

Master	Anzahl/Name	Beschreibung
Tasks	2	
	coordinate	Überprüft, ob der Roboter eine Koordinatenlinie überfährt und setzt die Variablen *x*, *y* entsprechend den überfahrenen Linien.
	asksensor3	Bessert den Wert des Lichtsensors L1 auf.
Unterprogramme	0	
Inline-Funktionen	0	

Tabelle 13

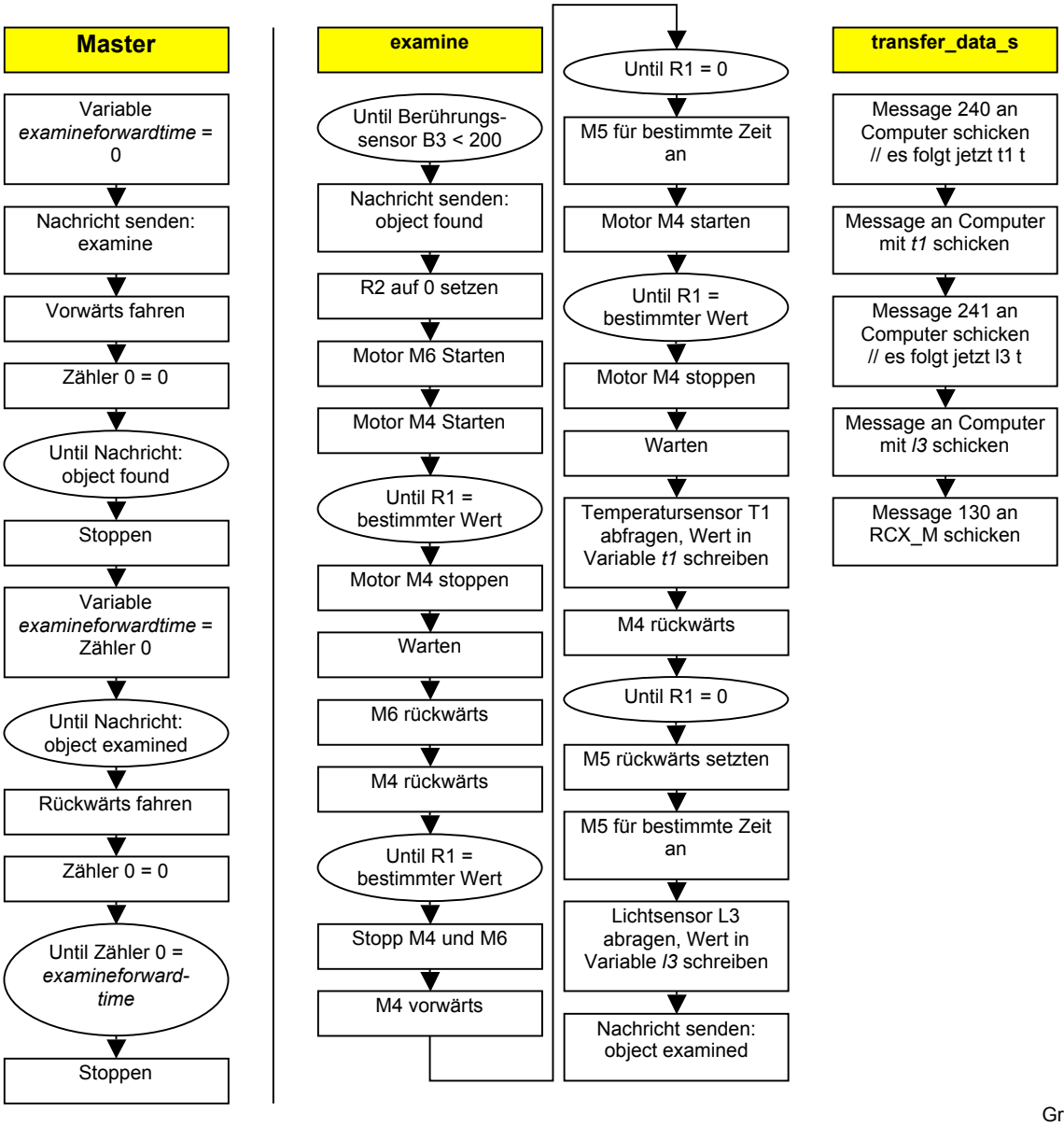
3.3.3 Die Teilfunktion examine

Die Bedingung für die Teilfunktion ist, dass der Roboter in seinem Zielquadrat angekommen ist und dem Untersuchungsobjekt zugewandt ist. Nach der Ausführung dieser Teilfunktion sollte der Roboter wieder an derselben Position stehen. Um das Untersuchungsobjekt ausfindig zu machen, ein Loch zu bohren, die Temperatur zu messen und den Aushub zu untersuchen, werden Sensoren und Motoren des zweiten RCX gebraucht. Die Hauptschwierigkeit dieser Funktion ist sicher, die beiden RCX's kommunikationsfähig zu machen. In den Teilfunktionen *get_job* und *transfer_data,* die später beschrieben werden, wird dieses Thema wieder eine Rolle spielen. Die restliche Ausführung der Aufgabe gestaltet sich einfach, es ist hauptsächlich ein Ablauf von Befehlen. Der RCX, der für die Fahrt und die Orientierung zuständig ist, nennt sich Master, der andere RCX, der für die Untersuchung sorgt, nennt sich Slave. In Grafik 9 ist der Code der Teilfunktion schematisch dargestellt:



Grafik 8

Der Aufbau ist folgender:
Der Slave hat zwei mögliche Aufgaben, die er ausführen kann: die Inline-Funktion *transfer_data_s* und die Inline-Funktion *examine*. Er überprüft sie immer auf Ausführbefehle vom RCX_M (Grafik 8). Wegen der Übersichtlichkeit sind beide Inline-Funktionen des Slave in Grafik 9 aufgeführt, für die Teilfunktion examine ist aber nur die gleich benannte Inline-Funktion examine von Bedeutung. Die Inline-Funktion transfer_data spielt im Kapitel 3.3.6.2 eine Rolle.



Grafik 9

Ablauf

Damit der Roboter nach der Untersuchung genau den Weg zurückfährt, den er für die Hinfahrt zurückgelegt hat, speichert er die Fahrzeit bis zum Objekt in der Variable *examineforwardtime*. Sobald der Master den Befehl zum Fahren gegeben hat, meldet er dies dem Slave. Wenn der Arbeitsarm an das Untersuchungsobjekt stösst, meldet dieser zurück, dass der Roboter beim Untersuchungsobjekt angekommen ist. Der Master stoppt und wartet, bis der Slave seine Überprüfungen abgeschlossen hat. Anschliessend fährt er an die Ausgangsposition zurück. Während der Slave seine Messungen durchgeführt, speichert er die Messwerte in zwei Variablen.

Kommunikation:

Mit dem RCX kann ich 256 verschiedene Nachrichten übermitteln. Für die Kommunikation musste ich ein Protokoll[14] schreiben. Mit den Nachrichten 0-99 werden Zahlenwerte von 0-99 übermittelt. Nachrichten zwischen 100 und 149 sind an den Master, solche zwischen 150 und 199 an den Slave und solche zwischen 200 und 249 an den PC gerichtet. Die Nachrichten 250 bis 255 sind allgemeine Nachrichten.

[14] Siehe Anhang unter Kommunikationsprotokoll

3.3.3.1 Die Inline-Funktion scm

Um zwischen zwei RCX's oder zwischen einem RCX und einem PC eine Nachricht zu übermitteln, reicht es, wenn diese einmal gesendet wird. Nun kann es aber sein, dass eine Nachricht den Empfänger nicht erreicht. Beim Programmieren der Teilfunktion examine bin ich das erste Mal auf dieses Problem gestossen. Wie kann man die Kommunikation verbessern? Eine einfache Variante ist die, dass eine Nachricht solange gesendet wird, bis eine Bestätigung für den Empfang dieser Nachricht eintrifft. Diese Idee setzte ich auch um. Ich hatte aber mit Problemen zu kämpfen.
Ein Beispiel:

- Infrarotschnittstellen können nur senden oder empfangen. Wenn jetzt der Sender permanent ein Signal abgibt, wird er das Bestätigungssignal nicht empfangen.

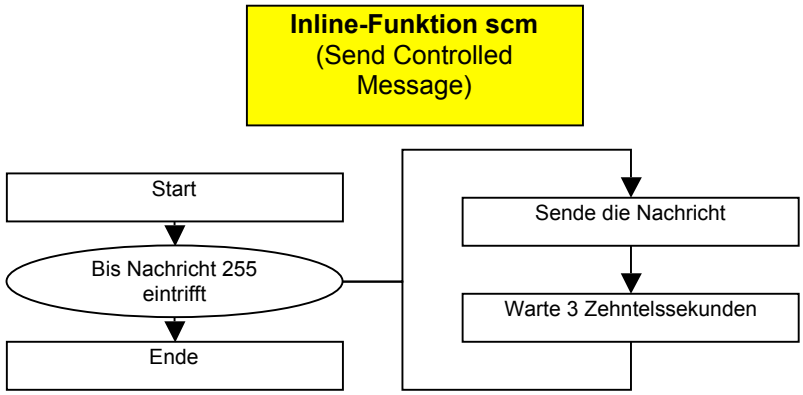
Als einfache Lösung bietet es sich an, die Nachricht in Impulsen auszusenden und es so einzurichten, dass, falls der Empfänger das Signal empfängt und dann zurücksendet, der Sender dank einer genug langen Wartezeit zwischen den Sendeeinheiten noch nicht am Senden ist.

Um diese Aufgabe zu erledigen, programmierte ich die Inline-Funktion *scm* (SendControlledMessage), da mehrmals Signale ausgetauscht werden. Der Inline-Funktion wird das Argument *message* übergeben; so weiss die Funktion, welche Nachricht sie senden muss.

```
// Funktion send controlled message

void scm (int message)
{
  ClearMessage ();
  do                          // Schleife, die Nachricht solange sendet,
  {                           // bis sie bestätigt wird.
    SendMessage (message);    // Sende die Nachricht.
    Wait (30);
  }
  while (Message () != 255);
}
```

Die Nachricht wird so lange gesendet, bis die Bestätigungsnachricht 255 eintrifft.



Grafik 10

Drei Zehntelssekunden reichen für den Empfänger, um die Nachricht zu empfangen und die Nachricht 255 zu senden. Zwischen dem Empfangen der Nachricht und dem Senden der Nachricht 255 wartet der Empfänger noch acht bis zehn

Hundertstelssekunden, um nicht schon zurückzusenden, wenn der Sender noch am Senden ist.
Durch die Inline-Funktion scm konnte ich die Übertragungssicherheit verbessern. Aber auch diese Art der Übertragung ist nicht 100 Prozent sicher, der Sender kann die Bestätigungsnachricht schlichtweg verpassen und sendet die Nachricht weiter, während der Empfänger sein Programm fortsetzt.

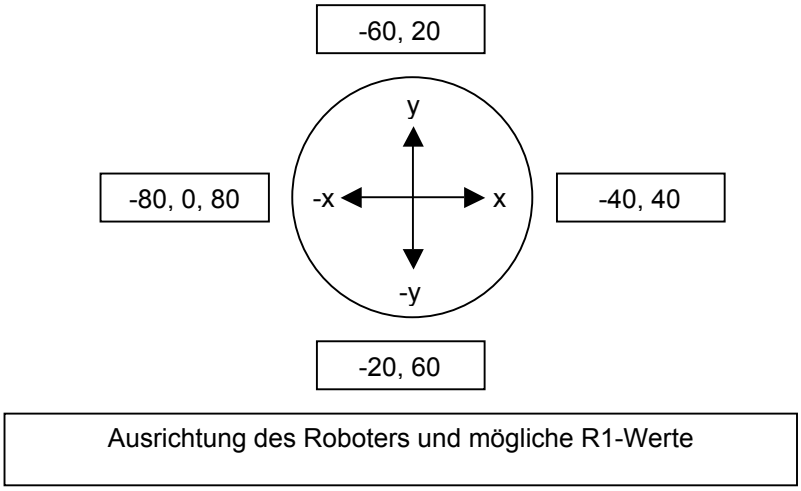
3.3.3.2 Detailtabelle der Teilfunktion examine

Master	Anzahl/Name	Beschreibung
Tasks	0	
Unterprogramme	0	
Inline-Funktionen	2	
	examine ()	Fährt den Roboter vorwärts zum Objekt und wieder zurück.
	scm (int message)	Sendet eine kontrollierte Nachricht.
Slave		
Tasks	0	
Unterprogramme	0	
Inline-Funktionen	2	
	examine ()	Untersucht das Objekt.
	scm (int message)	Sendet eine kontrollierte Nachricht.

Tabelle 14

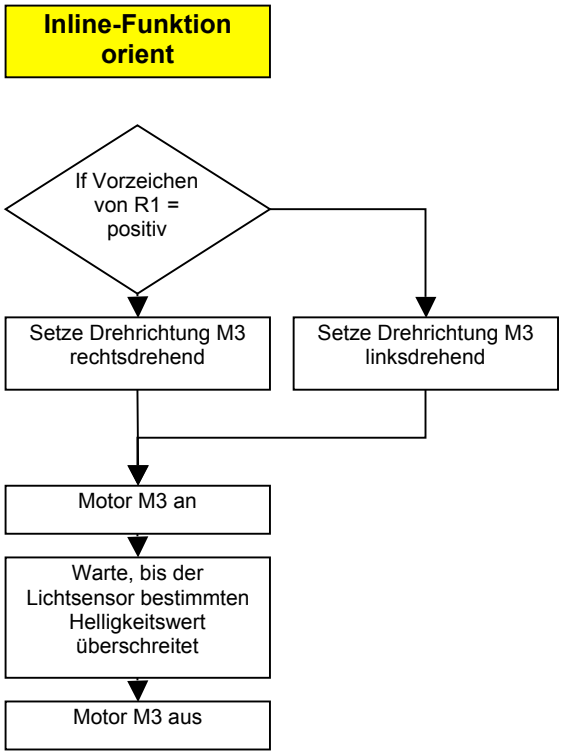
3.3.4 Die Teilfunktion orient

In welcher Richtung steht der Roboter gegenüber der Kompassnadel? Um das herauszufinden, existiert die Teilfunktion orient. Die Aufgabe wird von der Inline-Funktion *orient* übernommen. Wie wir schon wissen, wird die Position des Lichtsensors, der die Nadel misst, mit einem Rotationssensor gemessen. Dieser wird am Anfang so eingestellt, dass er den Wert null hat, falls der Roboter in negativer y-Richtung schaut. Da der Rotationssensor pro Lichtsensorumdrehung je nach Drehrichtung plus oder minus 80 zählt, entspricht auch der Rotationssensorwert 80 oder -80 einer Ausrichtung des Roboters in negativer y-Richtung.
Der Lichtsensor L2 dreht sich beim Suchen der Nadel immer in die Richtung, in die er den Betrag des Rotationssensorwertes verkleinert. Zeigt der Rotationssensor R1 zum Beispiel den Wert 50 an, wird L2 so drehen, dass die Werte des Rotationssensors abnehmen. Daraus folgt, dass der Betrag der Werte von R1 nie grösser als 80 werden kann.
In Grafik 11 sieht man die möglichen Rotationssensorwerte bei entsprechender Roboterausrichtung im Koordinatensystem.



Grafik 11

Der Ablauf der Inline-Funktion ist recht einfach:



Grafik 12

Etwas heikler war es schon, die Helligkeit der Kompassnadel zu bestimmen. Leider ist die Nadel auf beiden Seiten fast gleich hell, die Nordnadel hat im Gegensatz zur völlig weissen Südnadel als einziges Unterscheidungsmerkmal einen roten Strich.

Einstellen von R1 beim Start

Da der Roboter beim Start nicht weiss, wie die Magnetfeldlinien gegenüber dem Koordinatensystem stehen, richtet er sie beim Start gegeneinander aus. Der Kompass stellt sich beim Start des *maintasks* auf die hellere Nadelseite ein. Das funktioniert so, dass er erstens die Inline-Funktion *orient* ausführt. L2 ist jetzt über der Nadel. Da der Roboter beim Start in negativer y-Richtung steht, wird der Wert von R1 auf 60 gesetzt. Von nun an ist R1 auf das Koordinatensystem der Linien am Boden ausgerichtet. Das hat den grossen Vorteil, dass der Roboter unabhängig von der

Ausrichtung des Koordinatensystems gegenüber den magnetischen Polen funktioniert.
Um den nötigen Wert zur Erkennung einer Nadel zu ermitteln, baute ich ein kleines Programm.

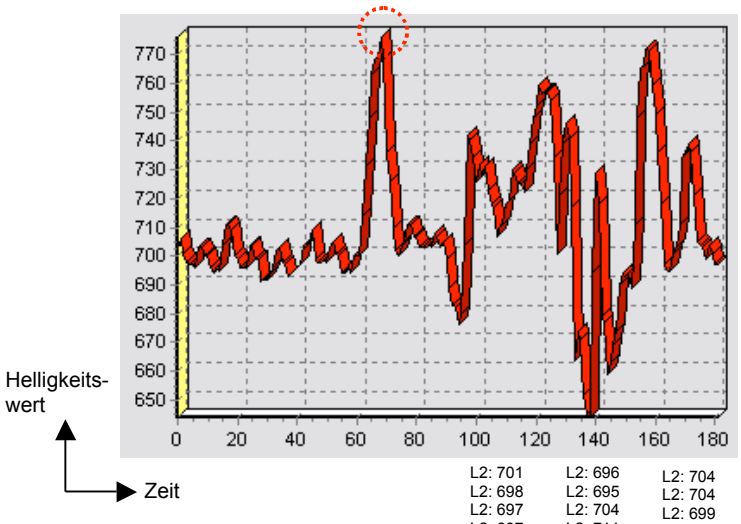
```
// Kompass analysieren

task main ()
{
  // Sensoren definieren
  SetSensorType (SENSOR 1, SENSOR TYPE LIGHT);
  SetSensorMode (SENSOR 1, SENSOR MODE RAW);
  SetSensor (SENSOR 2, SENSOR ROTATION);
  SetDirection (OUT B, OUT FWD); // Dreht die Drehrichtung
  ClearSensor (SENSOR 2);        // R1 auf 0 setzten (Es steht SENSOR_2,
                                 // weil er an Eingang 2 ist.)
  CreateDatalog (200);           // Einen Speicherblock mit 200 Werten erstellen
  On(OUT B);                     // Motor starten
  until (abs (SENSOR_2) >= 80)   // Drehbedingung für eine Umdrehung
  {
    AddToDatalog (SENSOR 1);     // Speichere den Lichtsensorwert
    Wait (4);                    // Warte 4/100 sek
  }
  Off (OUT_B);                   // Motor aus

}
```

Mit den Befehlen *CreateDatalog* und *AddToDatalog* werden im RCX Werte einem Speicher zugeordnet, der dann vom PC ausgewertet werden kann.
Ich liess L2 eine Runde über dem Kompass kreisen. Dies ergab folgende Werte:

Diagramm Lichtsensor über dem Kompass (einer Umdrehung)



Helligkeitswert des Lichtsensors

L2: 698	L2: 736	L2: 718	L2: 729
L2: 697	L2: 726	L2: 717	L2: 718
L2: 693	L2: 710	L2: 707	L2: 679
L2: 694	L2: 702	L2: 711	L2: 677
L2: 695	L2: 700	L2: 711	L2: 659
L2: 696	L2: 704	L2: 718	L2: 663
L2: 697	L2: 705	L2: 719	L2: 669
L2: 704	L2: 711	L2: 730	L2: 674
L2: 703	L2: 711	L2: 729	L2: 690
L2: 705	L2: 707	L2: 725	L2: 694
L2: 709	L2: 707	L2: 725	L2: 693
L2: 699	L2: 703	L2: 722	L2: 694
L2: 698	L2: 703	L2: 730	L2: 689
L2: 698	L2: 703	L2: 739	L2: 711
L2: 699	L2: 704	L2: 748	L2: 748
L2: 698	L2: 703	L2: 752	L2: 765
L2: 703	L2: 708	L2: 759	L2: 766
L2: 705	L2: 707	L2: 756	L2: 772
L2: 701	L2: 703	L2: 757	L2: 771
L2: 700	L2: 705	L2: 757	L2: 759
L2: 694	L2: 699	L2: 754	L2: 749
L2: 694	L2: 689	L2: 734	L2: 736
L2: 697	L2: 683	L2: 701	L2: 718
L2: 694	L2: 682	L2: 721	L2: 697
L2: 700	L2: 676	L2: 739	L2: 695
L2: 700	L2: 680	L2: 746	L2: 695
L2: 700	L2: 704	L2: 741	L2: 701
L2: 714	L2: 742	L2: 717	L2: 703
L2: 731	L2: 740	L2: 664	L2: 711
L2: 748	L2: 734	L2: 674	L2: 734
L2: 763	L2: 726	L2: 671	L2: 736
L2: 768	L2: 732	L2: 652	L2: 739
L2: 769	L2: 728	L2: 645	L2: 736
L2: 776	L2: 731	L2: 644	L2: 721
L2: 768	L2: 723	L2: 694	L2: 705

L2: 701	L2: 696	L2: 704
L2: 698	L2: 695	L2: 704
L2: 697	L2: 704	L2: 699
L2: 697	L2: 711	L2: 699
L2: 696	L2: 711	L2: 691
L2: 703	L2: 709	L2: 692
L2: 703	L2: 700	L2: 692
L2: 700	L2: 699	L2: 694
L2: 700	L2: 696	L2: 696
L2: 696	L2: 696	L2: 702
L2: 694	L2: 696	L2: 703

Grafik 13

Anhand des Diagramms und der Werte sieht man, dass der Lichtsensorwert nur bei einer Nadelspitze über den Wert 770 spring.

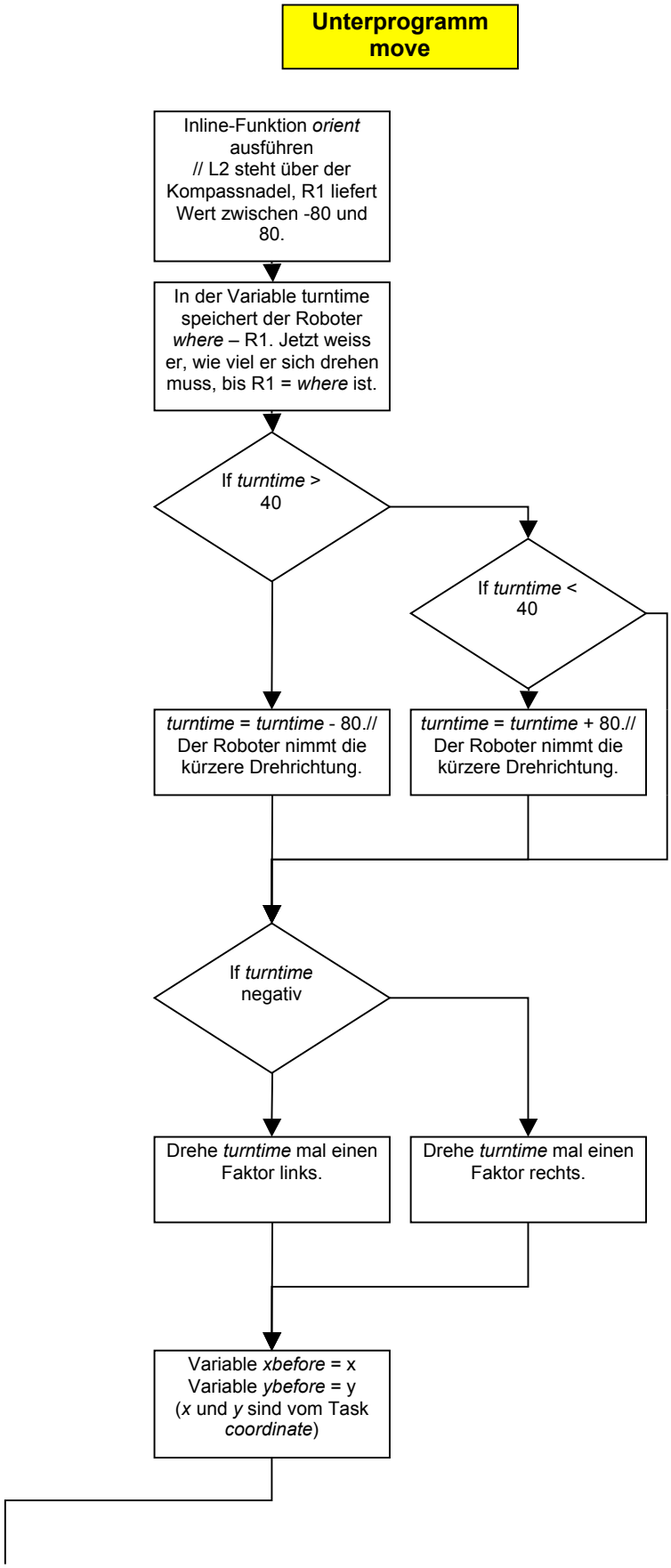
3.3.4.1 Detailtabelle Teilfunktion orient

Master	Anzahl/Name	Beschreibung
Tasks	0	
Unterprogramme	0	
Inline-Funktionen	1	
	orient ()	Sie findet heraus, wie der Roboter gegenüber der Kompassnadel im Raum steht.

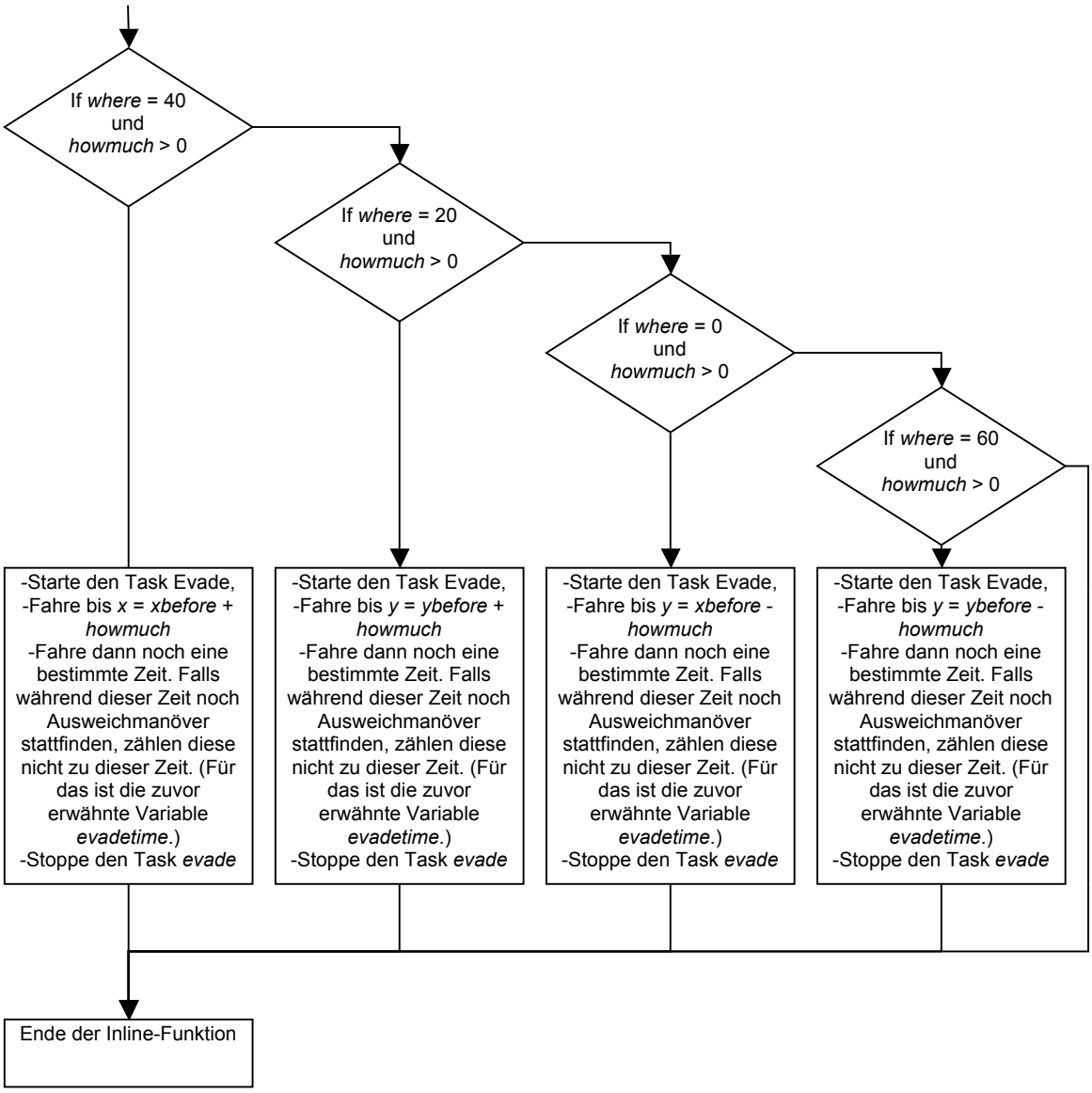
Tabelle 15

3.3.5 Die Teilfunktion move

Sämtliche Bewegungen des Roboters werden von der Teilfunktion move übernommen. Ihr Hauptbestandteil ist das Unterprogramm *move*. In einem ersten Versuch löste ich das Problem mit einer Inline-Funktion. Der Vorteil einer Inline-Funktion ist, dass ihr bei ihrem Aufruf Werte übergeben werden können. Die Funktion wird so definiert: ***void*** *move (**int** where, **int** howmuch).* Ich übergab ihr die Werte *where* und *howmuch. Where* sagt der Inline-Funktion, wohin sie fahren muss, und *howmuch,* wie lange. Das Problem war aber nun, dass diese Inline-Funktion im Code 26-mal aufgerufen wird. Dies bedeutet, dass sie beim Kompilieren und anschliessenden Übertragen 26-mal auf dem RCX gespeichert wird. Der Programmcode war zu lange. Ein Unterprogramm wird nur einmal auf den RCX geschrieben, hat aber den Nachteil, dass ihm keine Argumente übergeben werden können. Ich umgehe das Problem so: die zu übertragenden Werte speichert das Programm nun in den globalen Variablen *where* und *howmuch*. Bevor jetzt die Unterfunktion *move* im Code aufgerufen wird, werden die benötigten Werte in die globalen Variablen gespeichert. Das Unterprogramm bedient sich dann deren. Soll sich der Roboter bspw. nur drehen, ist *howmuch* 0. Für *where* kommen die Werte 0, 20, 40, 60 in Frage. Was sie bedeuten, sieht man in Grafik 11. Je höher die Zahl in *howmuch* ist, desto mehr Felder fährt der Roboter, nachdem er sich nach *where* ausgerichtet hat. Sobald der Roboter in Bewegung ist, startet er zusätzlich den Ausweichtask *evade.* Dieser sorgt dafür, dass der Roboter allfälligen Hindernissen ausweicht. Wird zum Beispiel während der Fahrt Berührungssensor B1 gedrückt, dreht sich der Roboter so lange auf der Stelle, bis B1 nicht mehr gedrückt wird. Diese Drehzeit wird immer in der Variable *evadetime* gespeichert. Der Roboter soll nach dem überquer einer Koordinatenlinie immer gleich weit ins Koordinatenfeld hineinfahren. Da das zeitabhängig ist, wird mit Hilfe der Variable *evadetime* die Zeit eines Ausweichmanövers nicht mit einberechnet.


Unterprogramm move
Inline-Funktion *orient* ausführen // L2 steht über der Kompassnadel, R1 liefert Wert zwischen -80 und 80.
In der Variable turntime speichert der Roboter *where* – R1. Jetzt weiss er, wie viel er sich drehen muss, bis R1 = *where* ist.
If *turntime* > 40
If *turntime* < 40
turntime = *turntime* - 80.// Der Roboter nimmt die kürzere Drehrichtung.
turntime = *turntime* + 80.// Der Roboter nimmt die kürzere Drehrichtung.
If *turntime* negativ
Drehe *turntime* mal einen Faktor links.
Drehe *turntime* mal einen Faktor rechts.
Variable *xbefore* = x
Variable *ybefore* = y
(*x* und *y* sind vom Task *coordinate*)

Grafik 14



Grafik 15

3.3.5.1 Detailtabelle der Teilfunktion move

Master	Anzahl/Name	Beschreibung
Tasks	1	
	evade	Weicht, während der Roboter fährt, Hindernissen aus.
Unterprogramme	1	
	move	Führt sämtliche Bewegungen des Roboters aus.
Inline-Funktionen	0	

Tabelle 16

3.3.6 Die Teilfunktion drive_to_aim

Die Aufgabe der Teilfunktion ist es, die erhaltenen Wegpunkte der Reihe nach mit der Teilfunktion move anzufahren. Sie überlegt sich, wie sie diesen Weg fährt, fährt ihn und richtet sich am Schluss noch nach dem Ziel aus.
Die Teilfunktion *drive_to_aim* besteht aus der Inline-Funktion *drive_to_aim*. Diese bedient sich häufig der Inline-Funktion *drive_to_coordinate*. Die Inline-Funktion drive_to_coordinate fährt den Roboter zu den nächsten Koordinaten. Diese Koordinaten sind in bis zu neun Variablen abgespeichert. Bis zu drei Wegpunktkoordinaten und eine Endkoordinate mit Ausrichtung können gespeichert werden.

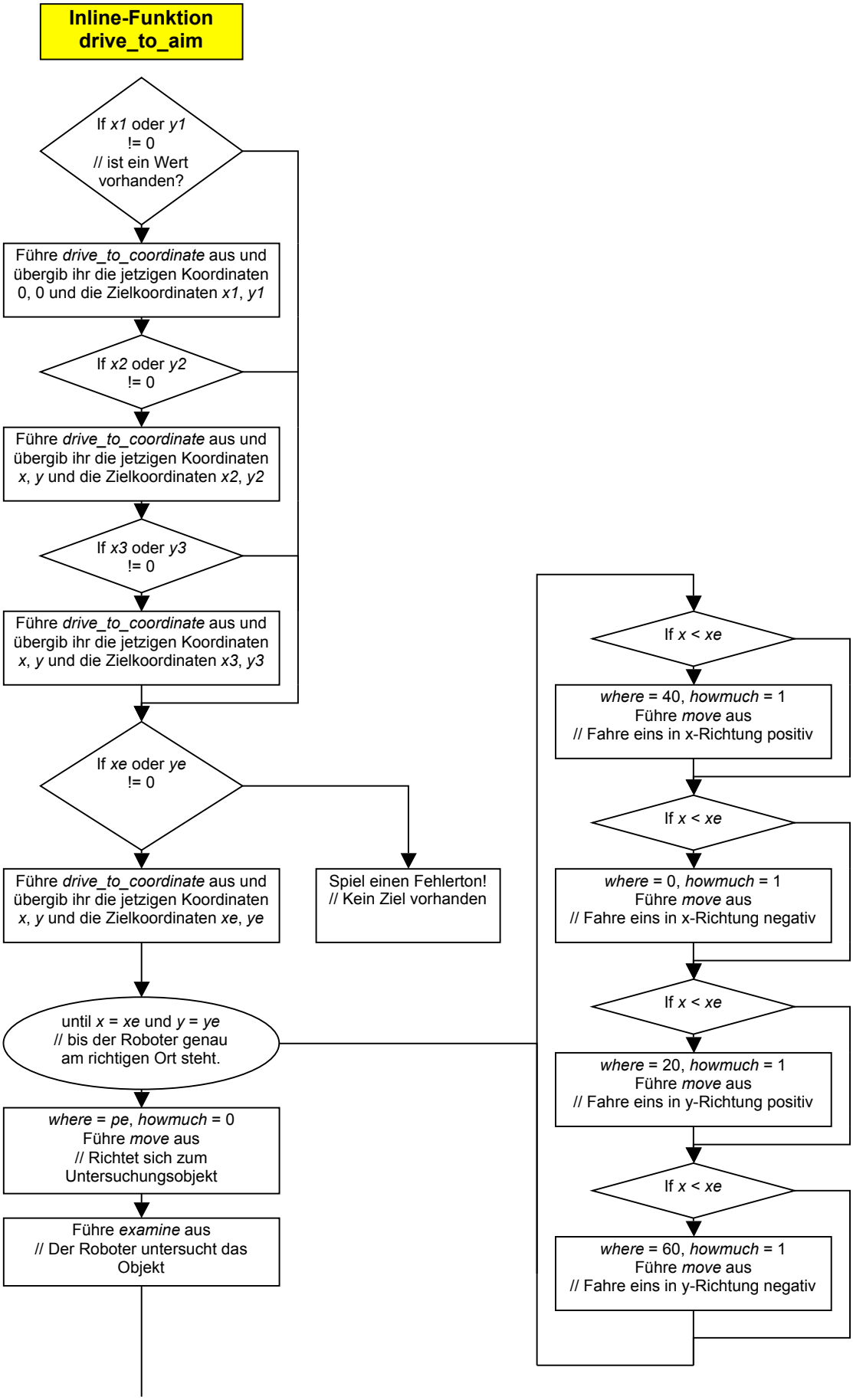
Variablen	Bedeutung
x1, *y1*	Koordinaten des ersten Wegpunktes
x2, *y2*	Koordinaten des zweiten Wegpunktes
x3, *y3*	Koordinaten des dritten Wegpunktes
xe, *ye*, *pe*	Zielkoordinaten und Ausrichtung

Tabelle 17

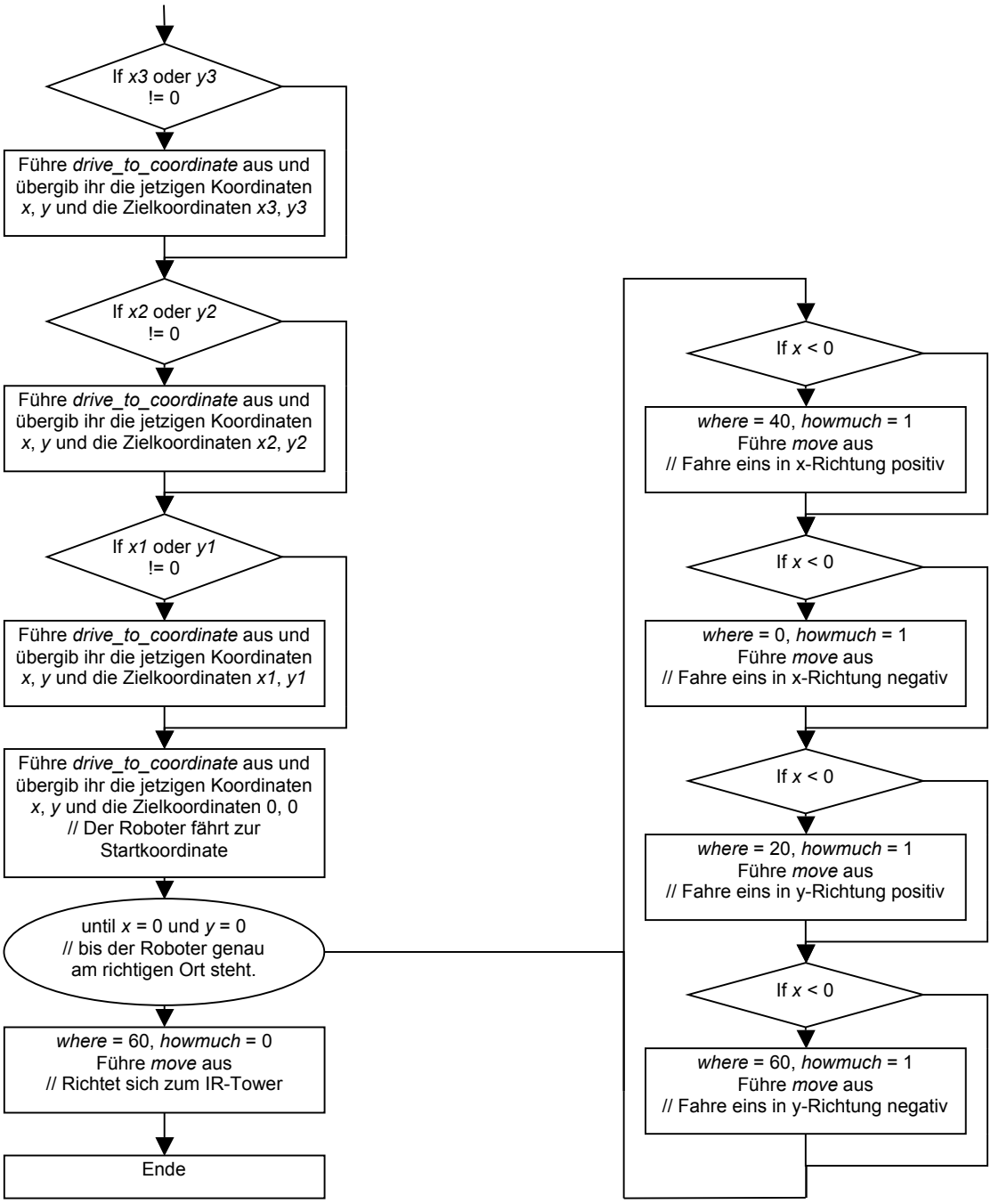
Die Wegpunkte eins, zwei und drei sind fakultativ; ist nicht einmal der erste Wegpunkt gegeben, fährt der Roboter direkt zum Ziel. Beim Rückweg fährt der Roboter die Koordinaten in umgekehrter Reihenfolge an, fährt am Schluss zur Koordinate 0, 0 und richtet sich gegen den IR-Tower aus.
Der Inline Funktion *drive_to_coordinate* werden 4 Argumente übergeben. Es sind dies die aktuelle Position (*nowx*, *nowy*) und die Koordinaten, die sie erreichen soll (*aimx*, *aimy*).
Definiert wird sie so:

void *drive_to_coordinate (**int** aimx, **int** aimy, **int** nowx, **int** nowy).*

Falls der Roboter das Ziel etwas verfehlen sollte, korrigiert er sich am Schluss, indem er so lange in eine entsprechende Richtung fährt, bis die Variablen *x*, *y* mit den Zielvariablen übereinstimmen.


Inline-Funktion drive_to_aim
If x1 oder y1 != 0 // ist ein Wert vorhanden?
Führe drive_to_coordinate aus und übergib ihr die jetzigen Koordinaten 0, 0 und die Zielkoordinaten x1, y1
If x2 oder y2 != 0
Führe drive_to_coordinate aus und übergib ihr die jetzigen Koordinaten x, y und die Zielkoordinaten x2, y2
If x3 oder y3 != 0
Führe drive_to_coordinate aus und übergib ihr die jetzigen Koordinaten x, y und die Zielkoordinaten x3, y3
If xe oder ye != 0
Führe drive_to_coordinate aus und übergib ihr die jetzigen Koordinaten x, y und die Zielkoordinaten xe, ye
Spiel einen Fehlerton! // Kein Ziel vorhanden
until x = xe und y = ye // bis der Roboter genau am richtigen Ort steht.
where = pe, howmuch = 0 Führe move aus // Richtet sich zum Untersuchungsobjekt
Führe examine aus // Der Roboter untersucht das Objekt
If x < xe
where = 40, howmuch = 1 Führe move aus // Fahre eins in x-Richtung positiv
If x < xe
where = 0, howmuch = 1 Führe move aus // Fahre eins in x-Richtung negativ
If x < xe
where = 20, howmuch = 1 Führe move aus // Fahre eins in y-Richtung positiv
If x < xe
where = 60, howmuch = 1 Führe move aus // Fahre eins in y-Richtung negativ

Grafik 16

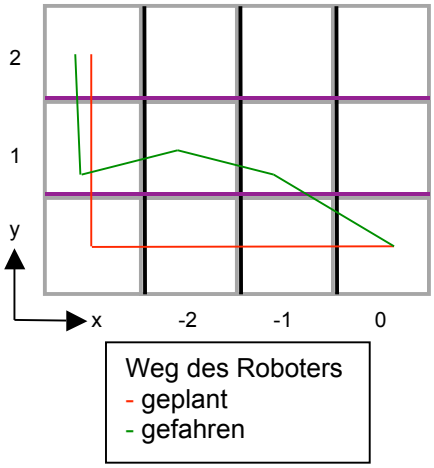


Grafik 17

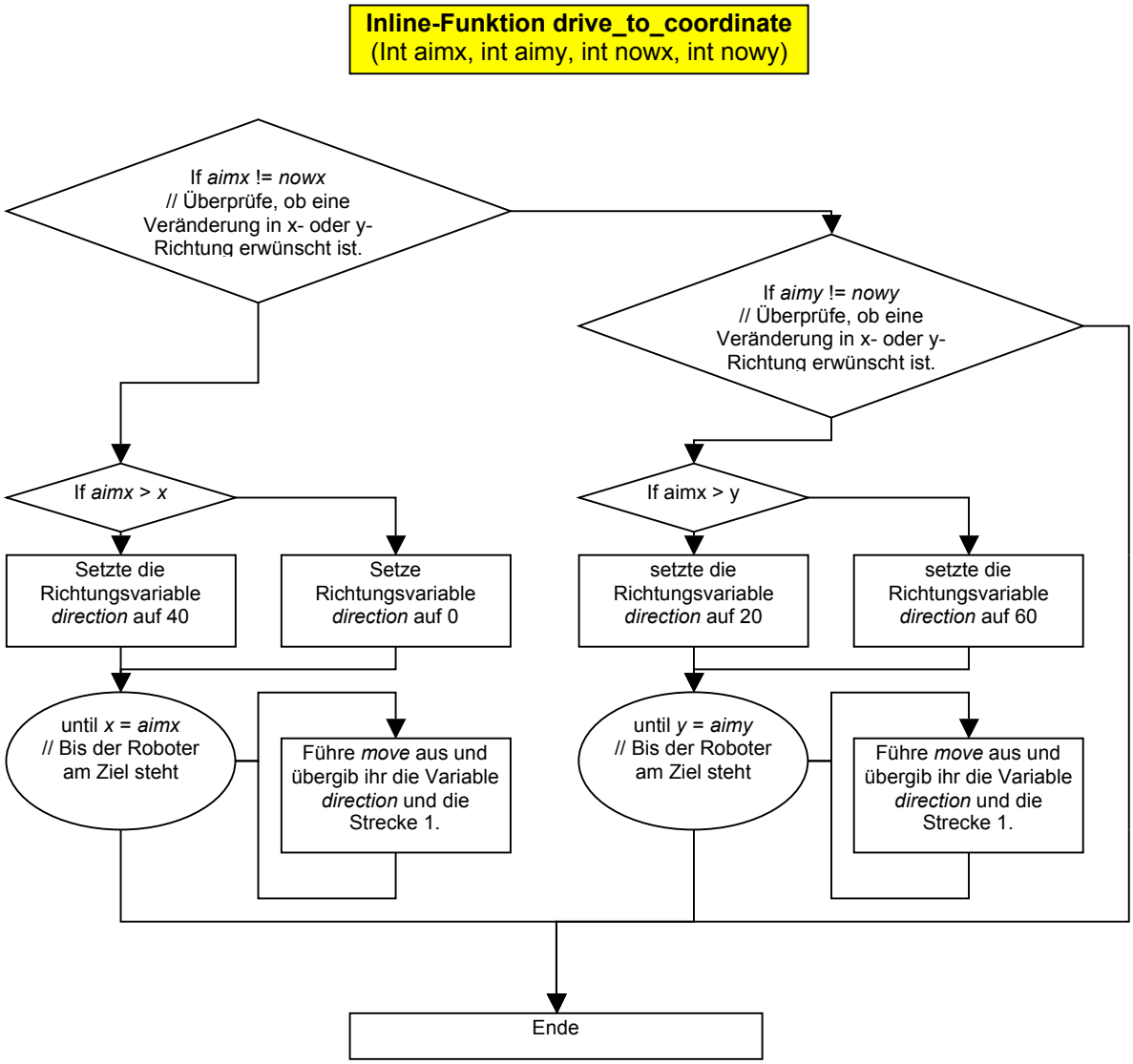
3.3.6.1 Die Inline-Funktion drive_to_coordinate

Der Code der Inline-Funktion drive_to_coordinate ist etwas kompliziert, daher folgt eine etwas ausführliche Erklärung:
Die Funktion stellt zuerst fest, ob der nächste Koordinatenpunkt eine Verschiebung auf der x- oder y-Achse erfordert. Dann ermittelt sie die Fahrrichtung. Mit Hilfe der Inline-Funktion fährt sie so lange, bis die aktuelle Achsenposition der geforderten

entspricht. Die Funktion hat den grossen Vorteil, dass Richtungsungenauigkeiten beim Fahren ausgeglichen werden können
Ein Beispiel: Der Roboter startet bei 0, 0. Zuerst soll er die Koordinate -3,0, dann die Koordinate -3,2 erreichen. Er fährt in Richtung 40 los, ist aber etwas ungenau, so dass er beim Erreichen von ***x***=-3 schon eine y Linie überquert hat, er befindet sich beim Punkt -3,1 anstatt -3.0. Dies ist aber nicht tragisch, da er einfach in Richtung 20 weiterfährt bis der *y*-Wert zwei ist und so das Ziel von -3,2 erreicht. Fahrfehler beim Ausführen der Funktion werden also bei der nächsten Ausführung ausgeglichen. Falls der Roboter die Endkoordinate nicht ganz erreicht, sucht er diese wie in Grafik 17 beschrieben am Schluss exakt auf. Da die Inline-Funktion *move* für jede Koordinatenlinie neu aufgerufen wird, spielt es auch keine Rolle, wenn der Roboter einmal in eine komplett falsche Richtung fährt, bei der nächsten Ausführung von *move* fährt er wieder in die richtige Richtung.



Grafik 18



Grafik 19

3.3.6.2 Detailtabelle der Teilfunktion drive_to_aim

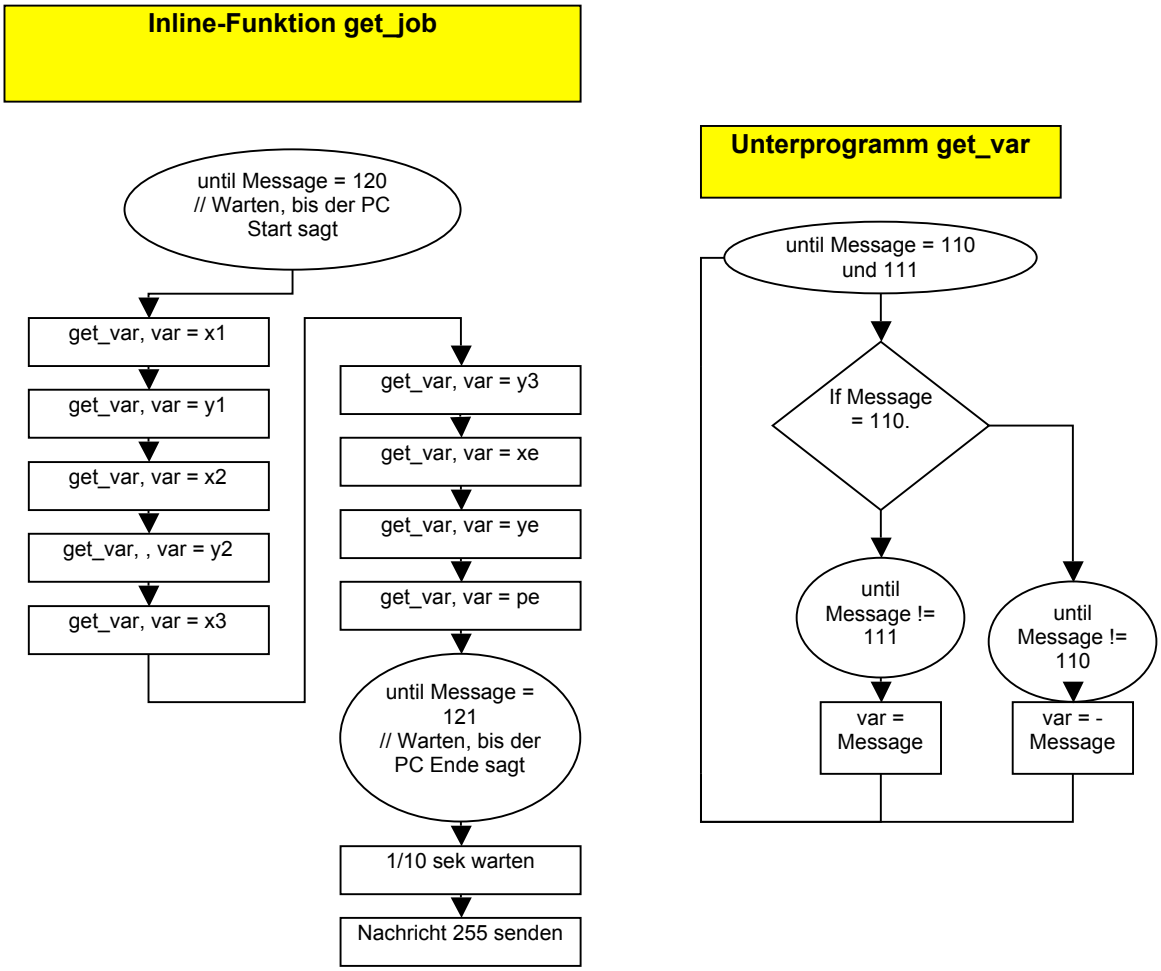
Master	Anzahl/Name	Beschreibung
Tasks	0	
Unterprogramme	0	
Inline-Funktionen	2	
	drive_to_aim ()	Fährt den Roboter zum Ziel und wieder zurück.
	drive to coordinate (int aimx, int aimy, int nowx, int nowy)	Fährt den Roboter zum nächsten Weg- oder Zielpunkt.

Tabelle 18

3.3.7 Die Teilfunktion get_job

Beim Start muss der Roboter wissen, wo er hinfahren muss. Diese Informationen erhält er vom PC. Die Teilfunktion get_job sorgt dafür, dass die Koordinatenvariablen

mit Koordinaten vom PC gefüllt wird. Damit der Maintask nicht allzu lange wird, schrieb ich den Code für diese Aufgabe in die Inline-Funktion *get_job*.



Grafik 20

Ablauf:
Der Roboter wartet, bis die Nachricht 120 eintrifft. Anschliessend führt er neunmal das Unterprogramm get_var durch und setzt jeweils die Koordinatenvariablen gleich der Variable *var*. Das Unterprogramm setzt die erhaltenen Nachrichten, bestehend aus Vorzeichen und Wert, in die Variable *var*. Am Schluss wartet der Roboter auf die Nachricht 121, die das Ende der Übermittlung bedeutet, und bestätigt diese.

3.3.7.1 Detailtabelle der Teilfunktion get_job

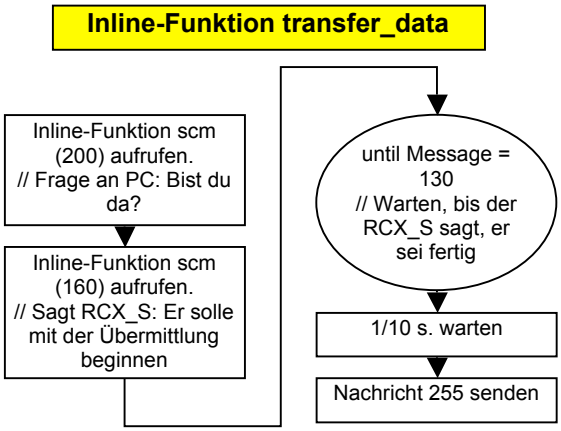
Master	Anzahl/Name	Beschreibung
Tasks	0	
Unterprogramme	1	
	get_var	Empfängt den Wert einer Koordinate und deren Vorzeichen.
Inline-Funktionen	1	
	get_job ()	Empfängt die Koordinaten der Weg- und

		Zielpunkte vom PC.

Tabelle 19

3.3.8 Die Teilfunktion transfer_data

Die Aufgaben der Teilfunktion transfer_data werden von der gleichnamigen Inline-Funktion transfer_data übernommen:



Grafik 21

Zuerst sendet der RCX_M dem PC ein Signal zum Testen, ob der IR-Tower in Reichweite ist. Anschliessend sendet er dem RCX_S ein Signal, damit dieser die gesammelten Werte dem PC übermittelt. Dazu benötigt der RCX_S die Inline-Funktion transfer_data_s (Siehe dazu Grafik 9). Am Schluss gibt der Roboter ein akustisches Signal von sich. Sobald der Beobachter dieses Signal hört, weiss er, dass die Mission erfolgreich verlief.

3.3.8.1 Detailtabelle der Teilfunktion transfer_data

Master	Anzahl/Name	Beschreibung
Tasks	0	
Unterprogramme	0	
Inline-Funktionen	1	
	transfer_data ()	Überprüft, ob der IR-Tower in Reichweite ist, und sagt dem Slave, er solle die Untersuchungsdaten übermitteln.
Slave		
Tasks	0	
Unterprogramme	0	
Inline-Funktionen	1	
	transfer_data_s ()	Sendet dem PC die Untersuchungsdaten.

Tabelle 20

4 Fazit

Aufgaben

(Die Aufgabenstellung ist kursiv vor die Beurteilung der Lösung gestellt)

4.1 Hauptaufgabe

- „Der User gibt am PC die Koordinaten eines Untersuchungsobjekts sowie einige Wegpunkte ein. Der PC sendet dem Roboter diese Koordinaten. Anschliessend fährt der Roboter zum Untersuchungsobjekt, bohrt mit einem Bohrer ein Loch hinein und misst im Loch die Temperatur. Das Gestein, das beim Bohren herunterfällt, fängt der Roboter auf, befördert es in ein Untersuchungsbecken und untersucht es auf seine Helligkeit. Anschliessend fährt der Roboter zurück und bringt so das Untersuchungsmaterial zur Basisstation. Dort sendet er die gemessenen Werte an den PC.“

Die Hauptaufgabe wurde gelöst. Optimierungen könnten sicher im Bereich der Zuverlässigkeit und der Arbeitsgeschwindigkeit gemacht werden.

4.2 Teilaufgaben

- *„1. Der Roboter muss sich orientieren können. Für diesen Zweck bringe ich am Boden Markierungen an, die ein Lichtsensor wahrnimmt. Die erhaltenen Werte werden durch den Roboter verarbeitet.“*

Die Teilaufgabe 1 wurde gut erreicht, Linien können mit einer Sicherheit von ungefähr 95 Prozent richtig eingelesen werden.

- *„2. Zusätzlich muss der Roboter immer wissen, wie er im Raum steht. Dies geschieht mit Hilfe eines Kompasses.“*

Theoretisch ist die Teilaufgabe 2 gelöst, in der Praxis zeigen sich jedoch zwei Probleme: Erstens dreht sich der Roboter nicht immer gleich schnell, so dass die Ausrichtung nicht sehr präzise ist. Zweitens verwechselt der Lichtsensor manchmal die beiden Nadelspitzen, was dazu führt, dass der Roboter ein Feld in die falsche Richtung fährt.

- *„3 Der Roboter muss kontrolliert in eine Richtung fahren und sich wenden können.“*

Die 3. Teilaufgabe wurde etwas speicherplatzaufwendig gelöst.

- *„4. Der Roboter muss das Objekt untersuchen können.“*

Das Problem in Teilaufgabe 4 ist, dass die RCX’s nicht immer fehlerfrei kommunizieren, ansonsten wurde die Aufgabe gelöst.

- *„5. Der Roboter muss anhand von Wegpunkten einen Weg zum Ziel fahren.“*

Der ganze Ablauf läuft etwas langsam, da nach jeder Vorwärtsbewegung um ein Feld die Richtung neu bestimmt wird, ansonsten wurde sie gut gelöst.

- *„6. Der Roboter muss die Koordinaten des Untersuchungsobjekts und der Wegpunkte empfangen können.“*

Die Koordinaten werden richtig eingelesen.

- *„7. Der Roboter muss die über das Untersuchungsobjekt gefundenen Informationen an den PC zurücksenden können.“*

Die Werte werden zum PC übertragen.

4.3 Nicht erreichte Aufgaben

Am Anfang meiner Planung für dieses Roboterprojekt stand eine zweite, noch fast wichtigere Aufgabe im Raum. Sie hiess explore. Die Idee war folgende: Der Roboter erkundet von seinem Startpunkt aus die Umgebung und speichert dabei die Standorte von Hindernissen und von für spätere Untersuchungen lohnenswerten Objekten. Anschliessend sendet er deren Koordinaten dem PC. Der PC macht daraus mit Hilfe einer Software eine Miniaturkarte der Umgebung. Der PC-User kann nun einfach ein Untersuchungsobjekt anklicken und der Roboter untersucht es. Falls diese zweite Aufgabe zustande gekommen wäre, hätte die jetzige Aufgabe search and examine noch eine weitere Teilaufgabe gehabt. Sie war die ehemalige Aufgabe acht.

- 8. Der Roboter muss den kürzesten Weg zum Ziel durch Hindernisse finden und diese in Koordinaten speichern.

Sie konnte nicht realisiert werden, weil der Roboter ohne die Hauptaufgabe explore keine Hindernisse kennt, und sich somit auch keinen Weg suchen kann.
Für die Hauptaufgabe explore waren drei Teilaufgaben aufgestellt.

- 9. Der Roboter muss ein vorgegebenes Gebiet erkunden und die Koordinaten von Hindernissen sowie von Untersuchungsobjekte speichern.
- 10. Der Roboter muss die Koordinaten dieses Untersuchungsgebietes vom PC empfangen.
- 11. Der Roboter muss die Koordinaten der lohnenswerten Untersuchungsobjekte an den PC senden.“

Warum erfüllte ich die Hauptaufgabe explore nicht?
Folgende Probleme stellten sich mir in den Weg:

- Der RCX verfügt nicht über genügend Speicherplatz, um sich Hindernisse zu merken.
- Ich war mit der Aufgabe search und examine so beschäftigt, dass weder die Zeit, noch meine Programmierkenntnisse ausgereicht hätten, um sie zu erfüllen.

Anfänglich überlegte ich mir noch einige Methoden, wie der Roboter ein vorgegebenes Feld erkunden könnte. Sie befinden sich im Anhang unter Grafiken.

4.4 Verbesserungen

Mein Roboter ist sicher verbesserungsfähig.
Einige Verbesserungsvorschläge:

- Es wäre sicher einen Vorteil, die Kompassnadel von zwei nebeneinander platzierten Lichtsensoren zu überwachen. Damit könnte der Roboter feststellen, in welche Richtung die Kompassnadel dreht und seine Fahrtrichtung laufend korrigieren.
- Auch der Boden würde vorteilhaft mit zwei Lichtsensoren abgesucht. Die Linienerkennungssicherheit würde dadurch erhöht.

5 Verzeichnisse

5.1 Literaturverzeichnis:

Quellen zu Punkt 2.2 Lego RCX 1.0 mit NQC Umgebung:

http://lug.mfh-iserlohn.de/lego/NQC_2.0r2.de.html
Titel: NQC Programmieranleitung
Version: 2.0 rev 2
Autor: Dave Baum
Titel des englischen Originals: NQC Programmer's Guide
Downloaddatum: 18.10.04

http://scholl.be.schule.de/faecher/inform/material/mindstorm/pdf/tutorial_d.pdf
Titel: Programmieren von LegoMindstorms-Robotern mit NQC
Version: Version 3.03
Autor: Mark Overmars
Deutsche Übersetzung: Martin Breuner
Downloaddatum: 18.10.04

5.2 Tabellenverzeichnis:

5.3 Verwendeter Editor

Um den NQC-Quellcode für den RCX zu schreiben und ihn dann an den PC zu übertragen, verwendete ich die Freeware Brixc Commande Center 3.3. Sie ist unter http://bricxcc.sourceforge.net/ zu beziehen.

6 Nachwort

6.1 Mein Kommentar zur Arbeit

Für mich war das Ziel dieser Maturarbeit, durch das Programmieren eines Roboters einen Einblick in die Welt des Programmierens zu erhalten. Dies ist sicher auch geschehen. Diese Arbeit war mit ein Grund, dass ich mich während den Sommerferien intensiv mit der Websiteprogrammiersprache PHP auseinandersetzte. Dabei lernte ich Grundzüge einer Programmiersprache kennen. Dies waren zum Beispiel die If-Anweisung oder dass es verschiedene Variablentypen (Integer, String, usw.) gibt. Auf der anschliessenden Suche nach einer für meinen Roboter passenden Programmiersprache lernte ich, dass es verschiedene Arten gibt, einen Roboter zu programmieren. Grafik- oder textbasierendes Programmieren standen zur Auswahl, selbst objektorientiertes Programmieren wäre möglich gewesen.
Da ich mich für NQC entschied, musste ich anschliessend die Handhabung dieser Sprache lernen. Anhand von vorgegebenen Beispielen und Erklärungen gelang es mir, ein Überblick über deren Möglichkeiten zu gewinnen. Das anschliessende Zusammenbauen des Roboters war eine gemütliche Angelegenheit und weckte Erinnerungen an meine Kindheit. Trotzdem war es nicht einfach, alle diese Sensoren und Motoren in meinem Roboter einzubauen. Planen, bauen, ausprobieren, auseinander nehmen und wieder zusammenbauen waren die Hauptbeschäftigungen. Nach diesem praktischen Teil programmierte ich wieder. Beim Suchen nach Lösungswegen für meine Aufgaben erkannte ich zum ersten Mal, wie schwierig Programmieren sein kann. Besonders die schwankenden Werte von Sensoren, sowie Probleme bei der Infrarotübertragung machten mir zu schaffen, manchmal war es auch nur ein Tippfehler. Die wenigsten Programme funktionierten auf Anhieb, Überlegungsfehler waren auch Gründe dafür. Nach vielen Versuchen musste ich mich mit den bestmöglichen Resultaten zufrieden geben. Jede Lösung eines Problems, insbesondere auch das Suchen einer Lösung brachte mir mehr Erfahrung im Programmieren. Als nun die Teilfunktionen auf dem Roboter funktionierten und ich alles zusammen auf den RCX übertragen wollte, sendete der Compiler mir folgende Meldung: „*Compile Failed, not enough room in RCX program memory*“. Diese Meldung brachte mich etwas ins Schwitzen. Aber auch für dieses Problem fand ich eine Lösung, ich hatte ersten Kontakt mit der Tätigkeit „Codeoptimierung“.
Die Hauptarbeit für die Maturaarbeit begann aber eigentlich schon früher und zog sich durch die ganze Zeit dieses Projekts: Kopfarbeit.
Zuerst überlegte ich mir Aufgaben, diese mussten immer wieder den Möglichkeiten angepasst werden. Dann beschäftigte ich mich mit dem Roboter. Ich machte mir Gedanken über sein Aussehen, über die Art der Sensoren und wie die Mechanik funktionieren könnte. Aber auch über die Realisierungsmöglichkeiten meiner Aufgaben im Bereich des Programmierens machte ich mir viele Gedanken. Das schwierigste Problem war, dem Roboter das Wissen um seine Position beizubringen. Bei all diesem Überlegen vergass ich leider manchmal fast, dass ein schriftliches Dokument das Ziel dieser Arbeit ist. So war die Zeit vor dem Abgabetermin manchmal etwas monoton und anstrengend. Bei einer zweiten Aufgabe dieser Art würde ich systematischer vorgehen.

6.2 Persönliche Bilanz

Ich erwähne, was mir am Roboterprogrammieren besonders positiv- und was besonders negativ auffiel, dies auch in Ausblick auf Punkt „Zukunft“.

Positives

- Meine Programmierkenntnisse wurden verbessert.
- Ich konnte mich mit der Thematik Roboterprogrammierung auseinandersetzen.
- Ich konnte mir Gedanken zu Problemen machen und sogar etwas Erfinden.

Negatives

- Ich arbeitete manchmal bis zu 10 Stunden pro Tag am PC. (Bei einer zweiten Durchführung würde ich die Arbeit am PC besser verteilen.)
- Allgemein fehlte mir etwas der Bezug zur Natur, auch Sport kam zu kurz.

6.3 Zukunft

Was ich bis jetzt noch nicht erwähnt habe, ist, dass ich diese Arbeit auch ein wenig im Ausblick auf die Berufswahl durchführte. Der Beruf des Robotikers weckt bei mir schon seit einigen Jahren ein gewisses Interesse. Bei genügend Freizeit, in der ich anderen Hobbies nachgehen kann, könnte er bei mir einmal Realität werden.

7 Dank

- Speziellen Dank geht an Herrn Fankhauser, der mich bei dieser Arbeit betreute.
- Auch danken möchte ich dem Gymnasium Burgdorf, speziell Herrn Aebi und Herr Stulz, die mir Ihre Lego Mindstorms Bausätze zur Verfügung stellten.
- Michael Christen möchte ich für das Ausleihen seiner Digitalkamera danken.
- Meiner Familie danke ich für ihre Unterstützung.

8 Eigenständigkeitserklärung

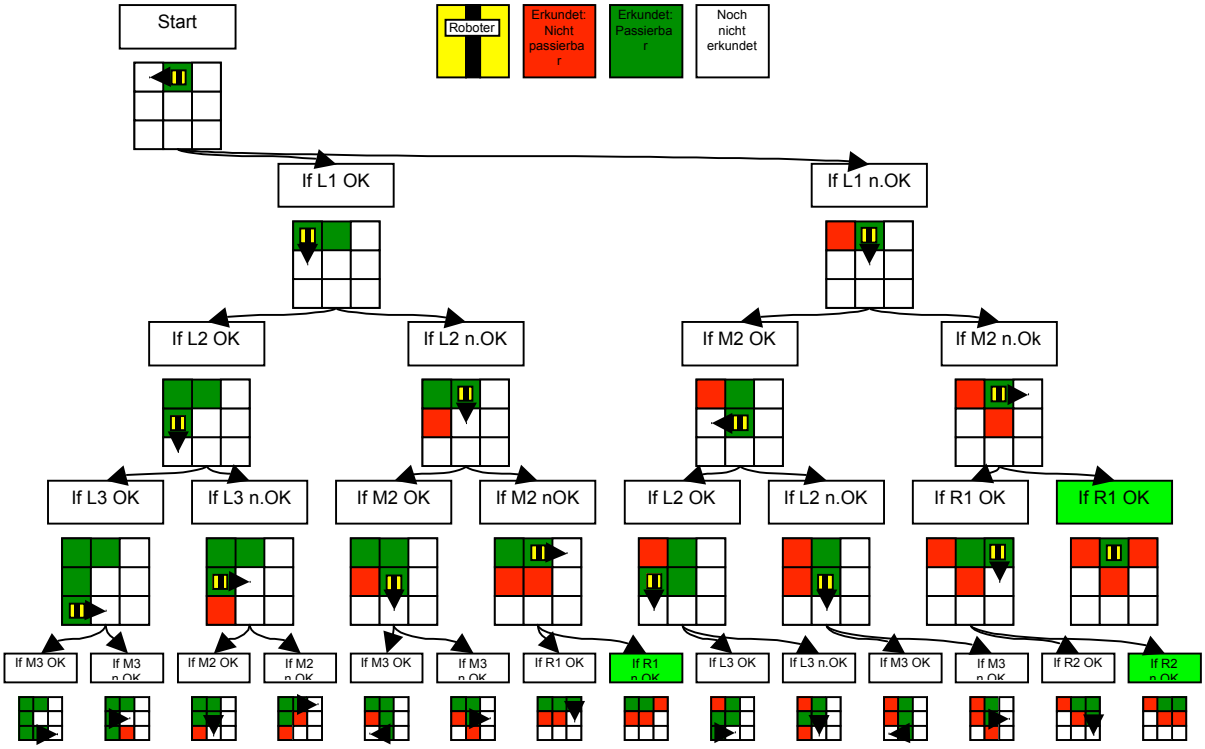
Ich bestätige hiermit, dass ich diese Arbeit eigenständig geschrieben habe.

Datum 1.11.2004 Unterschrift.....

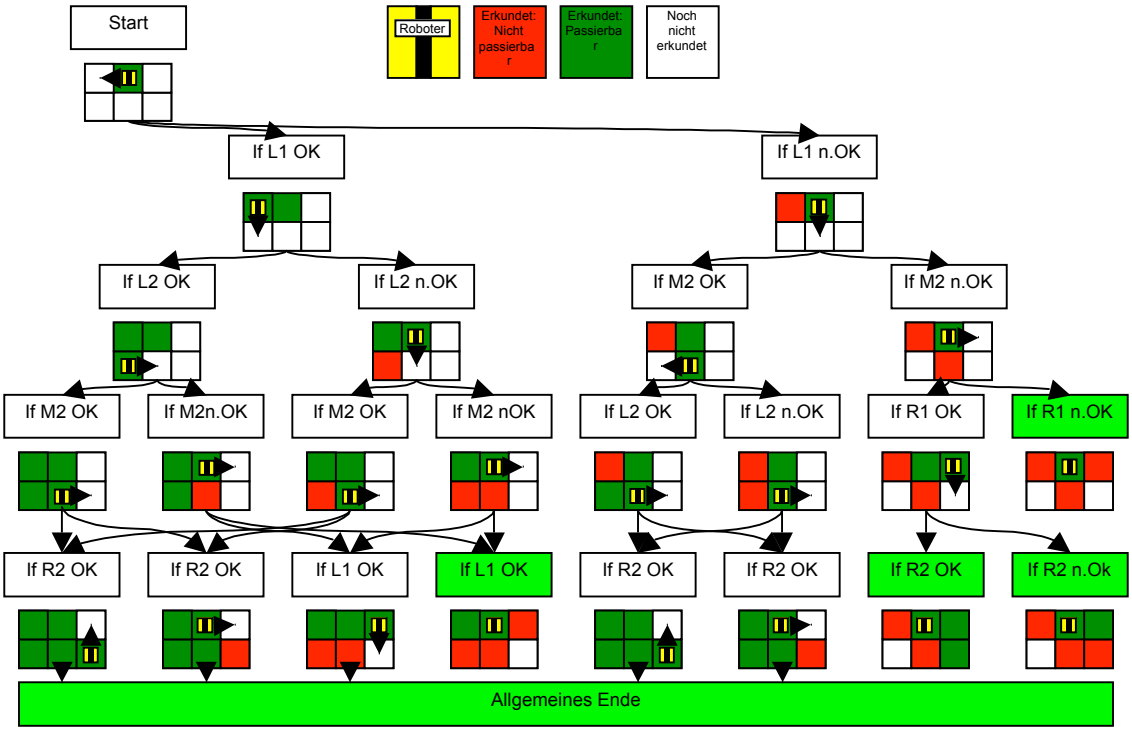
Anhang

Grafiken

Dies sind zwei Gedankenexperimente für ein Erkundungssystem für den Roboter.



Grafik 22



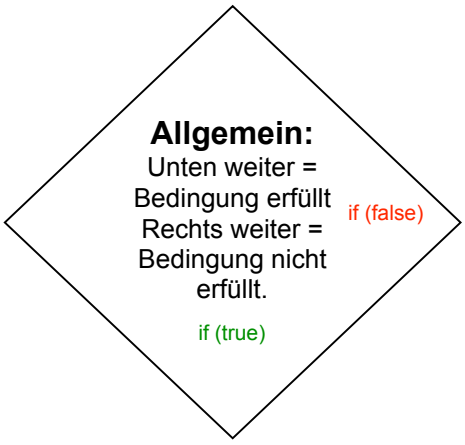
Grafik 23

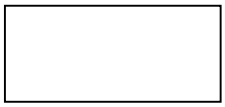
Kommunikationsprotokoll

Zahl	**Bedeutet**
0-99	Wert der Zahlen
100-149	Befehle für Master
150-199	Befehle für Slave
200-249	Befehle für PC
250-254	Reserve
110	Es folgt Koordinate
111	Es folgt Koordinate (Wert muss negativ sein)
120	Übermittlung starten
121	Übermittlung beenden
130	Übermittlung des RCX_S an PC erfolgreich
140	Objekt gefunden
141	Objekt untersucht
150	Führe examine aus
160	Führe transfer_data aus
200	Ping (Frage: „Bist du da?“)
240	Als nächstes folgt der Wert des Temperatursensors T11
241	Als nächstes folgt der Wert des Lichtsensors L3
255	OK, Befehl verstanden

Tabelle 21

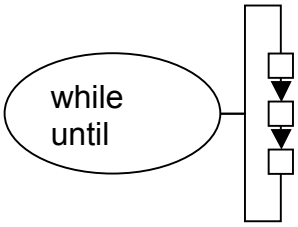
Zeichenerklärung zu den Grafiken:





Anweisungs
block

Nächster
Schritt



Während eine
Bedingung erfüllt,
oder bis eine
Bedingung erfüllt
ist, läuft eine
Schleife (kann
auch leer sein).

Grafik 24