# Eukalyptos Projektdokumentation


Praktikum am Institut für Wissenschaftliches Rechnen (IWR) der Universität Heidelberg

von Lutz Büch, Carlos Franke, Bastian Rieck

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

Dieses Kapitel soll einen Gesamtüberblick über das Projekt geben. Neben dem Praktikumsziel sind hier auch Grundsatzentscheidungen erläutert. Ebenso wird das von uns implementierte Konzept vorgestellt und beschrieben.

## 1.1 Namensgebung

Als Namen unseres Projektes wählten wir Eukalyptos, was einerseits auf Altgriechisch „Der Wohlverborgene“ bedeutet - ein schönes Motto für eine Fernsteuerung - andererseits Assoziationen zur Exotik ferner Länder und hochwertiger Kaugummis auslösen sollte. (daher auch der australische Koalabär im Programmlogo)

Weiterhin war auch unserem Testroboter ein Name zu geben. Hier wählten wir „Gingganz“, in Anlehung an das gleichnamige Gedicht von Christian Morgenstern, der beim Schreiben zwar höchstwahrscheinlich nicht an eine profane Fernsteuerung gedacht hat, dem wir mit unserer Wahl aber trotzdem eine Ehre zu erweisen hoffen.

> Ein Stiefel wandern und sein Knecht von Knickebühl gen Entenbrecht.
>
> Urplötzlich auf dem Felde drauß begehrt der Stiefel: Zich mich aus!
>
> Der Knecht drauf: Es ist nicht an dem; doch sagt mir, lieber Herre, – : wem?
>
> Dem Stiefel gibt es einen Ruck: Fürwahr, beim heiligen Nepomuk,
>
> ich GING GANZ in Gedanken hin . . . Du weißt, daß ich ein andrer bin,
>
> seitdem ich meinen Herrn verlor. . . Der Knecht wirft beide Arm' empor,
>
> als wollt' er sagen: Laß doch, laß! Und weiter zieht das Paar fürbaß.
>
> (Christian Morgenstern)

## 1.2 Praktikumsziel

Ziel unseres Projektes ist die Entwicklung eines Programmsystems zur Fernsteuerung eines „Lego Mindstorms“-RCX-Roboters. Es (das Projekt wie das System) wird sich im Wesentlichen in drei Komponenten gliedern:

- Die Benutzerschnittstelle
- Die Übersetzungseinheit
- Den Datenverkehr zwischen den Beiden.

Bei der Schnittstelle, die – wie der Name verrät – der Interaktion des Benutzers mit dem Programm dienen wird, ist Einfachheit der Bedienung das höchste Ziel. Ferner soll sie sowohl komplexe Aktionen – wie zum Beispiel das Vorwärtsfahren um eine bestimmte Strecke – direkt zur Verfügung stellen, als auch Zugriff auf Elementaroperationen des RCX bieten, was soviel heißt wie: Ansteuern der einzelnen Datenein- und -ausgänge. Zur Umsetzung dieser Überlegungen siehe Kapitel 3 über das GUI sowie Kapitel 2 über die Konfigurationsdatei.

Die zweitgenannte Komponente soll zuständig sein für die Übersetzung der Datenströme zwischen Schnittstelle und Roboter, und zwar in beide Richtungen. Sie ist in Kapitel 10 beschrieben.

Jedoch macht nicht die Steuerung alleine das Wesen einer Fernsteuerung aus, sondern eben auch die Ferne zwischen Steuerer und Gesteuertem: In der Praxis sollen die beiden eben beschriebenen Programmteile auf zwei verschiedenen, miteinander verbundenen Rechnern laufen. Und so liegt ein Hauptaugenmerk unserer Projektarbeit auf dem Datenaustausch zwischen ihnen, bei dem es einerseits auf größtmögliche Stabilität der Verbindung, andererseits auf die Minimierung der zu übertragenden Datenmengen ankommt. Mehr Informationen zu diesen Programmaspekten finden sich in Kapitel 8 und Kapitel 9.

Außerdem muss das System noch die Aufgabe übernehmen, den Benutzer über die Handlungen und die Umgebung des weit entfernten Roboters zu informieren. Besonders vielversprechend erscheint uns hierzu die Idee, den Roboter und seine Umgebung in einer virtuellen Realität nachzustellen. Dies wurde durch das Modellmodul umgesetzt. Siehe hierzu Kapitel 6.

Schließlich soll das System in seiner Gesamtheit sehr modular aufgebaut sein, sodass künftige Erweiterungen der einzelnen Komponenten denkbar sowie einfach durchzuführen sind. Siehe dazu Abschnitt 1.4.

## 1.3 Vorüberlegungen und Hindernisse

Es stand für uns fest schon zu Beginn fest, dass das Praktikum nach Möglchkeit mit Mitteln der OpenSource-Software realisiert werden sollte. Aufgrund der verwendeten Programme wie zum Beispiel NQC (siehe Kapitel 10) war die Entwicklungsplattform zunächst auf Linux beziehungsweise UNIX festgelegt.

Als kritischer Punkt erschien uns die Wahl eines Quelltextverwaltungssystems. Es sollte möglichst eine Versionskontrolle bieten, unter vielen Plattformen und Browsern funktionieren sowie einfach zu installieren sein. Neben den bekannten Lösungen wie CVS[1], SVN[2] und Trac[3] fanden wir ein weniger gebräuchliches, sehr mächtiges System des Fraunhofer Instituts: Das BSCW-System (Basic Support for Cooperative Work, `http://bscw.fit.fraunhofer.de`. Im Gegensatz zu den anderen Lösungen konnten wir hier bereits einen frei verfügbaren, öffentlich benutzbaren Server verwenden. Das BSCW-System bietet neben der einfachen Dokument- und Quelltextverwaltung die Möglichkeit, Objekte mit Notizen zu versehen und sie zu bewerten. Zudem können Internetadressen gespeichert und Dokumente direkt im Browser bearbeitet werden. Besonders interessant ist die Archivierungsoption: Hier verpackt das BSCW-System alle Dokumente eines Ordners in eine komprimierbare Datei, die dann zum Download angeboten wird. So kann man auf einen Computer schnell die neuesten Quelltexte aufspielen. Der einzige Wermutstropfen war die fehlende Quelltextverwaltung auf Betriebssystemebene. Der Benutzer muss seine Quelltexte somit immer selbst über den Browser aktualisieren und kann keine Syntax der Art `cvs commit` benutzen.

Es existiert eine kommerziell vertriebene Variante des BSCW-Systems. Diese ist unter `http://www.bscw.de/` genauer beschrieben. Laut Auskunft von „Orbiteam“ ist eine Lizenz für Universitäten kostenlos erhältlich.

Als problematisch erwies sich im Nachhinein das Durcheinander aus verschiedenen Zeichensätzen sowie Editoren zur Quelltextbearbeitung: Die Systeme im Robotiklabor speichern Texte beispielsweise mit dem Zeichensatz `UTF-8`, mit dem andere Distributionen noch Probleme haben. Späteren Praktikanten nach uns sei hier die Verwendung des Zeichensatzes `ISO-8859-15` ans Herz gelegt. Dieser ist auch unter Windows standardmäßig verfügbar und verursacht weniger Probleme.

## 1.4 Systemübersicht

Um ein möglichst modulares, erweiterbares Konzept zu verfolgen, verwendeten wir im Eukalyptos-Programm zwei Komponenten: Das Modul und den „Module Master“. Der „Module Master“ ist ein sogenannter Singleton. Das heißt, dass von der Klasse `module_master` nur genau eine Instanz pro Programm erzeugt werden kann. Auf diese Instanz kann dann global zugegriffen werden.

Der „Module Master“ vermittelt zwischen den einzelnen Modulen. Beim Programmstart[4] müssen zunächst im Quelltext alle zu verwendenden Module registriert werden. Die registrierten Module werden in einer Datenbank gespeichert. Diese kann eine beliebige Größe haben und ist nach oben hin nur durch den

[1] Concurrent Versions System, `http://www.nongnu.org/cvs/`
[2] Subversion, `http://subversion.tigris.org/`
[3] The Trac Project, `http://trac.edgewall.org/`
[4] Eine Registrierung kann auch später im Programm erfolgen. Da viele Module aber gleich zum Zeitpunkt ihrer Erzeugung ein anderes Modul benötigen, ist von diesem Verhalten abzuraten.

Hauptspeicher des Computers beschränkt[5].

Listing 1.1: Datenbankspeicherstruktur

```
1 struct module_info
2 {
3         char name[ MODULE_NAME_LENGTH + 1 ];
4         module* ptr;
5         int id;
6 };
```

Mit der Funktion `broadcast` können Nachrichten an alle Module geschickt werden. Diese Funktion wird in beinahe jedem Quelltext eingesetzt, um Fehlermeldungen, Warnungen und Informationen zwischen den einzelnen Modulen auszutauschen oder diese direkt dem Benutzer anzuzeigen.

Die Funktion `register_module` wird verwendet, um ein Modul in der Datenbank zu registrieren. Im Verlaufe der Registrierun wird das Modul durch `mod_load` auch geladen (siehe unten). Durch `get_module` kann ein Zeiger auf ein Modul angefordert werden. Als Parameter benötigt diese Funktion einen Teil des Modulnamens:

Listing 1.2: Benutzung des „Module Masters“

```
1 #include "module_net_tcp.h"
2
3 int main( void )
4 {
5         module_net_tcp mod_tcp;
6         module_net* mod_ptr;
7
8         module_master::ptr().register_module( &mod_tcp );
9         mod_ptr = dynamic_cast<module_net*>(
10        module_master::ptr().get_module( "module_net" ) );
11        module_master::ptr().broadcast( "Nachricht␣an␣alle␣
              Module" );
12
13        mod_ptr->connect( "localhost", 12345 );
14        return( 0 );
15 }
```

An diesem Beispiel lässt sich auch das Grundkonzept der Module sowie die Namenskonvention erläutern: Zunächst werden abstrakte Module mit einer Basisfunktionalität erstellt (beispielsweise `module_net`, siehe dazu auch Kapitel 8). Sie sollten einen Namen der Art „`abstract_module_`“ tragen (dieser lässt sich über die Funktion `get_name` setzen). Von diesen Modulen gibt es dann konkrete Ausprägungen, in diesem Fall `module_net_tcp`. Wie im obigen Listing zu erkennen ist, sucht `get_module` nach dem ersten Modul, das ein `module_net` im Namen trägt. Aus programmiertechnischer Seite kann es dann ganz normal über die Schnittstelle `module_net` bedient werden. Der **dynamic_cast** ist nötig, um den Zeiger, der auf ein Objekt vom Typ `module` zeigt, in einen Zeiger auf `module_net`

[5] In früheren Entwicklungsversionen hatte die Datenbank eine feste Größe. Dann wurde eine Version mit dynamischen Arrays erstellt. Da man hier aber sehr schnell Speicherlecks produziert, wird in der Release-Version von Eukalyptos die Datenbank durch die STL-Klasse `vector` verwaltet.

zu konvertieren, sodass die Funktionen der `module_net`-Schnittstelle verwendet werden können. Dazu muss die Vererbung unbedingt öffentlich sein (**public**).

Um eigene Module zu erstellen, muss von der Klasse `module` geerbt werden. Alle virtuellen, nicht implementierten Methoden sollten dann durch konkrete Implementierungen ersetzt werden.

Dieses Konzept hat uns den Austausch von Daten zwischen den unterschiedlichen Modulen extrem vereinfacht. So können zum Beispiel alle Module einen Zeiger auf das Konfigurationsmodul anfordern (siehe Abschnitt 2.1), um dann Daten aus der Konfigurationsdatei zu lesen:

Listing 1.3: Daten aus der Konfigurationsdatei lesen (aus module_map.cpp)

```
1 // Im Konstruktor von module_map:
2 mod_conf = dynamic_cast<module_conf*>( module_master::ptr().
    get_module(
3 "module_conf" ) );
4 if( mod_conf == NULL )
5 {
6         module_master::ptr().broadcast( "*module_map:␣Kein
7 ␣␣␣␣␣␣␣␣Konfigurationsmodul␣gefunden!␣␣Fatal!\n" );
8         return( -1 );
9 }
10
11 SENSOR_1_X = atof( mod_conf->da_datum( "SENSOR_1_X" ) );
```

# Kapitel 2

# Die Konfigurationsdatei und das Konfigurationsmodul

In diesem Kapitel geht es vor allem um jene Konstanten, die in der Konfigurationsdatei festgelegt werden. Einige wurden schon in anderen Kapiteln erklärt, aber hier soll noch einmal die Bedeutung jeder Konstante mit einigen Worten erläutert werden.

Eine Bemerkung soll vorausgeschickt werden. Für Angaben, die die Kenntnis des internen Koordinatensystems verlangen, gilt: Die „erste“ Dimension (auch „X“-Dimension) gibt die Breite von Karte und Roboter sowie den „Rechts-Teil“ relativer Positionen zur Robotermitte an. Die zweite gibt die Länge sowie den „Vorne-Teil“ an. Entsprechend für Richtungen.

## 2.1 Das Konfigurationsmodul

Um unsere Programme schnell und ohne neues Kompilieren modifizierbar zu machen, haben wir uns zu Anfang des Praktikums gleich nach einer Klasse umgesehen, die Konfigurationsdateien lesen kann. Wir sind im Internet auf eine solche gestoßen. Diese heißt „ConfigFile“ und ist frei verwendbar unter der „MIT Open Source Licence“. Mehr Informationen finden sich unter http://www-personal.umich.edu/~wagnerr/ConfigFile.html.

Ausgehend von dieser Klasse erstellten wir ein Konfigurationsmodul, das uns den Zugriff auf die Konfigurationsdatei ermöglichte. Diese Klasse ist als `module_conf_ConfigFile` in den Quelltext eingegangen. Ihre wichtigste Schnittstelle ist `da_datum`, denn mit dieser Funktion können Variablen aus der Konfigurationsdatei ausgelesen werden.

Listing 2.1: Die Benutzung des Konfigurationsmoduls

```
1 #include "module_conf_ConfigFile.h"
2
3 int main( void )
```

```
4  {
5          module_conf_ConfigFile mod_conf( "Gingganz.conf" );
6          module_conf* mod_conf_ptr;
7
8          module_master::ptr().register_module( &mod_conf );
9          mod_conf_ptr = dynamic_cast<module_conf*>(
10         module_master::ptr().get_module( "module_conf" ) );
11
12         mod_conf.da_datum( "ROBOTERBEFEHL" );    // Direkter
               Zugriff
13         mod_conf_ptr->da_datum( "EPM" );         // Indirekter
               Zugriff
14
15         return( 0 );
16 }
```

## 2.2 Kartenspezifikation

Die Umgebung, in der der Roboter sich bewegt und aufhält, wird im Modell durch eine rechteckige Karte dargestellt. Bei der aktuellen Implementierung sind deren Ausmaße fest und ändern sich auch dann nicht, wenn der Roboter über die Grenzen der Karte hinausfährt[1].

`EPM` „Einheiten pro Meter“ - die Auflösung der Karte. Sie sollte so gewählt sein, dass der schmalste Sensor des Roboter mehrere Einheiten breit ist.

`KARTENBREITE` Die Breite der Karte in Einheiten pro Meter (EPM). Ist z.B. EPM=200, so ist eine Karte von 4 Metern Breite mit 400 anzugeben.

`KARTENHOEHE` Die Höhe (oder Länge) der Karte in Einheiten pro Meter

## 2.3 Roboterspezifikation

Unsere Programme lassen eine genaue Steuerung und eine ebenso genaue Simulation der Bewegungen von Robotern zu, die der Nutzer erst nach der Entwicklungszeit baut. Dazu genügt es, einige Roboter-spezifische Werte durch Messungen und Messreihen zu ermitteln und in die Konfigurationsdatei einzutragen. Angaben über Sensoren sind exemplarisch für den 1. Sensor erklärt. Diese sind allerdings auch für die weiteren Sensoren einzutragen.

`ROBOTERNAME` Der Name des Roboters. Dient nur dazu, dass der Nutzer die Daten dem richtigen Roboter zuordnen kann.

`ROBOTER_LAENGE` Die Länge des Roboters in Einheiten pro Meter

`ROBOTER_BREITE` Die Breite des Roboters in Einheiten pro Meter

`ROBOTER_MITTE_X` Die erste Koordinate des Drehmittelpunktes des Roboters. Das ist der „Fixpunkt“ der Drehung des Roboters um seine eigene Achse,

[1] Dieses Verhalten ist für die jetzige Programmspezifikation durchaus erwünscht, denn hier liegt eine natürliche Spielfeldbegrenzung vor.

wenn er den Befehl `RECHTS` oder `LINKS` ausführt. Gemessen wird diese Koordinate von der linken Seite des Roboters in Einheiten pro Meter.

`ROBOTER_MITTE_Y` Die zweite Koordinate des Drehmittelpunktes des Roboters. Gemessen wird diese Koordinate von der unteren Seite des Roboters in Einheiten pro Meter.

`SENSOR_1_X` Die erste Koordinate der Position des Sensors, der am 1. Dateneingang angeschlossen ist, relativ zum Drehmittelpunkt des Roboters. Genauer: Anzugeben ist die Mitte des Sensors, die Koordinaten in Einheiten pro Meter.

`SENSOR_1_Y` Die zweite Koordinate der Position des 1. Sensors

`SENSOR_1_RTG_X` Die erste Koordinate der Richtung des 1. Sensors, relativ gemessen zu der Richtung des Roboters. Richtung des Sensor meint hier die Richtung, in die ein Berührungssensor eingedrückt wird. Der Richtungsvektor muss nicht normiert sein.

`SENSOR_1_RTG_Y` Die erste Koordinate der Richtung des 1. Sensors

`SENSOR_1_BREITE` Die Breite des 1. Sensors, gemessen in Einheiten pro Meter

`METERZEIT` Die Zeit in Hunderstel-Sekunden, die der Roboter benötigt, um 1 Meter zurückzulegen, in Abhängigkeit von der Batteriespannung in mV. Hierzu ist zunächst eine Messreihe durchzuführen, in der bei verschiedenen Batteriespannungen die gefahrenen Strecken bei fester Fahrdauer zu messen sind. Dabei ist es am leichtesten, sich einen Testbefehl zu schreiben, der den Roboter eine gewisse Zeit nach vorn fahren lässt. Die Strecke kann man direkt messen und die aktuelle Batteriespannung wird ja nach allen ausgeführten Befehlen angezeigt. Die Abhängigkeit der Größen kann man dann beispielsweise durch eine Ausgleichsgerade annähern. Hiermit haben wir im Test gute Erfahrungen gemacht. Es kann hier aber auch jegliche andere Formel in der Syntax der `Sim-Language` angegeben werden. Siehe dazu Kapitel 7.

`PERIODENDAUER` Die Zeit in Hunderstel-Sekunden, die der Roboter benötigt, um eine komplette Umdrehung (360° bzw. $2\pi$) zurückzulegen, in Abhängigkeit von der Batteriespannung in mV. Vorgehen zur Ermittlung einer Formel ähnlich wie bei `METERZEIT`.

`SIGMA_...` Alle Konstanten mit dem Präfix „`SIGMA_`“ sind wichtig für die Fehlerabschätzung. Sie geben Sim_Language-Formeln an, die im ersten Parameter von der Strecke bzw. vom Winkel, im zweiten Paramter von der Batteriespannung abhängen und die Abweichung (in Winkel bzw. Strecke) angeben. Diese Formeln sind wieder durch Messreihen zu ermitteln.

## 2.4 Roboterbefehle

Eine wichtiges Mittel zur bequemen Robotersteuerung ist das Implementieren von eigenen Befehlen. Damit ist nicht das Schicken von Quelltexten über `MASZGESCHNEIDERTER_BEFEHL` gemeint, sondern das feste Definieren von Befehlen in der Konfigurationsdatei.

Dazu gibt es zwei Wege. Der konsequentere und vielseitigere Weg ist es, direkt Quelltext anzugeben. Hier hat man keinerlei Einschränkungen. Man kann frei über alle Ein- und Ausgänge verfügen, Variablen definieren und so weiter. Allein muss man sich gewahr werden, dass man sich immer in einer Funktionsumgebung befindet, die Namen der Parameter sind vorgegeben und vom Datalog soll man auch die Finger lassen.

`BEFEHL` Um einen neuen Befehl zu implementieren, lege man einfach eine neue Konstante mit dem Namen des Befehls an. Der Wert, der zugewiesen wird, ist eine Zahl zwischen 1 und 250. Man muss darauf achten, dass keine Zahl doppelt definiert wird. Beispiel: `TANZ = 237`
Achtung: Außerdem muss in dem Roboter-Programm-Gerüst ein leeres `#define BEFEHL` angelegt werden.

`DIM_BEFEHL` Die „Dimension“ des Befehls. Dahinter verbirgt sich ganz einfach die Anzahl an Parametern, die der Befehl erwartet.

`NQC_BEFEHL` **und** `1_NQC_BEFEHL` Der Quelltext des Befehls. Wichtig ist eigentlich nur der Eintrag mit dem Präfix `1_NQC_`. Der andere ist nur ein Synonym.
Beim Verfassen des Quelltextes ist darauf zu achten, dass es nur derjenige Teil einer Funktion sein wird, der innerhalb der Funktionsumgebung steht. Das heißt die Deklaration der Funktion selbst muss und darf nicht hier erfolgen. Man schreibt hier einfach die Befehle nieder, die ausgeführt werden sollen. Die Parameter heißen übrigens `p_i`, wobei `i` für eine durchlaufende Nummerierung steht, die mit 0 beginnt.
Andere Befehle wie `VORWAERTS` oder auch eigene Befehle können mit dem Präfix „`DO_`“ aufgerufen werden, also zum Beispiel „`DO_VORWAERTS(90);`“
Zur Syntax: Generell ist nur die Syntax der verwendeten Lego-Programmiersprache (in dieser Implementierung NQC) zu beachten. Wichtig ist nur, dass man neue Zeilen im Quelltext mit „`~`“ beginnt. Außerdem müssen Gleichheitszeichen durch

```
@@~~~~$$aequumaequum$$^^^^@@
```

und #-Zeichen durch

```
@@~~~~$$cruxcrux$$^^^^@@
```

maskiert werden, um Konflikte mit der Syntax der aktuellen Konfigurations-Lese-Klasse zu umgehen. Ein Beispiel für einen syntaktisch richtigen Befehl:

```
1  NQC_VORWAERTS = @1_NQC_VORWAERTS
2  1_NQC_VORWAERTS = ~ SetDirection(OUT_A + OUT_C, OUT_FWD
       );
3  ~ SetPower(OUT_A + OUT_C, OUT_FULL);
4  ~ SetOutput(OUT_A + OUT_C, OUT_ON);
5  ~ Wait(p_0 * GEHFAKTOR / GENAUIGKEIT );
6  ~ SetOutput(OUT_A + OUT_C, OUT_OFF);
```

`CMD_BEFEHL` Diese Konstante gibt eine Beschreibung des Bewegungsablaufs innerhalb eines Befehls an. Siehe hierzu Kapitel 7.

### 2.4.1 Makros

Makros sind keine Befehle im eigentlich Sinne. Sie dienen nur einer Kurzschreibweise von einem Block von Befehlen, die bereits implementiert sind. Zu beachten ist, dass im Gegensatz zu einem richtig implementierten Befehl die Menge an Daten, die an den Roboter übertragen und von ihm behalten werden müssen, hier davon abhängt, welche und wieviele Befehle durch das Makro repräsentiert sind. Ein richtig implementierter Befehle mit eigenem Quelltext kann natürlich dasselbe und noch viel mehr als ein Makro leisten und kommt darüber hinaus auch mit weniger Datenvolumen in der Kommunikation aus. Dieses ist durch den RCX 2 relativ stark beschränkt. Daher sind Makros mit Vorsicht zu genießen.
Zur Definition eines Makros reicht es, Eine Dimension (Präfix „`DIM_`“), also die Parameteranzahl, und einen Eintrag mit Präfix „`MAKRO_`“ anzugeben. Dieser hat die gleiche Syntax wie die Einträge mit „`CMD_`“ für die Simulation, dürfen aber alle bereits implementierten Befehle benutzen, nicht nur die Elementarbefehle `VORWAERTS, RUECKWAERTS, RECHTS` und `LINKS`.

`MAKRO_BEFEHL` Neben der Dimension braucht man für ein Makro nur diesen Eintrag. Er hat die gleiche Syntax wie ein Ausdruck der Sim-Language (siehe Kapitel 7), allerdings darf man hier alle bereits implementierten Befehle aufrufen, bis auf andere Makros.

## 2.5 Konstanten für die Module

Einige Module benutzen Konstanten, die leicht über die Konfigurationsdatei zu ändern sein sollen.

`ROBOTERTYP` Hier stehen Kommandozeilen-Optionen für das Kompilieren des Roboterprogramms.

`HAUPTSPEICHERPLATZ` Die Nummer desjenigen Speicherplatzes im RCX 2.0-Blocks, auf den das eigentliche Roboterprogramm geladen wird

`FREISPEICHERPLATZ` Die Nummer desjenigen Speicherplatzes im RCX 2.0-Blocks, auf den das Programm geladen wird, das den Quelltext aus `MASZGESCHNEIDERTER_BEFEHL` enthält

`SENDER` Kommandozeilenoptionen für das Schicken einer Nachricht über „msg“. Damit wird bisher nur angegeben, ob ein USB-Turm verwendet wird, oder nicht. Benutzt man einen seriellen IR-Turm, trägt man hier nichts ein.

`PORT` Der Netzwerkport, der für die Kommunikation zwischen „Steuer“- und „Übersetzungsechner“ gedacht ist

`PFAD_LEBENSDAUER` Die Zeit, die ein zurückgelegter Pfad in der Karte des Modells sichtbar bleibt (in Sekunden)

`SENDEVERSUCHE` Die Anzahl an Versuchen, die der „übersetungsrechner“ unternehmen darf, dem Roboter den aktuellen Block an Befehlen zu übermitteln.

`MAX_ANZAHL_BEFEHLE` Die obere Grenze an Codes (d.h. Befehle und Parameter), der Roboter in einem Befehlsblock aufnehmen kann.

`GENAUIGKEIT` Da der RCX 2.0-Roboter nur mit Ganzzahlen rechnen kann, bedarf es eines Tricks, die Genauigkeit der Batteriespannungs-abhängigen Faktoren in den Elementarbefehlen `VORWAERTS, RUECKWAERTS, RECHTS` und `LINKS` zu erhöhen. Dazu wird zuerst der Faktor `GENAUIGKEIT` zu diesen Werten multipliziert und anschließend in den eigentlichen Befehlen wieder herausdividiert. Wichtig ist, dass man bei der Wahl dieses Wertes darauf achtet, dass man dadurch nicht die Roboter-interne Wertgrenze von vorzeichenlosen 16 Bit Integer, also etwa 65.000, nicht überschreitet.

## 2.6 Reservierte Codes

Im Folgenden werden einige Werte erklärt, die im Prinzip frei gewählt werden können. Das ist aber nicht nötig und man muss dabei auch einiges beachten. Bei den Codes

### 2.6.1 Spezielle Roboterbefehle

Dies sind Roboterbefehle, die nicht zu einer konkreten Funktion gehören, sondern die eine spezielle Bedeutung im Ablauf des Roboterprogramms haben. Da diese im Spektrum von 1-255 liegen müßen, die der Roboter empfangen kann, haben wir die oberen 5 Werte 251-255 hierfür reserviert. Damit schränken wir die übrigen Befehle und - viel wichtiger - Parameter am wenigsten ein. Denn um Fehlverhalten auszuschließen, dürfen auch Parameter diese Werte niemals annehmen. Das wird Programm-intern abgefangen.

`START_PHASE_1` Befiehlt dem Roboter, mit dem Lauschen auf Befehle zu beginnen.

`START_PHASE_2` Befiehlt dem Roboter, mit der Ausführung der Befehle zu beginnen.

`ABBRUCH` Beendet das Roboterprogramm brutal, lässt den Roboter jedoch angeschaltet.

`ROBOTERBEFEHL` Markiert den Anfang des Befehlsblocks, den man dem Roboter anschließend schickt.

### 2.6.2 Andere reservierte Codes

`MASZGESCHNEIDERTER_BEFEHL` Ein Wert, der anders als andere Befehle, nicht als Roboternachricht, sondern als Systemnachricht über das Netzwerk geht. Dabei handelt es sich um einen Befehl, mit dem man direkt Quelltext in der aktuell verwendeten Lego-Programmiersprache (in dieser Implementierung NQC) schicken und ausführen lassen kann.

`FATALER_FEHLER` Markiert Fehler im Datalog. In der Regel passiert dies nur bei einem Speicherüberlauf, nachdem man zu schnell Codes an den Roboter zu verschicken versucht hat. Dies wird durch das Programm aber verhindert.

`EMPFANGSLISTE_ANFANG` Markiert den Anfang der Empfangsliste. Eine solche wird im Datalog angelegt, sobald der Roboter erfolgreich den Empfang neuer Befehle erkannt hat.

`EMPFANGSLISTE_ENDE` Markiert das Ende der Empfangsliste.

`AUSFUEHRUNGSPROTOKOLL_ANFANG` Markiert im Datalog nach der Ausführung der aktuellen Befehle den Anfang der Ausführungsliste. Dient vor allem dazu, festzustellen, dass auch alles korrekt abgelaufen ist.

`AUSFUEHRUNGSPROTOKOLL_ENDE` Markiert das Ende der Ausführungsliste.

`BEFEHL_VERWEIGERT` Falls Befehle während der Ausführung geschickt werden, werden diese verweigert. Damit man das nachvollziehen kann, wird dies aber mit vorangestelltem `BEFEHL_VERWEIGERT` im Datalog festgehalten. Dieses Ereignis ist aber ohne Fremdeinwirkung unmöglich.

`UNGUELTIGER_BEFEHL` Falls ein Befehl nicht im Roboterprogramm implementiert wird. Das ist der Fall, wenn der Nutzer vergessen hat, im oberen Teil des Roboter-Programm-Gerüsts ein leeres `#define BEFEHL` anzulegen. Hierbei ist `BEFEHL` natürlich durch den Namen des zu implementierenden Befehls zu ersetzen.

`BATTERIESPANNUNG` Nach dieser Marke steht in der Ausführungsliste im Datalog die Batteriespannung in mV.

`SENSOR_1_AN` Markiert den Punkt in der Ausführungsliste, an dem der Roboter mit Sensor 1 angestoßen ist. Nach der aktuellen Implementierung bricht der Roboter nach diesem Ereignis sofort alle Befehle ab. Es folgt die Zeit in Hundertstelsekunden, die vom zuletzt begonnenen Befehl bis zum Abbruch schon ausgeführt wurde. Dieser Wert ist wichtig für die Simulation. Analog für Sensor 2 und 3.

`SENSOR_1_AUS` Markiert den Punkt in der Ausführungsliste, ab dem der Sensor keine Berührung mehr spürt.

# Kapitel 3

# Das GUI

Dieser Abschnitt beschreibt die grafische Benutzeroberfläche (GUI). Wir wollen hier nicht nur auf die Benutzung des GUI eingehen, sondern auch einen Überblick über die Verwendung von FLTK bieten.

## 3.1 Von der Wahl einer Grafikbibliothek

Die Wahl der Grafikbibliothek ist offensichtlich für das Projekt von großer Bedeutung. Wählt man eine unflexible, schwierig zu bedienende, so hat man letzendlich mehr mit dem Programmieren der GUI als mit den anderen Dingen zu kämpfen. Daher haben wir lange gesucht, bis wir uns endgültig für eine Bibliothek entschieden haben. Einige unserer Auswahlkriterien:

- Stabilität
- Einfache Bedienung
- Unterstützung mehrerer Plattformen
- Integrierbar in C++
- Funktionen der 2D-Grafik[1]

Im Verlaufe langwieriger Suchvorgänge schränkten wir die Bibliotheken auf zwei ein: `Qt` und `FLTK`. Diese seien hier kurz vorgestellt.

### 3.1.1 Qt

Qt ist ein sehr großes, ausgereiftes Toolkit zur Erstellung von GUIs. Es gibt diverse externe Anwendungen, wie zum Beispiel `QtDesigner`, die den Programmierer bei der schnellen Erstellung der GUI unterstützen. Zusätzlich bietet Qt Funktionen zur Netzwerkprogrammierung sowie diverse Module, die den Zugriff auf 2D- und sogar 3D-Funktionen ermöglichen.

[1] Als besonderer Bonus galt die zusätzliche Fähigkeit zur 3-dimensionalen Darstellung von Daten.

Leider ist der Code, der in Qt geschrieben werden muss, sehr umständlich und lang. Zudem ist der Installationsaufwand durch die vielen Hilfsapplikationen unverhältnismäßig hoch. Daher wurde diese Bibliothek verworfen.

### 3.1.2 FLTK

FLTK ist eine sehr kleine Bibliothek, die sich ganz der plattformunabhängigen Benutzeroberflächenerstellung verschrieben hat. Es gibt viele zusätzliche Module, um Zugriff auf 2D- sowie 3D-Grafiken zu haben. Ebenso werden die gängigeren Bildformate (JPEG, GIF, PNG) unterstützt. Durch die Verwendung der jeweilig nativen OpenGL-Treiber einer Plattform ist FLTK unter jedem Betriebssystem nahezu gleich performant. Gleichzeitig sind keine Code-Änderungen für den Kompiliervorgang unter einer anderen Plattform nötig.

FLTK verfügt über ein sehr frei ausgelegtes Fensterkonzept. Die Erstellung eines großen Hauptfensters, das den Bereich in ein 3D-Fenster und ein 2D-Fenster einteilt, ist somit möglich. Dies erlaubt uns maximale Flexibilität beim Einsatz von FLTK. Alles in allem erschien uns diese Grafikbibliothek als beste Wahl für unser Projekt.

In Abschnitt 3.5 ist FLTK genauer beschrieben.

## 3.2 Installation unter FreeBSD

Sofern die „FreeBSD Ports“, eine Sammlung von Kompilieranweisungen für viele Programme, installiert sind, ist die Installation sehr einfach. Als root-Benutzer führe man auf der Kommandozeile aus:

```
cd /usr/ports/x11-toolkits/fltk/
make install clean
```

Danach ist FLTK installiert und einsatzbereit.

## 3.3 Installation unter Gentoo Linux

Unter Gentoo Linux[2] ist die Installation noch unproblematischer. Auf der Kommandozeile genügt:

```
emerge fltk
```

## 3.4 Installation unter Windows mit Dev-C++

Dev-C++, eine bekannte freie Entwicklungsumgebung für Windows, kennt FLTK auch. Über das „WebUpdate“ beziehungsweise den „PackageManager“ kann FLTK sehr schnell eingebunden werden.

[2]Siehe http://www.gentoo.org/

## 3.5 Verwendung von FLTK

Die Grundklasse von FLTK ist das sogenannte „`Widget`“. Die meisten anderen Objekte stammen von einem Widget ab. Daher können viele Typen von Elementen im Code gleichartig behandelt werden. Beispiele hierfür sind alle Arten von Buttons oder Label. Jedem Widget kann mindestens ein „Callback“ zugewiesen werden. Dabei handelt es sich um eine Funktion, die bei Interaktion[3] des Benutzers mit dem Widget aufgerufen wird. Ausgehend von diesem Grundwissen können sehr schnell Benutzeroberflächen erstellt werden.

Im Folgenden seien einige Standardoperationen vorgestellt, die man in FLTK desöfteren durchführen muss.

### 3.5.1 Ein Beispielmakefile

Da die FLTK-Bibliotheken auf jedem System an anderen Stellen zu finden sind, gibt es das Programm `fltk-config`, das die entsprechenden Linker- und Compilerflags ausgibt. Unser Makefilegerüst zur Erstellung von FLTK-Programmen sieht wie folgt aus:

Listing 3.1: Grundgerüst eines Makefiles

```
1  CC=g++
2  CFLAGS=`fltk-config --cxxflags` -c
3  LDFLAGS=`fltk-config --ldflags --use-gl --use-images`
4  SOURCES= <Quelltexte>
5  OBJECTS=$(SOURCES:.cpp=.o)
6  EXECUTABLE= <Programmname>
7
8  all: $(SOURCES) $(EXECUTABLE)
9
10 $(EXECUTABLE): $(OBJECTS)
11         $(CC) $(LDFLAGS) $(OBJECTS) -o $@
12
13 .cpp.o:
14         $(CC) $(CFLAGS) $< -o $@
15
16 clean:
17         rm *.o *.core
```

Das Programm sollte dann durch durch die Eingabe von `make` auf der Kommandozeile kompilieren[4].

Es gibt noch weitere Schalter für `fltk-config`. Diese sind in der Hilfe zu `fltk-config` zu finden (`man fltk-config`).

### 3.5.2 Erstellung von Fenstern mit Steuerelementen

Um ein eigenes Fenster zu erstellen, kann man von bereits vorhandenen FLTK-Klassen erben. Für ein Standardfenster mit einigen Steuerelementen ist die Ba-

[3] Dies kann ein Klick sein oder eine Aktion wie das Eintippen von Text in ein Textfeld.

[4] Unter Windows funktioniert dies normalerweise nicht. Für manche Entwicklungsumgebungen wie beispielsweise Dev-C++ gibt es aber vorkonfigurierte Pakete, die das Erstellen von FLTK-Programmen ohne Makefiles erlauben.

sisklasse `Fl_Window` geeignet. Am einfachsten ist es, wenn alle Steuerelemente, die zum Fenster gehören, im Konstruktor der Fensterklasse erstellt werden.

Mit der Funktion `callback` kann man die Callbackfunktion eines Widgets setzen. Der erste Parameter ist das Widget, das den Callback verursacht hat. Er wird von FLTK automatisch gesetzt. Mit dem zweiten Parameter, einem **void**-Zeiger, kann man der Callbackfunktion Objekte übergeben. Da die Callbacks innerhalb einer Klasse allesamt statisch sein müssen, ist dies meist nötig, um den Zugriff auf nicht-statische Klassenvariablen zu ermöglichen.

`tooltip` setzt den Text, der erscheint, wenn man sich mit der Maus über das Steuerelement bewegt. Im Konstruktor jedes Widgets kann man wiederum (in dieser Reihenfolge) die x- und y-Position des Widgets in, dessen Breite und Höhe ( jeweils in Pixeln) sowie ein Label festlegen. Wo das Label angezeigt wird, hängt vom Widget ab. Bei einem Button ist es zum Beispiel die Beschriftung, bei einem Eingabefeld ein kurzer Hinweis, der über dem Feld angezeigt wird.

**Hinweis:** Die Codebeispiele bauen aufeinander auf. Sie entstammen unserem Code und sollten daher funktionsfähig sein.

Listing 3.2: Ein einfacher Callback, der nichts tut

```
1 void GUI_Window::cb_cmd_send_code( Fl_Widget* w, void* ptr )
2 {
3         // Auf Nachricht reagieren
4 }
```

Beim Aufruf des Konstruktors muss unbedingt beachtet werden, dass der Konstruktor der Basisklasse ebenso aufgerufen wird.

Listing 3.3: Ein Fenster mit einem Button (teilweise aus `gui.cpp`)

```
1 #include <FL/Fl.h>
2 #include <FL/Fl_Window.h>
3 #include <FL/Fl_Button.h>
4
5 class GUI_Window : public Fl_Window
6 {
7         public:
8                 GUI_Window(int width, int height, char* title)
9                 : Fl_Window( width, height, title )
10                {
11                        btn_send = new Fl_Button( 430, 38,
                              220, 26, "Quelltext␣senden" );
12                        btn_send->tooltip( "Ein␣Tooltip" );
13                        btn_send->callback( cb_cmd_send_code,
                              this );
14                };
15                virtual ~GUI_Window( void )
16                {
17                        delete btn_send;
18                };
19        private:
20                Fl_Button* btn_send;
```

```
21                  static void cb_cmd_send_code( FL_Widget* w,
                        void* ptr );
22  };
```

### 3.5.3 Erstellung von OpenGl-Fenstern

Wie eingangs erwähnt, ist FLTK zur Darstellung von OpenGL-Objekt fähig. In dem von uns erstellten GUI gibt es ein Unterfenster, das als Steuerelement im Hauptfenster vorhanden ist. Dieses zeigt beispielsweise eine Darstellung des Roboters. Jedwede Darstellung wurde auf eine ähnliche Art und Weise realisiert, wie sie hier beschrieben ist.

Listing 3.4: Ein OpenGL-Fenster, das ein weißes Dreieck enthält

```
1  #include <FL/Fl.h>
2  #include <FL/gl.h>
3  #include <FL/Fl_Gl_Window.h>
4
5  class GUI_GL_Window : public Fl_Gl_Window
6  {
7          public:
8                  GUI_GL_Window( int x, int y, int width, int
                       height, char* title )
9                  : Fl_Gl_Window( x, y, width, height, title )
10                 {
11                 };
12
13                 void GL_Window::draw( void )
14                 {
15                         glBegin( GL_TRIANGLES );
16                         glVertex3f( -1.0, 0.0, 0.0 );
17                         glVertex3f( 1.0, 0.0, 0.0 );
18                         glVertex3f( 0.0, 1.0, 0.0 );
19                         glEnd();
20                 }
21 };
```

### 3.5.4 Timer in FLTK

Das obige Listing hat einen Schwachpunkt: Das Dreieck bleibt statisch. Man kann es natürlich über OpenGL-Funktionen wie `glRotatef` rotieren lassen, doch diese Änderungen werden zunächst nicht angezeigt. Dies hängt damit zusammen, dass man FLTK mitteilen muss, wenn ein Neuzeichnen von Fenstern angebracht ist. Für diesen Zweck – und für viele andere[5] – gibt es in FLTK „Timeouts“.

Ihre Verwendung ist sehr einfach: Außer einer Funktion, in der die gewünschten Änderungen beschrieben sind, braucht man nur die Funktion `Fl::add_timeout`. Als ersten Parameter übergibt man ihr die Zeit, nach der die Funktion im zweiten Parameter ausgeführt werden soll. Der dritte Parameter ist ein optionaler

[5] Das GUI von Eukalyptos verwendet Timeouts beispielsweise, um in regelmäßigen Intervallen abzufragen, ob neue Nachrichten über das Netzwerk eingetroffen sind.

**void**-Zeiger. Über diesen Zeiger kann man den statischen Timeouts Objekte einer Klasse zur Verfügung stellen, auf die sie sonst keinen Zugriff hätten (wie bei den Callbackfunktionen auch).

**Hinweis:** Die Timeout-Funktionen sind nicht allzu genau. FLTK versucht, die angegebenen Zeiten einzuhalten, doch eine Garantie dafür, dass der Timeout tatsächlich genau nach der angegebenen Zeit aufgerufen wird, gibt es nicht. In der Praxis treten bei vernünftig gewählten, wenig CPU-lastigen Funktionen als Timeouts jedoch keine Probleme auf.

Listing 3.5: Eine Timeout-Funktion (aus `gui.cpp`)

```
1 void GUI_Window::cb_redraw_gl_window( void* ptr )
2 {
3         GUI_Window* window = reinterpret_cast<GUI_Window*>(
              ptr );
4         window->gl_win->redraw();
5
6         Fl::repeat_timeout( 1.0 / 25.0, cb_redraw_gl_window,
              ptr );
7 }
```

Durch Verwendung von `repeat_timeout` wird der Timeout nach der angegebenen Zeit wiederholt. Lässt man diesen Aufruf weg, so wird der Timeout nur einmal ausgeführt.

Listing 3.6: Einen Timeout hinzufügen (aus `gui.cpp`)

```
1 // Im Konstruktor von GUI_Window:
2 Fl::add_timeout( 1.0 / 25.0, cb_redraw_gl_window, this );
```

### 3.5.5 Menüs in FLTK

Beinahe jede grafische Benutzeroberfläche enthält Menüs. FLTK macht ihre Verwendung sehr einfach. Im Wesentlichen werden hier nur weitere Callbacks verwendet.

Der Konstruktor von `Fl_Menu_Bar` benötigt die Argumente x-, y-Positon, Breite und Höhe. In diesem Beispiel ist `width` die Breite des Fensters.

Mit der Struktur `Fl_Menu_Item` kann man den Aufbau des Menüs beschreiben. Der erste Parameter gibt den Namen des Menüpunktes an. Dabei werden etwaige „&“ als unterstrichende Schnellzugriffszeichen dargestellt. Im unteren Beispiel kann man mit `ALT+D` schnell auf das „Datei“-Menü zugreifen. Mit dem zweiten Parameter kann man einen weiteren Schnellzugriff definieren: `FL_CTRL+'q'` reagiert auf `STRG+Q`, `FL_F+5` auf Druck von `F5` usw. Der dritte Parameter gibt die bereits erwähnte Callbackfunktion an. Mit dem vierten, optionalen, Parameter kann man den Typ des Menüpunktes genauer definieren. `FL_SUBMENU` steht für ein Untermenü, das weitere Menüpunkte enthält. `FL_MENU_DIVIDER` fügt nach dem Menüpunkt eine Trennlinie ein.

Listing 3.7: Ein einfaches Menü mit dem oben erstellten Callback

```
1 // Zusaetzliche Includes:
```

```
#include <FL/Fl_Menu.h>
#include <FL/Fl_Menu_Bar.h>

// [...]
// Im Konstruktor von GUI_Window:
Fl_Menu_Item menu_items[] =
{
        { "&Datei", 0, NULL, NULL, FL_SUBMENU },
        { "T&est", 0, cb_cmd_send_code, this },
        { NULL }, // Ende des "Datei"-Menues

        { NULL }
};

// menu ist vom Typ Fl_Menu_Bar*
menu = new Fl_Menu_Bar( 0, 0, width, 30 );
menu->copy( menu_items );
```

### 3.5.6 Texteingabe

Das beste Widget zur Texteingabe ist `Fl_Text_Editor`. Gegenüber anderen Widgets wie zum Beispiel `Fl_Multiline_Input` hat es den Vorteil, dass man jederzeit Scrollleisten hinzufügen kann.

`Fl_Text_Editor` benötigt einen `Fl_Text_Buffer`. Diese Widget wird nur indirekt angezeigt und speichert den Text, den der Benutzer eingibt. Mit der Funktion `text` kann der aktuelle Text des Puffers gesetzt werden.

Listing 3.8: Erstellung eines Editorfensters mit Textbuffer (aus gui.cpp)

```
// Zusaetzliche Bibliothek:
#include <FL/Fl_Text_Editor.h>

// In der Klassendefinition zu GUI_Window:
Fl_Text_Editor* text_in;
Fl_Text_Buffer* text_in_buffer;

// Im Konstruktor von GUI_Window:
text_in = new Fl_Text_Editor( 10, 365, 265, 270 );
text_in_buffer = new Fl_Text_Buffer;
text_in->buffer( text_in_buffer );
text_in->buffer->text( "Hallo, Welt!" );
```

### 3.5.7 Textausgabe

Für die Textausgabe ohne Änderungsmöglichkeit seitens des Benutzer gibt es das Widget `Fl_Text_Display`. Es benötigt ebenso einen `Fl_Text_Buffer`, doch sonst unterscheidet sich die Verwendung nicht vom oben vorgestellten `Fl_Text_Editor`.

Listing 3.9: Erstellung eines Ausgabefensters mit Textbuffer (aus gui.cpp)

```
// Zusaetzliche Bibliothek:
#include <FL/Fl_Text_Display.h>

```

```
// In der Klassendefinition zu GUI_Window:
Fl_Text_Display* text_out;
Fl_Text_Buffer* text_out_buffer;

// Im Konstruktor von GUI_Window:
text_out = new Fl_Text_Display( 285, 365, 380, 270 );
text_out_buffer = new Fl_Text_Buffer;
text_out->buffer( text_out_buffer );
text_out_buffer->text( "Dies wird angezeigt." );
```

Eine Leiste zum Scrollen des Textes wird automatisch hinzugefügt. Wenn das Ausgabefenster automatisch zur letzten hinzugefügten Zeile springen soll, verwendet man die Funktion `scroll`. Sie erwartet als ersten Parameter die Zeile, zu der gescrollt werden soll. Der zweite Parameter gibt an, an welches Zeichen dieser Zeile er Cursor gesetzt werden soll. Der Aufruf von `count_lines` zählt alle Zeilen von Anfang bis Ende.

Listing 3.10: Zur letzen Zeile des Ausgabefensters scrollen

```
text_out->scroll( text_out_buffer->count_lines( 0,
text_out_buffer->length() ), 0 );
```

### 3.5.8 Fenster anzeigen

Nachdem nun das Fenster mit Steuerelementen und Callbacks versehen wurde, ist es Zeit, dieses auch anzuzeigen. Die Funktion `Fl::run()` startet eine Schleife, die erst durch das Schließen des Programmfensters beendet wird. Jegliche „Programmlogik“ findet dann durch Callbacks oder Timeouts der verschiedenen Fenster statt[6].

Listing 3.11: FLTK-Grundgerüst

```
// Entsprechende Bibliotheken einfuegen

int main( void )
{
        GUI_Window GUI( 800, 600, "Fenstertitel" );
        GUI.show();
        Fl::run();
        return( 0 );
}
```

### 3.5.9 Meldungsfenster und Eingabedialoge

Um den Benutzer auf Informationen aufmerksam zu machen, kennt FLTK die Funktion `fl_message` und `fl_alert`. In beiden Fällen kann man diesen Funktionen einen C-String ( `const char*` ) übergeben. In diesem String können die von `printf`[7] bekannten Bezeichner eingefügt und genutzt werden.

[6] Am Konstruktor des GUI-Fensters wird deutlich, dass das FLTK-Konzept konsistent ist: Wie jedes Widget gibt man im Konstruktor die Breite, Höhe sowie einen Bezeichner an.

[7] Siehe hierzu auch `man printf` auf der Konsole.

Listing 3.12: Informationsnachricht, die einen Wert anzeigt

```
1 #include <FL/fl_ask.h>
2
3 int main( void )
4 {
5         int answer = 42;
6         fl_message( "Die␣Antwort␣auf␣alles␣ist...%i.\n",
              answer );
7         return( 0 );
8 }
```

Um den Nutzer Werte eingeben zu lassen, gibt es `fl_input`. Sie hat als ersten Parameter einen Bezeichner für das Eingabefeld, beispielsweise „Bitte Dateiname eingeben“. Der zweite Bezeichner ist der Standardwert, der im Eingabefeld zu finden ist, zum Beispiel „paths.dat“. Weitere Argumente beziehen sich auf `printf`-Bezeichner die im Bezeichner des Eingabefeldes vorkommen. `fl_input` gibt einen C-String ( **const char*** ) zurück. Diese ist nur gültig bis zum nächsten Aufruf der Funktion.

Listing 3.13: Lässt den Nutzer eine Eingabe tätigen

```
1 #include <FL/fl_ask.h>
2
3 int main( void )
4 {
5         int number = 21;
6         fl_input( "Was␣ist␣2*%i?\n", "Bitte␣hier␣Antwort␣
              eingeben",
7         number );
8         return( 0 );
9 }
```

### 3.5.10 Weiterführende Informationen

Eine kleine Sammlung von Webseiten, die beim Umgang mit FLTK unterstützen:

- Die FLTK-Dokumentation unter http://www.fltk.org/doc-1.1/toc.html.
- FLTK-Beispielprogramme im Quelltext unter http://svn.easysw.com/public/fltk/fltk/trunk/test[8]
- „Erco's FLTK Cheat Page“ unter http://seriss.com/people/erco/fltk bietet viele Beispiele zu häufigen Aufgaben.
- Bei „Neon Helium Productions“ unter http://nehe.gamedev.net gibt es interessante Beispiele und Erklärungen zu beinahe allen OpenGL-Features.

[8] Bei der Installation von FLTK werden die entsprechenden Beispielquelltexte eventuell schon automatisch mitinstalliert.

## 3.6 Die Benutzeroberfläche

Das GUI des Projekts ist möglichst funktional gehalten. Es gibt ein Eingabefeld, in das der Nutzer seine Befehle eingeben kann. Die Syntax der Befehle ist in Kapitel 7 beschrieben. Direkt neben der Befehlseingabe findet sich das Ausgabefenster. Hier werden die Nachrichten (Warnungen, Fehler, Informationen) der einzelnen Module angezeigt. Weitere Informationen finden sich in Kapitel 5.

Weitere Ausgaben sind die aktuelle Batteriespannung des Roboters sowie die geschätzten Fehler in der Position des Roboters. Das Batteriespannungsfeld zeigt die Spannung in drei unterschiedlichen Farbgebungen an: Grün steht für eine sehr gute bis gute Spannung (ab 8.0V aufwärts), Gelb für eine mittelmäßige Spannung (7.6V - 8.0V), rot für eine sehr niedrige Spannung (ab 7.6V abwärts). Diese Werte haben sich in der Praxis als sinnvoll für die Beurteilung der Genauigkeit herausgestellt. Siehe dazu auch Abschnitt 6.2.
Die vorhandenen Buttons machen die am häufigsten verwendeten Funktionen schnell zugänglich. Sie sind ebenso im Menü verfügbar. Für die Bedienung mit der Tastatur haben wir einige Shortcuts vorgesehen:

`STRG+O` Öffnet den Dialog zum Herstellen einer Verbindung.

`STRG+Q` Beendet das Programm.

`F1` Schaltet den Roboter sofort aus. Dieser „Not-Aus-Funktion“ wurde scherzhaft die Taste zugewiesen, die in den meisten Programmen für „Hilfe“ steht.

`F5` Überträgt den vom Benutzer eingegebenen Quelltext zum Roboter, der ihn dann ausführt.

`F6` Öffnet den Dialog zum Speichern der Karte. Damit werden alle gefahrenen Pfade des Roboters in Wertepaaren der Art (Startpunkt x-Position, Startpunkt y-Position), (Endpunkt x-Position, Startpunkt y-Position) gespeichert. Sie können zur Auswertung mit einem Programm wie `GNUPlot` angezeigt werden. Siehe auch Kapitel 6.

Das größte Element im GUI ist die Übersichtskarte. Sie stellt die folgenden Objekte dar:

- Den Roboter. Der Teil von ihm, der nach vorne zeigt, wird rot eingefärbt. Der Teil von ihm, der nach hinten zeigt, blau.
- Die gefahrenen Pfade des Roboters. Hier gibt es zwei Unterscheidungen: Zum einen werden grüne Pfade gezeichnet. Sie sind permanent und zeigen dem Benutzer, welchen Teil der Karte der Roboter bereits gesehen hat. Zum anderen werden rote Pfade gezeichnet. Diese verbinden nach einer Bewegung des Roboters seine aktuelle Position mit der ursprünglichen Position, damit der Benutzer die Bewegung besser nachvollziehen kann. Rote Pfade werden nach der Zeit, die in der Konfigurationsdatei als `PFAD_LEBENSDAUER` angegebenen ist, sanft ausgeblendet (siehe dazu auch Kapitel 2).

- Die Hindernisse. Sie werden als kleine, weiße Linien dargestellt. Ein zur Linie senkrechter Strich deutet an, in welcher Richtung der Roboter mit dem Objekt kollidiert ist. Zur genauen Angabe der Sensorposition siehe Kapitel 2.

# Kapitel 4

# Der Parser

Dieses Kapitel erklärt den Befehlsparser, der den Übergang von der Nutzereingabe hin zur Weiterverarbeitung und -leitung durch die Programme schafft.

## 4.1 `module_parser`

`module_parser` ist das Modul, das die Nutzereingaben in Netzwerknachrichten verwandelt, die die Befehle enthalten. Diese können dann über das Netzwerk an den Server, und von dort aus an den Roboter geschickt werden.
Zunächst ersetzt es alle vorkommenden Befehlsmakros durch die Befehle, die dieses vereint. Siehe hierzu auch Kapitel 7 über die Simulationssprache. Ist dies geschehen, enthält der Eingabetext nur noch Elementarbefehle, deren Befehlscodes und sonstige Eigenschaften in der Konfig-Datei gespeichert sind. Damit erstellt der Parser für jedes Vorkommen eines Befehls eine Netzwerknachricht. Dafür steht die Klasse `saved_net_msg` bereit. Siehe hierzu auch Abschnitt 8.3 über das Netzwerkprotokoll. Stimmt die Parameteranzahl nicht mit der in der Nutzereingabe überein, gibt `parse` einen leeren Vektor zurück.

### 4.1.1 Die Syntax

Die Syntax ist sehr einfach und intuitiv. Jeder Befehl wird mit einer direkt nachfolgenden Klammer aufgerufen. Diese Klammer enthält die Parameter, die durch Kommata getrennt werden. Befehle werden nur durch „Whitespaces“, also durch Leerzeichen, Zeilenumbrüche oder Tabulatoren getrennt.
In der Konfig-Datei kann man nachlesen, welche Befehle verfügbar sind. Sie sind eindeutig daran erkennbar, dass für jeden auch ein Eintrag mit dem Präfix „`DIM_`“ vorhanden ist. Dieser Eintrag gibt die Anzahl an erwarteten Parametern für diesen Befehl an. Diese muss beachtet werden; es gibt keine „überladenen“ Befehle. Groß- und Kleinschreibweisen des Befehlsnamen und Mischformen daraus sind äquivalent. Befehlsbezeichner dürfen aus allen druckbaren Zeichen außer einer öffnenden Klammer bestehen. Ein Beispiel für eine syntaktisch richtige Eingabe: `VORWAERTS(50) ReChTs(90) rueckwaerts(45)`
`links(39)`

# Kapitel 5

# Das Log-Modul

Hier wird das Log-Modul erklärt, das Fehlermeldungen und Warnungen speichert und für die Ausgabe durch die GUI bereithält.

## 5.1 `module_log`

Der Nachrichtenverkehr zwischen den Modulen wird zentralisiert über den Modul-Master abgewickelt. Ein Modul schickt eine Nachricht auf den Weg, indem es die Funktion `broadcast` des Modul-Masters aufruft. Dieser leitet sie an alle Module, die bei ihm registriert sind, weiter. Jedes Modul soll dann geeignet darauf reagieren. In der Regel folgt überhaupt keine Reaktion. Doch das Modul `module_log` reagiert auf jede Meldung. Es speichert alle eingehenden Nachrichten in einer Log-Datei. Ihr Dateiname ist als Konstante fest im Quelltext von `module_log` definiert.
Andererseits hat das Modul einen Zwischenspeicher („Cache“), der eine gewisse Anzahl an eingegangenen Nachrichten aufnehmen kann. Diese Zahl ist ebenfalls fest definiert. Aus dem Cache liest die GUI nun im Sekundentakt alle Nachrichten aus. Ist dies geschehen, verwirft das Log-Modul diese.

### 5.1.1 Format

Im Zwischenspeicher von `module_log` werden die Nachrichten so, wie sie sind, abgespeichert. In der Log-Datei wird jede Nachricht in folgendem Format abgespeichert:

```
<Zeitstempel> <Typ> <Inhalt>
```

**Zeitstempel** ist die Zeit bei Eingang der Nachricht im Format

```
<Tag>.<Monat>.<Jahr> <Stunde>:<Minute>:<Sekunde>
```

**Typ** kann durch ein der Nachricht vorangestelltes „*“, „#“ oder „!“ als „Fehler:“, „Warnung:“ oder „Information:“ festgelegt werden. Fehlt dieser Präfix, wird von einer Information ausgegangen.

**Inhalt** ist die eigentliche Nachricht abzüglich des Präfix.

Beispiel:
`*module_parser: Anzahl der Parameter ist nicht korrekt.`
wird zu

```
[31.12.2006 23:59:59] Fehler: module_parser: Anzahl der Parameter ist nicht korrekt.
```

Zu Beginn eines Programmlaufs setzt das Log-Modul selbst die Nachricht

```
!-------- ANFANG DER SITZUNG --------
```

ab, sodaß verschiedene Sitzungen im Log sauber auseinandergehalten werden können. Wird das Programm normal beendet, folgt

```
!--------- ENDE DER SITZUNG ---------
```

### 5.1.2 Meldungen im Server

Da das Log-Modul nur eine Instanz auf dem Programm des Steuerrechners hat, werden alle Nachrichten, die auf dem Übersetzungsrechner (auch: “Server“) anfallen, über das Netzwerk an den Klienten weitergeleitet. Dabei wird wieder ein Vorteil des modularen Aufbau der Programme offenbar. Die Instanz des Netzwerkmoduls bekommt wie alle Module alle Nachrichten des Servers. Durch die Variable „is_server“ ist festgelegt, dass es die Instanz auf der Serverseite ist. So kann es dafür sorgen, dass die Nachrichten über das Netzwerk an den Klienten weitergeleitet werden. Die GUI, die die Netzwerknachrichten entgegennimmt, erkennt anhand des Nachrichtenkopfes (siehe Abschnitt 8.3), dass es sich um eine Nachricht handelt, die das Log-Modul speichern soll. Da die übrigen Module des Klientenprogramms allerdings nicht auf die Nachrichten des Serverprogramms reagieren sollen, wird direkt `module\_log::handle\_msg()` aufgerufen.

# Kapitel 6

# Das Modell

Dieses Kapitel beschäftigt sich mit den Funktionen des Kartenmodules (`MODULE_MAP`). Es beschreibt außerdem die Interna des von uns erstellten Modells zur Positionsbestimmung des Roboters.

## 6.1 Auswertung von Messdaten

Um unsere Messdaten auszuwerten, setzten wir verschiedene frei verfügbare Programme ein:

- Zur linearen Regression erwies sich ein Applet der Universität Innsbruck als sehr hilfreich. Es findet sich unter http://www.mathe-online.at/nml/materialien/innsbruck/regression/. Nach der Eingabe der Messdaten (siehe unten), kann man sich detaillierte Statistiken erstellen lassen sowie eine Ausgleichsgerade plotten.
- Zur schnellen Analyse von Messdaten auf der Kommandozeile ist das Programmpaket `|STAT` von Gary Perlman (http://www.acm.org/~perlman/stat/ geeignet.

Eine beispielhafte Auswertung der Messdaten könnte so aussehen: In der Datei `daten-vorwaerts.txt` sind, durch ein Leerzeichen getrennt, in jeder Zeile Werte für die Spannung in mV sowie die gefahrene Strecke in cm zu finden. Um mit `|STAT` eine lineare Regression durchzuführen, genügt der Aufruf

```
dm s2 s1 < daten-vorwaerts.txt | regress
```

Auf der Kommandozeile. Dies weist `|STAT` an, die Datenspalten zu vertauschen[1]. Diese Ausgabe dieses Befehls wird dann an das Programm `regress` weitergeleitet, das zu `|STAT` gehört und eine lineare Regression durchführt. Auf diese Weise kann man mit wenigen Befehlen sehr komplexe Auswertungen durchführen. Daher ist ein Blick in die `man`-Seite von `|STAT` empfohlen.

[1] Eine Regression wird immer nach der Variablen durchgeführt, die in der zweiten Spalte steht. In unserem Fall wollten wir aber die Strecke in Abhängigkeit der Batteriespannung berechnen.

## 6.2 Zusammenhang von Spannung und Geschwindigkeit

Um zu üeberprüfen, wie die Spannung mit der Geschwindigkeit des Roboters zusammenhängt, haben wir eine Messreihe durchgeführt. Wir maßen hier, wie weit der Roboter kommt, wenn beide Motoren mit voller Kraft[2] betrieben wurden. Die Laufzeit jedes Versuchs betrug 8s.

Aus einer einfachen Überlegung heraus vermuteten wir einen linearen Zusammenhang zwischen Spannung und zurückgelegter Strecke: Stellt man sich die Batterien des Roboters als ideale Stromquelle vor und das elektrische Verhalten des Roboters wie das eines ohm'schen Widerstandes, ist die erbrachte Leistung $P$ gleich Strom mal Spannung: $P = I \cdot U$ oder auch, wenn man den Strom durch Spannung und den konstanten Widerstand ausdrückt: $P = \frac{U^2}{R}$ Leistung aber ist freigesetzte Energie pro Zeit; und an dieser Energie hat, so nehmen wir weiter an, die kinetische Energie E des Roboters einen proportionalen Anteil (der Einfachheit halber 1). Mit der bekannten Formel für die Bewegungsenergie erhält man (hierbei ist $v$ die Geschwindigkeit, $m$ die Robotermasse und $t$ die Zeit):

$$\frac{U^2}{R} = \frac{\frac{1}{2} \cdot m \cdot v^2}{t}$$

Aus der Gleichung liest man leicht ab, dass Spannung und Geschwindigkeit in diesem Modell proportional sind. Die Daten scheinen dies auch zu bestätigen.

Siehe dazu Abbildung 6.2 und Abbildung 6.2.

## 6.3 Zusammenhang von Spannung und Drehwinkel

Es lag aus den nämlichen Gründen nahe, wiederum einen linearen Zusammenhang zu vermuten. Wir maßen nun den Winkel, den der Roboter durch eine 8-sekündige Drehung einnimmt. Der lineare Zusammenhang ergab sich bei der Auswertung. Siehe dazu auch Abbildung 6.3 und Abbildung 6.3.

## 6.4 Umsetzung im Programm

Dieser Abschnitt beschäftigt sich mit der Arbeit hinter den Kulissen, die nötig ist, wenn der Roboter seine Position im Modell ändern soll.

### 6.4.1 Datenspeicherung

Am Anfang war es geplant, die Karte in ein Raster aus Kartenelementen zu unterteilen. Jedes Element sollte eine bestimmte Dimension haben, beispielsweise 0.5cm auf 0.5cm. Für jedes einzelne Element könnte man dann den Typ

[2] `OUT_FULL`, siehe auch „The NQC Programmer's Guide“, http://bricxcc.sourceforge.net/nqc/doc/NQC_Guide.pdf.

festlegen, Winkel speichern und vieles mehr. Aufgrund des sehr großen Speicheraufwandes (bei der oben erwähnten Genauigkeit umfasst eine Karte von 200cm auf 200cm bereits 160000 Objekte) sowie des enorm großen Rechenaufwands verwarfen wir diese Idee in der Praxis wieder. Stattdessen wird nun ein Vektor aus „interessanten“ Kartenelementen gebildet. In ihm werden nur noch die relevanten Stücke der Karte abgespeichert – in unserem Fall also nur Hindernisse. Der größte Teil der Karte, die Freifelder, werden nirgendwo explizit gespeichert. Das GUI verwendet dann die Hindernisdaten, um die Karte grafisch darzustellen.

Um dem Benutzer dennoch einen Überblick über die bereits besuchten Gebiete der Karte zu geben, entschieden wir uns dazu, Pfade einzuführen. Diese Objekte haben nur einen Anfangs- und einen Endpunkt sowie eine zuvor spezifizierte Zeichenbreite, die jedoch nur das GUI betrifft. Sie sind somit extrem komprimiert speicherbar und ermöglichen es uns, bereits befahrene Flächen oder Bewegungen des Roboters auf der Karte entsprechend einzufärben.

Insgesamt erscheint uns die aktuelle Lösung zur Datenspeicherung selbst bei sehr großen Karten mit vielen Kartenelementen flexibler und schneller als der ursprünglich geplante Ansatz.

### 6.4.2 Simulation von Bewegungen

Für die Simulation von Roboterbewegungen und die Speicherung von Objektddaten ist `module_map` zuständig. Sobald der Benutzer einen Quelltext eingegeben hat, um ihn an den Roboter zu senden, wird zunächst über `estimate_execution_time` die Ausführungszeit der Kommandos abgeschätzt. Danach werden die Kommandos zur späteren Verarbeitung zwischengespeichert. Die Gesamtheit der Kommandos mit Parametern wird von nun an Befehlsblock[3] genannt.

Die Abarbeitung vom Roboter erfolgt, wenn alle Befehle syntaktisch korrekt waren. Während der Abarbeitung speichert der Roboter den aktuellen Befehl in seinem Datalog. Dieses wird am Ende des Befehlsblocks ausgelesen und an das GUI zur weiteren Verarbeitung geschickt. Hier wird durch die Funktion `parse_datalog` des Kartenmoduls die Überprüfung des Datalogs gestartet. Dazu werden die zuvor gespeicherten Befehle mit den ausgeführten Befehlen verglichen.

Für den Fall, dass keinerlei Diskrepanzen auftreten, wird die Funktion `execute_command` aufgerufen. Sie zerlegt den Befehl zunächst in Elementarbefehle. Das heißt, dass ein Tanz, der beispielsweise aus Vorwärtsbewegungen und Drehungen besteht, in Einzelteile zerlegt wird. Das Kartenmodul führt dann sequentiell die Bewegungen aus, beispielsweise „Fahre 4 Einheiten vorwärts, drehe dich im rechten Winkel nach links, fahre 10 Einheiten vorwärts“. Siehe hierzu auch den das Kapitel 7 über die Simulationssprache. Jede erfolgreich ausgeführte Bewegung erstellt automatisch Pfade. Dies sind Verbindungslinien zwischen zwei Roboterpositionen, um die Positionsänderungen des Roboters bei komplexeren Bewe-

[3] Er darf eine gewissen Größe nicht überschreiten. Siehe dazu auch Kapitel 2 über die Konfigurationsdatei.

gungen leichter nachvollziehen zu können.

Der weitaus interessantere Fall ist der, bei dem nicht alle Kommandos ausgeführt wurden. Momentan reagiert unser Programm nur auf Sensormeldungen. Falls der Roboter also mit etwas kollidiert ist, kann die `execute_command` diese erkennen. Im Datalog steht die Zeit, die seit der Ausführung des Befehls vergangen ist. Anhand dieser Daten rechnet die Simulation dann aus, welche Elementarbefehle („Vorwärts“, „Links“ etc., siehe oben) zu welchem Teil ausgeführt werden konnten: Beispielsweise könnte der Nutzer geplant haben, 42cm zu fahren. Diese Aktion nimmt 10s in Anspruch. Nach 2.3s ist der Roboter jedoch mit einem Hindernis kollidiert. `execute_command` errechnet nun, wie weit der Roboter im Kartenmodell gekommen ist und stellt seine Position ebenso wie im fehlerfreien Fall dar. Zusätzlich wird an die entsprechende Stelle ein Hindernis eingefügt.

**Nebenbemerkung:** Grundsätzlich wird zwischen den verschiedenen Sensoren unterschieden. Das GUI zeichnet die Hindernisse daher so, dass man erkennen kann, mit welchem Sensor eine Kollision statt fand.

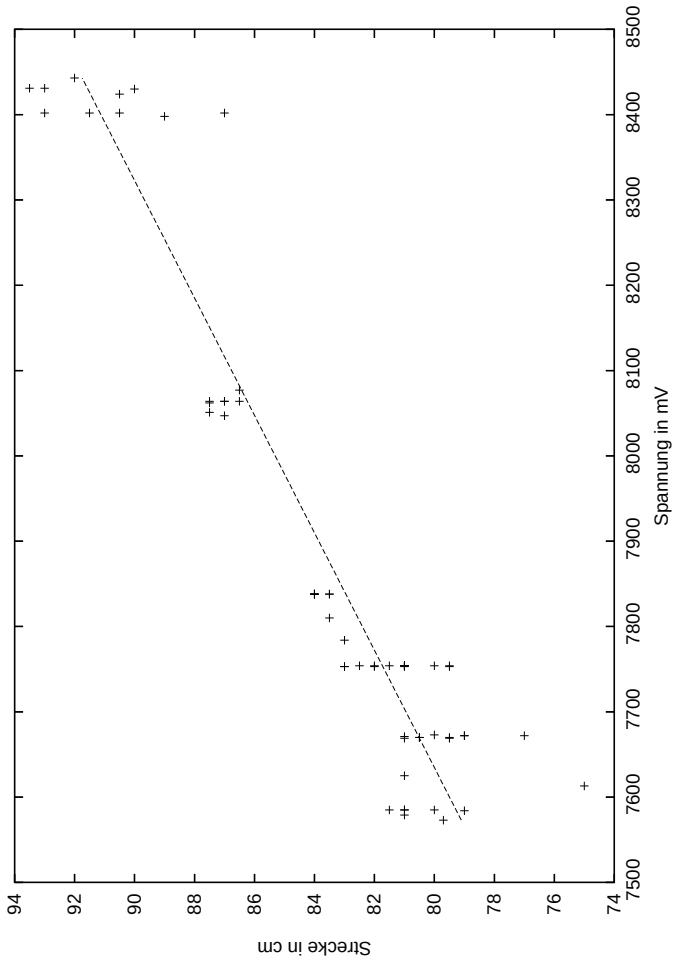


Abbildung 6.1: Ergebnis der Bewegungsmessreihen mit Ausgleichsgerade

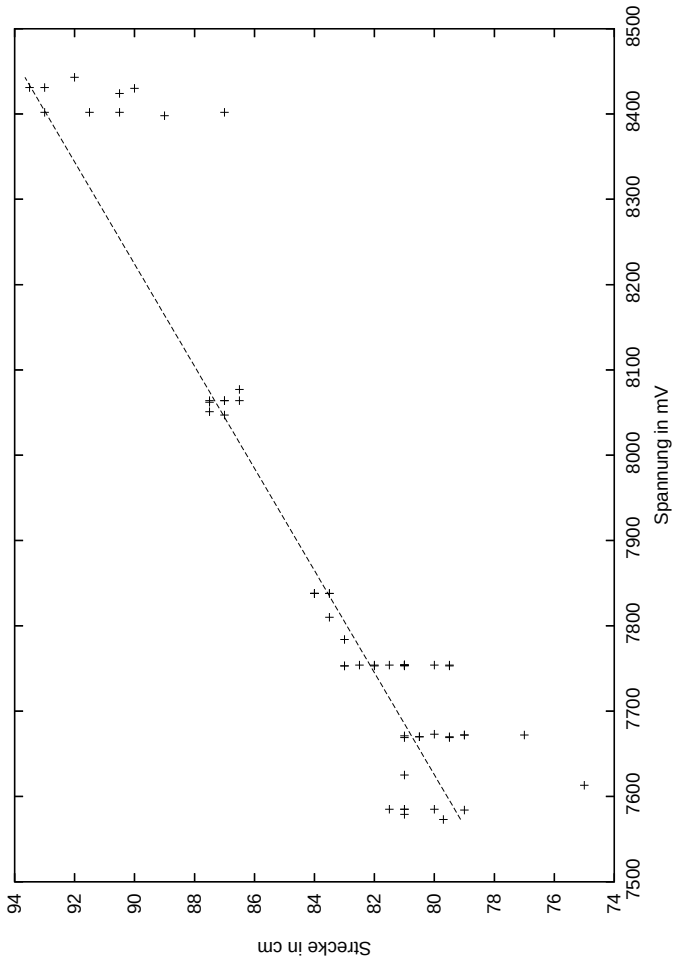


Abbildung 6.2: Ergebnis der Bewegungsmessreihen mit Ausgleichsgerade (beim Berechnen der Ausgleichsgerade wurden „Ausreißer“ berücksichtigt)

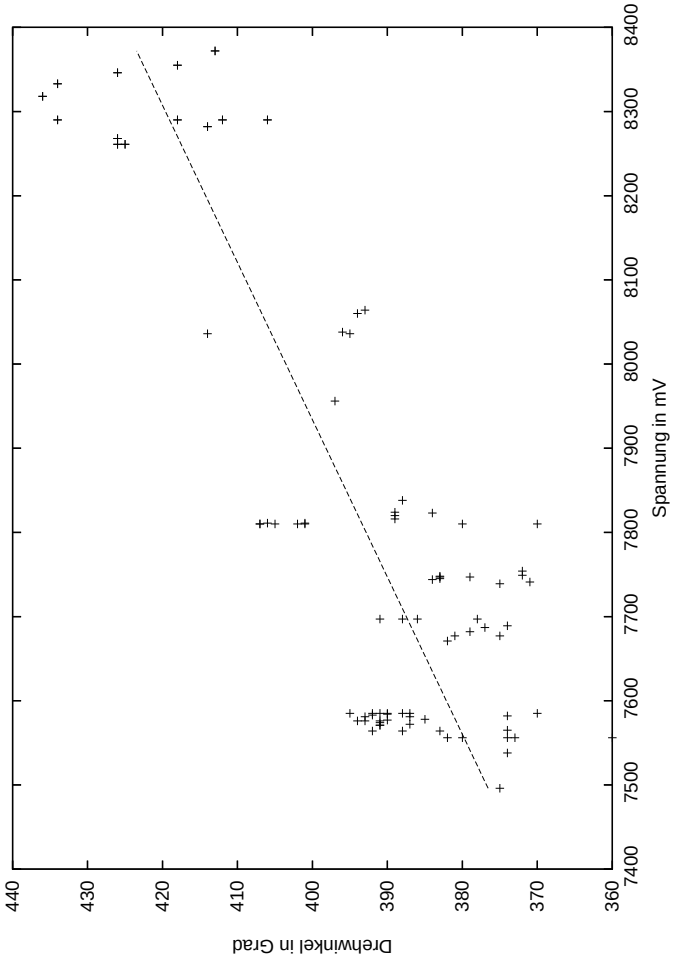


Abbildung 6.3: Ergebnis der Winkelmessreihen mit Ausgleichsgerade

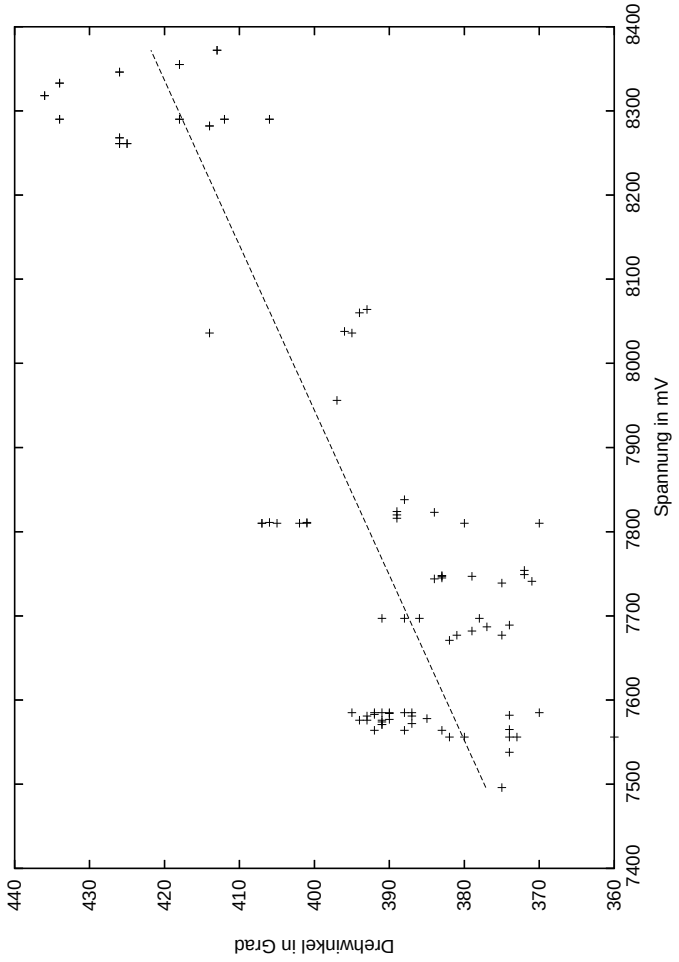


Abbildung 6.4: Ergebnis der Winkelmessreihen mit Ausgleichsgerade (beim Berechnen der Ausgleichsgerade wurden „Ausreißer“ berücksichtigt)

# Kapitel 7

# Die Simulationssprache

In diesem Kapitel geht es um die Befehlsbeschreibungssprache `Sim_Language` und die Klasse `term`, mit der sich Formeln speichern und auswerten lassen.
Die Befehlsbeschreibungssprache dient hauptsächlich dazu, vom Nutzer neu entworfene Befehle richtig durch das Modell darstellen zu können.

## 7.1 Die Befehlsbeschreibung

`Sim_Language` haben wir eingeführt, um es dem Nutzer zu ermöglichen, eigene Befehle genauso durch das Modell darstellen lassen zu können, wie die Elementarbefehle `VORWAERTS, RUECKWAERTS, RECHTS` und `LINKS`. Denn das Modell soll ja nicht nur für diese elementaren Bewegungen korrekte Simulationen liefern, sondern auch für komplexe Befehle, die erst nach der Entwicklungszeit vom Nutzer entworfen werden.
Um also einen neuen Befehl für das Modell verständlich zu machen, d.h. ihm zu verstehen zu geben, welche Bewegungen konkret ausgeführt werden, geht man wie folgt vor. Hat man Namen, Codenummer und den auszuführenden Quelltext in der Konfigdatei festgelegt, muss man sich selbst überlegen, welche Bewegungen bei diesem Befehl ausgeführt werden, damit diese im Modell simuliert werden können. Dies geschieht vermittels der Sprache `Sim_Language`, deren Syntax der der Eingabe in die GUI mit Absicht sehr gleicht (siehe Abschnitt 4.1.1). Für der Beschreibung verwandt werden dürfen bei der aktuellen Implementierung nur die vier Standardbefehle `VORWAERTS, RUECKWAERTS, RECHTS` und `LINKS`. Angenommen, man hat einen Befehl `QUADRAT` programmiert, der den Roboter ein Quadrat der Kantenlänge 50 abfahren läßt. Um ihn zu beschreiben, gebe man in der Konfigurationsdatei (siehe hierzu auch Kapitel 2) folgendes ein:

```
DIM_QUADRAT = 0
CMD_QUADRAT = VORWAERTS(50) RECHTS(90) VORWAERTS(50) RECHTS(90)
VORWAERTS(50) RECHTS(90) VORWAERTS(50) RECHTS(90)
```

Die Parameteranzahl muss immer mit angegeben werden. In diesem Beispiel ist sie 0, der Befehl ist also statisch. Da aber ein sinnvoller Befehl in der Regel nicht statisch ist, sondern von Parametern abhängen sollte, müssen das auch die Elementarbefehle in der Beschreibungssprache. Es ist also genauso möglich, einen Befehl zu beschreiben, nach dessen Aufruf der Roboter ein Quadrat beliebiger

Kantenlänge abfährt. Das heißt, in der Beschreibung müssen in den Parametern der Elementarbefehle Formeln stehen, die irgendwie von den übergebenen Parametern des Befehls `QUADRAT` abhängen.

## 7.2 Formeln als Beschreibungselement

In dem Beschreibungstext eines Befehls kann der Nutzer für die Parameter neben Konstanten auch Formeln angeben, die von den Parametern des Befehls selbst abhängen. Die Anzahl dieser Parameter ist durch den Nutzer festgelegt worden. Sie steht in dem Eintrag mit dem Präfix „`DIM_`“. Um einen Parameter in einer Formel referenzieren zu können, wird eine laufende Nummer, beginnend mit 0, vergeben. Die Parameter heißen dann: `p_0, p_1, p_2,...`
Zurück zum Beispiel: Zunächst legt man `QUADRAT` auf die Parameterzahl 1 (statt 0 wie bisher) fest. Dazu gebe man in die Konfig-Datei `DIM_QUADRAT = 1` ein. Dieser Parameter soll die Kantenlänge festlegen. Im Beschreibungstext sollte dann folgendes stehen:

```
DIM_QUADRAT = 1
CMD_QUADRAT = VORWAERTS(p_0) RECHTS(90) VORWAERTS(p_0) RECHTS(90)
VORWAERTS(p_0) RECHTS(90) VORWAERTS(p_0) RECHTS(90)
```

Die Formeln können beliebige Ausdrücke mit den Grundrechenarten `+ - * /` und der Potenzierung `^` sein. Es kann beliebig verschachtelt werden, es muss aber nicht alles geklammert werden - „Potenz-vor-Punkt-vor-Strich-Rechnung“ wird von Sim-Language erkannt. Als Zahlen akzeptiert werden nur Ganzzahlen, aber gebrochene Werte können einfach durch einen Bruch erreicht werden.
Beispiel für eine syntaktisch korrekte Formel:

```
1/((1/(30*800))*(1631/10+(28/100)*p\_0))+p\_2^2
```

Zum technischen Hintergrund: Eine Formel wird von einem Algorithmus nach und nach in einen einzigen Term (Objekt der Klasse `term`) verwandelt, der nur einen Operator und zwei Operatoren hat. Als Operatoren kommen Zahlen, Variablen (`p_0, p_1, ...`) oder wiederum Terme in Frage. Durch diese Verschachtelungsmöglichkeit lassen sich beliebig komplexe Formeln speichern und leicht auswerten.
Mit diesen wenigen Operationen lassen sich übrigens auch boolean-Parameter simulieren: Angenommen, man wollte von einem zweiten Parameter abhängig machen, ob der Roboter das Quadrat im oder gegen den Uhrzeigersinn abfahren soll. Während man im Quelltext des Befehls einfach den Wert abfragt und dann entsprechend handelt, kann man das Verhalten in der Beschreibungssprache wie folgt ausdrücken:

```
DIM_QUADRAT = 2
CMD_QUADRAT = VORWAERTS(p_0) RECHTS(90*(1-p_1)) LINKS(90*p_1)
VORWAERTS(p_0) RECHTS(90*(1-p_1)) LINKS(90*p_1) VORWAERTS(p_0)
RECHTS(90*(1-p_1)) LINKS(90*p_1) VORWAERTS(p_0) RECHTS(90*(1-p_1))
LINKS(90*p_1)
```

Befiehlt man nun `QUADRAT(75,1)`, simuliert das Modell eine Fahrt durch ein Quadrat mit Seitenlänge 75 gegen den Uhrzeigersinn, mit `QUADRAT(75,0)` in

Gegenrichtung. Wichtig ist natürlich, dass man immer für die Kohärenz von Quelltext und Beschreibungstext Sorge trägt. Dies entfällt, wenn man den Befehl als Makro (siehe Abschnitt 7.3.2) implementiert. Falls er tatsächlich nur ein Quadrat abfahren soll, ist dies zwar möglich und sehr einfach, aber es hat gegenüber dem vollwertigen Quelltextbefehl auch Nachteile. Wichtig ist aber, dass für beide Formen der Implementation da Modell die richtige Simulation liefert.
Natürlich kann das Modell bei einem so beschriebenen Befehl, wie bei jedem Standardbefehl, auch den korrekten Haltepunkt bei einem Anstoßen des Roboters rekonstruieren. Wieder ist hier die Beschreibung zu befragen.
Achtung: Zurzeit lassen sich durch Sim_Language keine Konstrukte wie for-Schleifen innerhalb eines Roboterbefehls wiedergeben.

## 7.3 Andere Anwendungen im Programm

Das Wissen darüber, welche Bewegungen der Roboter bei einem Befehl macht, der ursprünglich gar nicht implementiert war, ist auch an anderen Stellen von Vorteil.

### 7.3.1 Zeitprognosen

Wenn der Server gerade sichergestellt hat, dass alle Befehle richtig beim Roboter angekommen sind und der Roboter beginnt, diese auszuführen, muss der Server auf das Ende der Ausführung warten, damit er die Kommunikation zuende führen kann. Damit der Server nicht zu lange wartet, aber auch nicht zu oft eine kurze Warteschleife durchlaufen muss, kann die Ausführdauer durch die Beschreibung ziemlich genau prognostiziert werden.
Und hier kommt auch noch ein andere Anwendung von Sim_Language zum Tragen. Denn die Elementarbefehle (`VORWAERTS, RECHTS,...`) werden bei der Kompilierzeit der Roboterprogramms derart angepasst, dass sie zu der dann aktuellen Batteriespannung exakt vorbestimmt Einheiten pro Parametereinheit zurücklegen oder drehen. Das heißt konkret, dass `VORWAERTS(5)` den Roboter immer genau 2.5cm nach vorn bewegen und `RECHTS(17)` den Roboter immer genau um 17° nach rechts drehen sollte. Für das Modell gibt der Parameter also direkt die Strecke an, die gefahren wird.
Aber die Ausführzeit hängt leider weder linear mit der zurückgelegten Strecke noch mit dem Drehwinkel zusammen, wie wir feststellen mußten. Vielmehr hängt sie immer auch von der aktuellen Batteriespannung des Roboters zusammen. Der Nutzer soll davon möglichst nichts merken. Er soll immer das Parameterspektrum 1–250 für seine Befehle haben. Da der Roboter aber nur ebendiese Werte empfangen kann, muss die Umrechnung, die die fallende Batteriespannung nötig macht, Roboter-intern passieren. Daher wird vor jedem Neukompilieren des Roboterprogramms ein Faktor in dessen Quelltext geschrieben, der für die richtige Ausführungsdauer sorgt. Und die Formeln der Fahr- und der Drehbewegung in Abhängigkeit von der Batteriespannung werden eben wieder durch einen Term von Sim_Language ausgedrückt.

### 7.3.2 Makros

Falls man einen Befehl implementieren will, der ausschließlich die normalen Bewegungen benutzt, ist es möglich, hierfür einfach ein Makro anzulegen. Gegenüber der oben erklärten Quelltextvariante ergeben sich keine Einschränkungen. Und die Implementation ist ebenso einfach. Die Parameteranzahl („Dimension“) ist ebenso anzugeben wie gehabt. Die Angabe des Quelltextes entfällt natürlich. Es genügt zur vollständigen Implementation den Beschreibungstext anzugeben, aber nicht mit dem Präfix „`CMD_`“, sondern mit „`MAKRO_`“.
Allerdings muss man sich vor Augen halten, dass ein Makro eben nur eine Ersetzung ist. Ein Aufruf von `QUADRAT` als Makro zöge dann nicht mehr nur das Senden von einem Code und zwei Parametern an den Roboter nach sich, sondern von 8 Codes mit je einem Parameter. Da der Roboter aber nur eine begrenzte Anzahl von etwa 20 Zahlen zwischenspeichern kann, die danach ausgeführt werden, sollte man anstatt von sehr langen Makros lieber einen neuen Befehl schreiben, der dasselbe leistet, aber dann nur mit einem Code und ein paar Parameter aufgerufen werden kann.

# Kapitel 8

# Netzwerkkommunikation

Dieses Kapitel nimmt Bezug auf die in Eukalyptos implementierten Netzwerkmodule. Es stellt die verwendeten Netzwerkprotokolle vor und erläutert die Verwendung der Module.

## 8.1 `module_net`

`module_net` ist ein abstraktes Basismodul, das die Grundfunktionalität eines Netzwerkmodules beschreibt. Es enthält Prototypen für Sende- und Empfangsfunktionen. Eine Implementierung dieses Moduls muss zudem die Möglichkeit bieten, Verbindungen zu anderen Computern herzustellen. Ein Client-Server-Modell ist aber seitens der Schnittstelle ebenso möglich. Dies war schließlich auch der Weg, den wir in diesem Projekt beschritten haben.

## 8.2 `module_net_tcp`

`module_net_tcp` ist eine mögliche Implementierung des oben beschriebenen Moduls. Alle Netzwerkfunktionen sind hier über TCP/IP realisiert. Dazu werden die Standard-UNIX-Sockets verwendet, die unter den größeren Plattformen Windows und Linux zur Verfügung steht. Bei Windows finden sich die meisten Netzwerk-Bibliotheken leider unter anderen Namen wieder als bei UNIX, sodass dieses Modul momentan noch nicht plattformunabhängig verwendbar ist. Durch Verwendung von WinSock kann dies bei Bedarf geändert werden.
TCP/IP wurde von uns als übertragendes Netzwerkprotokoll ausgewählt, da es eine automatische Fehlerkorrektur bietet und Pakete im Gegensatz zu Protokollen wie UDP auch mit einer hohen Latenzzeit noch ankommen.

### 8.2.1 Basisfunktionalität

Das Modul beherrscht sowohl einen Client- als auch einen Servermodus. Damit sind einerseits TCP/IP-Verbindungen zu beliebigen Computern auf beliebigen Ports möglich, andererseits kann man auf jedem Computer mit TCP/IP-Stack einen Server einrichten, der an einem bestimmten Port auf Verbindung „lauscht“.

Alle Funktionen sind blockierend und nicht nebenläufig; das Modul wartet also zunächst auf einen Client, wenn man es als Server verwendet. Erst daraufhin wird der nachfolgende Quelltext ausgeführt.
Der Server unterstützt zur Zeit nur eine aktive Clientverbindung. Mehr ist für unser Projekt nicht notwendig. Eine Erweiterung ist aber denkbar.

### 8.2.2 Verwendung

In diesem Beispiel wird das Modul als Client verwendet, der sich auf den Port 12345 des lokalen Computers verbindet. Danach versendet er eine Nachricht an den Server.

Listing 8.1: Verwendung als Client

```
1 module_net_tcp mod_tcp;
2 mod_tcp.connect( "localhost", 12345 );
3 mod_tcp.send_msg( "Hello, World!", strlen( "Hello, World!" ) )
    ;
```

Hier wird nun ein Server erstellt, der auf dem Port 1024 „lauscht“. Sobald eine Client-Verbindung hergestellt wurde, wartet er darauf, dass etwas gesendet wurde, und zeigt dieses dem Benutzer an.

Listing 8.2: Verwendung als Server

```
1 module_net_tcp mod_tcp;
2 mod_tcp.connect( 1024 );
3
4 char* buf;
5 while( mod_tcp.has_msg() >= 0 )
6 {
7         buf = new char[ mod_tcp.has_msg() ];
8         mod_tcp.recv_msg( buf, mod_tcp.has_msg() );
9         printf( "recv: %s\n", buf );
10        delete[] buf;
11 }
```

## 8.3 Das Eukalyptos-Protokoll

Unter Verwendung der oben beschriebenen Module haben wir ein einfaches Protokoll zur Kommunikation mit dem Serverrechner entwickelt. Es geht von einem Client-Server-Modell aus: Der Computer mit der grafischen Benutzeroberfläche stellt den Client dar, wohingegen der Computer, der die direkte Kommunikation über den Infrarot-Turm mit dem Roboter abwickelt, die Serverrolle übernimmt. Das Protokoll setzt nun auf der obersten Schicht, der Anwendungsschicht, des OSI-Referenzmodells[1] auf: Jede Netzwerknachricht besteht zunächst aus einem Header. Dieser enthält den Typ der Nachricht, das zu sendende Kommando sowie die Größe der Parameternachricht in Bytes. Die Parameter der Nachrichten folgende dem Header als einfacher Strom von Bytes.

[1] Das Open Systems Interconnection Reference Model ist ein Schichtenmodell zur Netzwerkkommunikaton. Es standardisiert die Netzwerkkommunikation. Auf der Anwendungsschicht finden sich unter anderem Protokolle wie HTTP und FTP.

Listing 8.3: Header des Protokolls (gekürzt)

```
1 class net_msg_hdr
2 {
3         unsigned char type;
4         unsigned char cmd;
5         size_t param_size;
6 };
```

Momemtan werden die Nachrichtentypen `MSG_TYPE_SYSTEM` und `MSG_TYPE_ROBOT` verwendet. Der erste Typ von Nachrichten beschreibt alle Systemkommandos, wie beispielsweise das Ausschalten des Roboters. Der zweite Typ von Nachrichten beschreibt alle Nachrichten, die direkt an den Roboter gesandt werden sollen, beispielsweise Kommandos wie „Vorwärts“. Da der RCX2.0 nur Nachrichten von 0-255 akzeptiert, reicht der Datentyp **unsigned char** aus.

### 8.3.1 Systemnachrichten

Im Folgenden werden die implementierten Systemnachrichten kurz vorgestellt. Siehe hierzu auch die Datei `protocol.h`.

`MSG_PING` Eine Ping-Nachricht ohne Parameter, mit der Client und Server überprüfen können, ob die Netzwerkverbindung noch besteht. Diese Nachricht wird von Client und Server erkannt, aber nicht beantwortet.

`MSG_ANIMA` Veranlasst den Server, das Roboterprogramm neu auf den RCX2.0 zu übertragen. Hat keine Parameter.

`MSG_INANIMA` Schaltet den Roboter aus. Hat keine Parameter.

`MSG_EXECUTION_TIME` Im Parameterblock dieser Nachricht wird die voraussichtliche Dauer der Befehlsausführung in Sekunden übertragen. Dies ist nötig, damit das Serverprogramm nicht das Datalog des Roboters abfragt, bevor dieser alle Befehle ausgeführt hat (siehe hierzu auch Kapitel 6).

`MSG_EXECUTED_COMMANDS` Als Parameter enthält diese Nachricht das Datalog des Roboters nach Ausführung der Befehle. Dieses Datalog wird dann zur Auswertung an das Modell übergeben. Eine genauere Beschreibung ist in Kapitel 6 zu finden.

`MSG_BATTERY_VOLTAGE` Der Parameter dieser Nachricht gibt die aktuelle Batteriespannung in mV an.

`MSG_LOG` Zur Übertragung von wichtigen Hinweisen und Fehlermeldungen vom Servermodul zum Client wird diese Nachricht verwendet. In ihrem Parameterblock findet sich die entsprechende Meldung wieder. Diese kann beispielsweise durch das Logmodul (siehe Kapitel 5) angezeigt werden.

### 8.3.2 Roboternachrichten

Dieser Typ von Nachrichten wird mit allen Parametern an den Roboter weitergereicht. Welche Nachrichten auftreten können, hängt vom Benutzer des Programms ab; dieser kann nach Belieben Änderungen in der Konfigurationsdatei (siehe Kapitel 2) durchführen.

Eine Konversation mit dem Roboter besteht aus einer Anzahl von Nachrichten, die im Block an den Roboter weitergeleitet werden. Nach Abarbeitung der Nachrichten ist für den Roboter die Konversation beendet. Im Verlauf einer Kommunikation gibt der Roboter dem Benutzer mehrmals die Gelegenheit, das Datalog auszulesen. Dies wird beispielsweist eingesetzt, um zu überprüfen, ob der Roboter alle Befehle erfolgreich empfangen hat.
Die folgende Liste führt alle Nachrichten auf, die der Roboter zum Zeitpunkt der Fertigstellung des Programms versteht.

`START_PHASE_1` Befiehlt dem Roboter, nach Befehlen zu lauschen. Diese Nachricht geht jeder Konversation mit dem Roboter voraus.

`START_PHASE_2` Befiehlt dem Roboter, mit der Ausführung aller empfangenen Befehle zu beginnen.

`ROBOTERBEFEHL` Signalisiert dem Roboter, dass alle Signale der aktuellen Konversation Roboterbefehle sind. Denkbar ist hier das Hinzufügen anderer Befehlstypen, die beispielsweise auf die Interna des RCX2.0 Einfluss nehmen.

`ABBRUCH` Stoppt die Ausführung aller Befehle, lässt den Roboter aber angeschaltet. Nach Empfang dieses Kommandos kann eine neue Kommunikation mit dem Roboter erfolgen.

`MASZGESCHNEIDERTER_BEFEHL` Lässt den Roboter mit der Ausführung der vom Benutzer selbst erstellten Quelltexte beginnen.

# Kapitel 9

# Infrarot-Kommunikation

Als physisches Mittel zum Informationsverkehr mit dem Roboter dient unserem Projekt der USB-Sende- und Empfangsturm aus dem Baukasten „Mindstorms: Robotik Inventions System 2.0“ - der Lego-Standardweg. Warum wir diese Wahl trafen und wie die technischen Details der Nahsteuerung des Roboters aussehen, ist Thema dieses Kapitels.

## 9.1 Der Lego-USB-Turm

Bei der Wahl des Gerätes gab es von vorneherein nicht viele Optionen, da die RCX-Bausteine nur Signale aus infrarotem Licht empfangen können. Der USB-Turm von Lego hat zwei bedeutende Schwächen:

- Die Sendereichweite, d. h. die Entfernung, in welcher die Signale vom RCX noch empfangen werden, ist sehr gering und obendrein von der Umgebungshelligkeit abhängig. In unseren Tests schwankte die maximale Reichweite für die Übertragung von Programmen zwischen einem (volle Laborbeleuchtung) und etwas mehr als zwei Metern.
- Das passive Erwarten und Empfangen von Nachrichten ist unmöglich. Der Turm besitzt nämlich keinen Bereitschaftsmodus, sondern ist nur kurz nach eigener Sendeaktivität für Signale von außen empfänglich.

Diesen Mali standen folgende Vorteile entgegen:

- Einfachheit der Benutzung. Da der USB-Turm für den RCX 2 gebaut und die benutzten Programme auf ihn abgestimmt sind, würden wir uns nicht mit Treiberdetails oder dem RCX-Protokoll auseinanderzusetzen haben.
- Einfachheit der Reproduktion. Unser Ziel war von Anfang an ein System, das auch Anfänger und Dilettanten in der Kunst der Legorobotik mühelos verwenden können. Dass keine außergewöhnlichen Geräte benötigt werden, ist ein erster, wichtiger Schritt in diese Richtung.

Alles in allem blieben wir also beim USB-Turm. Ob sich mit anderen Geräten - zum Beispiel dem Universalgerät „IRTrans“ (http://www.irtrans.de/) - bessere Ergebnisse erzielen lassen, mögen nachfolgende Entwickler überprüfen.

## 9.2 Software zur Kommunikation mit dem Roboter

### 9.2.1 Installation unter Linux

Die Installation der nötigen Programme verlief auf den Laborrechnern (PCs mit SuSe Linux als Betriebssytem) fast problemlos. Dies sind:

- Der Treiber „LegoUSB“ (http://sourceforge.net/projects/legousb/), als Kernelmodul erhältlich.
- Der Linux-NQC-Kompilierer (http://bricxcc.sourceforge.net/nqc/), ein Konsolenprogramm, welches auch Funktionen für die Kommunikation bietet.

Der LegoUSB-Treiber wurde vom Systemadministrator der Laborrechner (Thomas Klöpfer ) für uns installiert. Bei der Kompilierung des NQC-Kompilierers war zu beachten, dass vorher im Quelltext die den USB-Turm betreffende Zeile entkommentiert wird, da dieser ansonsten nicht funktioniert:

```
# uncomment this next line if you have the USB tower library installed
# USBOBJ = rcxlib/RCX_USBTowerPipe_linux.o
```

Schließlich gab es noch Schwierigkeiten mit dem Gerätenamen des Turms im /dev-Verzeichnis, da jener nicht mit dem übereinstimmte, unter welchem der NQC-Kompilierer den Turm suchte. Die Lösung war eine symbolische Verknüpfung des fehlenden Namens, die wieder Thomas Klöpfer für uns anlegte.

### 9.2.2 Installation unter FreeBSD

NQC ist mittlerweile auch für das BSD-Derivat FreeBSD (http://www.freebsd.org verfügbar. Hier ist die Installation[1] mit weniger Hürden verbunden. Die folgenden Eingaben sind auf der Kommandozeile zu tätigen:

```
cd /usr/ports/lang/nqc/
make install clean
```

Danach ist der NQC-Compiler einsatzbereit.

**Warnung:** Der NQC-Compiler unter FreeBSD hat noch mit ein paar Bugs zu kämpfen. So hängt sich das System beispielsweise reproduzierbar auf, wenn man versucht, auf einem seriellen Turm NQC-Kommandos wie `nqc -run` auszuführen. Die Ausführung von Kommandos ist davon jedoch nicht betroffen, sodass Eukalyptos unter FreeBSD perfekt funktioniert.

[1] Im Folgenden wird davon ausgegangen, dass der Benutzer über root-Rechte verfügt und die sogenannten „FreeBSD Ports“, eine Sammlung von Kompilierungsanweisungen für viele Programme, auf dem Computer installiert hat. Dies ist bei allen Standardinstallationen der Fall.

### 9.2.3 Funktionen zum Datenaustausch - Die Qual der Wahl

Was die Wahl der Schnittstellen für den Datentransfer vom Rechner zum Roboter und zurück anging, schien die Auswahl zunächst verwirrend groß. Doch schmolz die Anzahl der Möglichkeiten schon nach einer flüchtigen Betrachtung dahin: Der Unfähigkeit des Sendeturms zu passivem Nachrichtenempfang wegen mussten alle Ideen verworfen werden, bei denen der Roboter von sich aus Daten gesendet hätte - aus der Sprache NQC seien die Funktionen `SendMessage()` und `SendSerial()` genannt. Als einzige uns sichtbare Lösung blieb der sogenannte „Datalog“ übrig, ein Speicher im RCX, welcher nur beschrieben und gelöscht, nicht jedoch intern gelesen werden kann, der sich jedoch von Ferne auslesen lässt. Der NQC-Kompilierer bietet hierzu die Funktionen `-datalog` und `-datalog_full`:

```
nqc -Susb -datalog
nqc -Susb -datalog_full
```

Die NQC-Funktionen zum Löschen, Erstellen und Beschreiben des Datalogs lauten folgendermaßen:

```
CreateDatalog(0);
CreateDatalog([Gr"o"se in int]);
AddToDatalog([int]);
```

Von den Funktionen zum Datentransfers vom Rechner zum RCX - `-remote`, `-raw` und `-msg` - ließen sich die ersten beiden auch rasch ausschließen:

- Das `-remote`-Kommando ist auf die Nachahmung der Signale der originalen Lego-Fernbedienung zugeschnitten und dementsprechen in seinen Möglichkeiten beschränkt: Im Wesentlichen lassen sich die drei Ausgänge des RCX an- und ausschalten. Hinzu kam, dass Dokumentationen und Anleitungen zur Benutzung dieser Funktion im Internet äußerst rar sind. Für unsere Pläne brauchten wir etwas Flexibleres.

- Das `-raw`-Kommando ist zwar sehr flexibel: Es dient nämlich zur manuellen Kontrolle eines jeden einzelnen Bits, welches gesendet wird. Jedoch wollten wir ja ein komplexes Programmsystem entwickeln, also keine Zeit mit den Feinheiten des Lego-Protokolls vergeuden.

- Das `-msg`-Kommando war die Lösung. Mit der Möglichkeit, Signale von der Größe eine Bytes zu verschicken, bietet es bei Weitem ausreichenden Freiraum für eine eigene kleine Nachrichtensprache, zudem ist der Gebrauch äußerst simpel. Schon die folgende Zeile reicht aus, die Zahl 42 zu versenden:

  ```
  nqc -Susb -msg 42
  ```

  Dass der Roboter unsere Befehle richtig verstünde, würde ein NQC-Programm besorgen müssen.

## 9.3 Die harte Wirklichkeit

Das erste Hemmnis, welches der Lego-USB-Turm seinen Benutzern auferlegt, hatten wir mit dem Datalog also beseitigt. Blieb das zweite: Wie sollte mit der geringen Sendeleistung des Turmes umgegangen werden? Zwar waren gute Programme das Hauptziel, nicht Gerätschaften, aber zumindest das 2x2 Meter große, quadratische Versuchsgelände im Labor wollten wir mit unserer Fernsteuerung abdecken können, ohne den Turm am Roboter zu montieren (der dann das Kabel hinter sich hergeschleift hätte) oder ihn dem Roboter in den Weg zu stellen. Unsere Ansatzpunkte waren

- die Raumbeleuchtung.
- die Position des Turmes.
- die Stellung des RCX-Blockes in der Roboterkonstrukion.
- Empfangs- und Sendehilfen für den Roboter aus Aluminium.

### 9.3.1 Beleuchtung

Die einfachste Methode, um die Übertragung zu verbessern. Bei völlig Dunkelheit (vermutlich auch bei niedriger Temperatur, was wir aber nicht ausprobiert haben) funktionierte das Senden und Empfangen beiderseits am Besten, darauf wollten wir uns jedoch nicht beschränken. Wer hat schon Spaß an seinem Roboter, wenn er ihn nur im Keller benutzen und ihm nicht einmal zusehen kann?

### 9.3.2 Die zwei Türme

Die Frage der Positionierung des Sendeturmes gliedert sich in zwei:

- Wo soll der Turm hin?
- Wie beim Hummel kriegen wir ihn dorthin?

Das erste ist offenbar ein rein stereometrisches Problem und lässt sich leicht lösen: An einer Ecke des Versuchsgeländes plaziert, hat die IR-Verbindung $\sqrt{2^2 + 2^2}m = 2 * \sqrt{2}m$ zu überbrücken, was deutlich suboptimal ist. Die beste Sicht auf alle vier kritischen Punkte, die Ecken nämlich, hat man von einer schwebenden Position über der Mitte der quadratischen Fläche aus. Diesem Ideal näherten wir uns auf zwei Wegen an:
Erstens mit einer sockelartigen Konstruktion aus Lego, von welcher aus der Sendeturm immer noch aus einer Ecke, allerdings schräg aus einer Höhe von 22 Zentimetern auf das Versuchsgelände hinabblickte. Die Resultate waren nicht zufriedenstellend: In den entfernten Ecken war der Empfang empfindlich gestört, teils sogar unmöglich. An diesem Punkt unserer Versuche stellten wir fest, dass der RCX Nachrichten zwar problemlos von fast allen Seiten aus empfangen kann, dass seine Rückmeldung aber nur beim Turm ankommt, wenn dieser im Blickfeld des kleinen Sendenfensters vorne am Baustein liegt.
Zweitens hängten wir den Sendeturm in einer Höhe von etwa $1,40$ Metern mittig über dem Versuchsgelände auf und brachten ihn mit Legosteinen als Ballast in

Abbildung 9.1: Der aufgehängte IR-Turm

eine waagrechte Lage. Dies verbesserte die Verbindung deutlich, nur die übermittlung vom Roboter zum Turm funktionierte noch nicht, wenn die Frontseite des RCX von jenem weg zu einer nahen Ecke hinwies. Abbildung 9.3.2 illustriert dieses Vorgehen.

### 9.3.3 Montage des RCX

Die neue Position des Infrarotturmes hängend über dem Boden legte nahe, es mit einem hochkant stehenden RCX zu versuchen, das Sendefensterlein nach oben gerichtet. Und siehe da: Mit dem so modifizierten Roboter gab es keinerlei Empfangs- oder Sendeprobleme mehr, und zwar weder mit dem hängenden, noch mit dem seitlich erhöht aufgestellten Turm. Da sich jedoch beim Umbau die Masse des Roboters mehr im Drehzentrum konzentriert und sich somit sein Trägheitsmoment vermindert hatte, wären die zu diesem Zeitpunkt unseres Projekts schon durchgeführten Messungen zur Rotationsgeschwindigkeit des Roboters (siehe Abschnitte 6.2 und 6.3) zu verwerfen und neu durchzuführen gewesen, hätten wir mit der Hochkant-Lösung weiterarbeiten wollen. So gaben wir sie auf und experimentierten weiter.

### 9.3.4 Antennen

Der Durchbruch gelang schließlich, indem wir die zwischen Roboter und hängendem Sendeturm hin- und herstrahlenden Signale mit einer schüsselförmigen Antenne aus Aluminiumfolie an der Front des Roboters bündelten.

Diese Methode ist eigentlich sehr naheliegend, wenn man bedenkt, dass Tag für

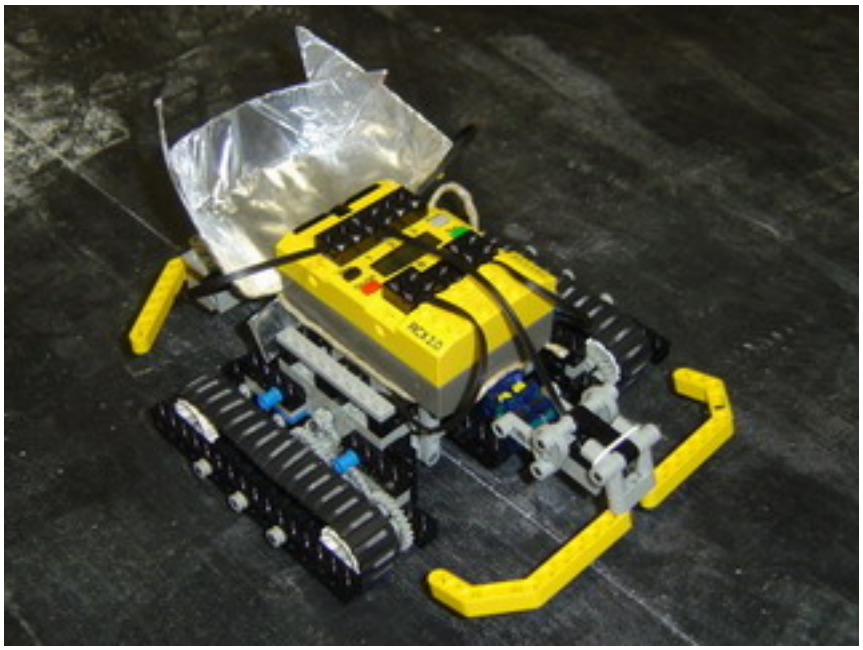

Abbildung 9.2: Der Roboter mit optimierter Antenne

Tag Aluminiumfolie in zahllosen Öfen der Welt infrarote Wärmestrahlen, die von den eingewickelten Speisen ausgehen, mit ihrer glänzenden Seite zu diesen zurückwirft und so den Garungsprozess fördert. Eben mit infrarotem Licht hat man es ja auch bei den RCX zu tun!

Eine geeignete Form war schon schwieriger zu ermitteln. Zuerst konzentrierten sich unsere Experimente auf nach oben hin konvexe Formen - von Zylinderschalen bis zu Kugeln -, da diese die Signale in möglichst viele Richtungen weiterzuleiten versprachen. Der Erfolg jedoch blieb aus, vermutlich wurden die Signale zu stark gestreut und dadurch zu undeutlich. Auch die konkaven Formen funktionierten nicht auf Anhieb. Schließlich gelang es mit einer Antenne mit den ungefähren Ausmaßen $8 \times 12 \times 3$ cm (Länge, Breite, Höhe), deren Form zwischen einer halbierten Badewanne und einer Schneeschaufel anzusiedeln wäre, den Signalstrom so zu bündeln, dass er bei leicht gedämpfter Beleuchtung auf dem ganzen Versuchsgelände funktionierte. In Abbildung 9.3.4 ist das Ergebnis der Optimierungsbemühungen zu sehen.

Zwei andere Ideen, die vorsahen, die Banden des Versuchsgeländes mit Aluminium auszukleiden oder den Turm selbst mit einem bündelnden Alu-Trichter zu versehen, brachten wir nicht zur Ausführung.

### 9.3.5 Zusammenfassung

Folgende Maßnahmen ließen die Infrarotverbindung schließlich zufriedenstellend arbeiten:

- Leichte Reduzierung der Raumbeleuchtung (um ein Viertel)

- Horizontale Aufhängung des Sendeturmes über der Mitte des Versuchsgeländes
- Bündelung der Infratosignale durch eine Aluminiumantenne am Roboter.

# Kapitel 10

# Der Übersetzungsrechner

Dieses Kapitel beschreibt die Softwarekomponenten des Übersetzungsrechners. Dabei handelt es sich um den Rechner, an den der Infrarotturm angeschlossen ist. Neben der Beschreibung des Kompiliermoduls und der Schnittstelle zu NQC soll hier auch ein kurzer Überblick über den genauen Ablauf der Befehlsübertragung gegeben werden.

## 10.1 `module_linuxnqc`

Der kanonische NQC-Kompilierer für Linux[1] ist seinem Wesen nach mehr als nur ein Kompilierer, vielmehr auch eine vollwertige Schnittstelle zur Kommunikation mit dem RCX, wie schon auf S.46 erwähnt worden ist. Daher bildet er für beide zentralen Module des Übersetzungsrechners die Grundlage, die da wären: Das Kommunikationsmodul und das Kompiliermodul.
Damit etwaige modulunspezifische Funktionen, welche nur das Programm nqc beträfen, nicht doppelt geschrieben und später gewartet werden müssten, schufen wir eine zusätzliche Basisklasse für diese Funktionen: `module_linuxnqc`, direkt abgeleitet von `module`:

```
1 struct module_linuxnqc :
2   public module
3 { . . .
```

Um nun eine Modulklasse zu erzeugen, die neben `module_linuxnqc` noch weitere Stammklassen hat, ist mehrfache Vererbung geboten:

```
1 struct module_test_linuxnqc :
2    public module_test, public module linuxnqc
3 { . . .
```

Dieser Kunstgriff macht leider die Konvertierung von Zeigern auf Instanzen der mehrfach abgeleiteten Klassen in `module*`-Zeiger etwas unkomfortabler. Ein klassischer Aufruf der Art

```
1 module_test_linuxnqc* gnomon1;
2 module* gnomon2 = (module*)gnomon1;
```

[1] Hier sei ein weiteres Mal auf http://bricxcc.sourceforge.net/nqc/ verwiesen.

bewirkt einen Kompilierfehler, da - anschaulich gesprochen - der Kompilierer nicht weiß, in welcher Reihenfolge er die Konversion vornehmen soll: `module_test_linuxnqc* → module_test* → module*` oder `module_test_linuxnqc* → module_linuxnqc* → module*`. Brachiale Gewalt löst das Problem:

```
1 module_test_linuxnqc* gnomon1;
2 module* gnomon2 = reinterpret_cast<module*>(gnomon1);
```

Zurück zu `module_linuxnqc`. Unsere Implementierung der Klasse lässt nachfolgenden Entwicklern beeindruckende Entfaltungsmöglichkeiten (anders gesagt: eine Menge Arbeit), denn sie enthält nur eine einzige Methode, und diese obendrein in der einfachsten irgend sinnvollen Ausführung: `proba_rescriptum_nqc` (lat.: prüfe den Rückgabewert von nqc), welche einen Wahrheitswert zurückliefert, übergibt man ihr eine int-Variable als Argument; nämlich **false** im Falle von -1, **true** in allen anderen Fällen. Dieses Vorgehen legt die man-Seite von `system` nahe:

> The system() function returns the exit status of the shell as returned by waitpid(2), or -1 if an error occurred when invoking fork(2) or waitpid(2). A return value of 127 means the execution of the shell failed.

## 10.2 `module_comp, module_comp_linuxnqc`

Die abstrakte Modulbasisklasse `module_comp` definiert die Schnittstellen, welche alle Module zur Kompilation von Roboterprogrammen einem Programmierer zu bieten haben. Das ist neben der obligatorischen elementaren Kompiliere-einen-Quelltext-Methode vor allem eine komplexere Funktion, welche das Hauptprogramm des Roboters kompilieren soll. Ferner gibt es eine private Methode, die Roboterprogrammtexte aus einer Programmiersprache in eine andere transskribieren können soll. Hierbei dachten wir an die Möglichkeit der Entwicklung einer gänzlich abstrakten Programmiersprache, um Roboterprogramme zwischen Systemen, die nur Kompilierer für verschiedene Sprachen (NQC, Lejos, pbForth . . .) bieten, trotzdem portabel zu machen. (Voraussetzung dafür wäre natürlich, daß es für die jeweilige Sprache ein ausreichend implementiertes `module_comp` gäbe.) Die Implementierung eines solchen Systems jedoch, und sei es nur für NQC, hätte den Rahmen unseres Praktikums zweifellos gesprengt. Und so blieb die Schnittstelle unerfüllt.
Denn kompliziert genug war alleine schon die eben noch „komplexer“ genannte Funktion zur Kompilierung des Roboterhauptprogrammes, welche in `module_comp_linuxnqc` den Namen `confunde_fundamen` (lat.: verschmelze/kompiliere die Grundlage) trägt. Hier ein äußerst knapper überblick über ihre drei Arbeitsschritte:

1. Für die Feinjustierung des Hauptprogrammes (siehe Kapitel 11) und später für das digitale Modell der Roboterumgebung wird der Wert des Roboters aktueller Batteriespannung benötigt. Zu diesem Zweck wird ein Hilfsprogramm übertragen, das diesen Wert in den Datalog schreibt, wonach er von außen gelesen werden kann.

2. Im Gerüst des Roboterhauptprogrammes sind eine Vielzahl von Löchern automatisch zu füllen, darunter viele Konstanten aus der Konfigurationsdatei, einige Konstanten, die aus der Batteriespannung mittels Formeln

aus der Konfigurationsdatei jedesmal neu berechnet werden müssen, und nicht zuletzt die Implementierungen sämtlicher Funktionen des Roboters (was Vorwärtsfahren beinhalten kann, Hüpfen, Skilaufen, Deklamieren, et c.).

3. Schließlich wird der erzeugte NQC-Programmtext kompiliert, wobei auf die gesondert definierte, elementare Kompiliermethode `confunde` zurückgegriffen wird.

## 10.3 `module_robocom, module_robocom_linuxnqc`

Auch das Roboterkommunikationsmodul wird zunächst durch eine abstrakte Klasse beschrieben: `module_robocom`. Diese Basisklasse sieht folgende Schnittstellen vor:

- `anima` aktiviert den Roboter.
- `inanima` schaltet den Roboter ab.
- `reanima` haben wir in unserem Falle als schnelles Anschalten des Roboters ohne Neuübertragung des Hauptprogramms verwirklicht.
- `mitte_iussum` sendet dem Roboter einen Befehl. (Argumente sind vom Typ **char**$*$)
- `accipe_nuntios` übergibt die vom Roboter empfangenen Rückmeldungen (und empfängt sie zuvor aktiv neu, je nach verwandter Technik und Implementierung).
- `ping` - selbsterklärend.

Die Verwirklichung in `module_robocom_linuxnqc` zeigt sich wie folgt:

### 10.3.1 `anima, reanima, inanima`

- `anima` führt im Wesentlichen nur drei Aktionen aus (um jegliche Fehlerbehandlung, v. a. mit der Methode aus `module_linuxnqc`, siehe Abschnitt 10.1, gekürzt):
  - Aufruf von `confunde_fundamen` aus dem Kompiliermodul.
  - Herunterladen des Programmes mit dem Befehl `nqc -Susb -d fundamen.temp.rcx`
  - Starten des auf den Roboter geladenen Programmes mit `nqc -Susb -run`
- `reanima` beschränkt sich auf den letzten der drei `anima`-Befehle.
- `inanima` hingegen hat nach unseren Recherchen keine Möglichkeit, den Roboter direkt auszuschalten. (Unser Roboterprogramm bietet zwar eine solche Aktion, die sich ferngesteuert aktivieren lässt, siehe Abschnitte 8.3.2 und 2.6.1, aber `inanima` muss ja vom gerade laufenden Programm unabhängig sein.) Deswegen macht die Funktion einen kleinen Umweg und setzt die Zeitspanne, nach welcher der Roboter sich standardmäßig von selbst abschaltet, sofern niemand ihm zu tun gibt, auf die minimale Zeit (eine Minute): `nqc -Susb -sleep 1`

### 10.3.2 `mitte_iussum`

Sehr unspektakulär. Einfach ein System-Aufruf von:

```
nqc -Susb -msg ...
```

Falls nicht die Nummer, sondern der Name eines Befehles als Parameter übergeben wurde, sucht `mitte_iussum` die Nummer aus der Programmoptionsdatei (siehe Kapitel 2).

### 10.3.3 Zwei Neulinge: `mitte_ordinem`, `obsequere_ordini`

Neulinge? Ja, denn wir hatten ergeizigere Ziele, als den Roboter nur mit vorgefertigten Kommandos wie „rückwärts“ fernsteuern zu können. Vielmehr sollte das Programm die Möglichkeit bieten, spontan eigene NQC-Quelltexte einzugeben, ferngesteuert zu kompilieren, zu laden und auszuführen. Genau dazu dienen `mitte_ordinem` und `obsequere_ordini`, die sich folglich frei aus dem Lateinischen übersetzen lassen als: „Sende Programm“, „Befolge Programm“.
Für dieses Vorhaben erwies es sich als Vorteil, dass RCX mit mehr als einem Programmspeicherplatz ausgestattet sind. Zwei genügen bereits, dass das Hauptprogramm des Roboters nicht gelöscht werden muss, um einen „Schneiderleinbefehl“ auszuführen, wie wir die nutzerbestimmten NQC-Quelltexte in Anlehung an „maßgeschneidert“ nannten.
Zusammengenommen folgen die beiden Funktionen grob folgendem Plan:

1. Vor allem Anderen werden dem Schneiderleinprogramm zu Beginn und zu Ende des `main`-Prozesses einige Zeilen hinzugefügt, mittels derer unser Programm erkennen kann, ob der Roboter sein obskures Programm völlig unbekannter Länge fertig abgearbeitet hat - er schreibt dann nämlich eine charakteristische Zahl in seinen Datalog, der sich bekanntlich von auszen auslesen lässt. Aus diesem Grund raten wir hiermit ausdrücklich davon ab, in selbstgechriebenen NQC-Programmen, die zu Eukalyptos kompatibel sein sollen, auf den Datalog zuzugreifen.

2. Es folgt die Kompilierung des Programmes unter Verwendung des Kompiliermoduls.

3. Der aktive Speicherplatz auf dem RCX wird ferngesteuert (mit dem Befehl `nqc -Susb -pgm`, verborgen in der beschützten Methode `permuta_horreum` gewechselt.

4. Das Programm wird auf den Roboter geladen.

5. Das Hauptprogramm auf seinem alten Speicherplatz wird vorerst neugestartet. Damit geht `mitte_ordinem` zu ende.

6. `obsequere_ordini` wechselt nun nicht etwa ferngesteuert auf den neuen Speicherplatz zurück und startet das dort liegende Programm, nein: Es sendet, möglicherweise am Ende einer langen Kette normaler Befehle - Konfektionsbefehle sozusagen - dem Roboter ein Signal mit der Bedeutung „Jetzt schau in deinen anderen Speicher, vergiss alles, was vorher war, und arbeite nach dieser Anleitung weiter“.

Der letzte Schritt mag etwas befremdlich anmuten, aber er hat jenen entscheidenden Vorteil, der schon angedeutet ward: Wenn vor dem Schneiderleinbefehl eine Kette von Konfektionsbefehlen stehen soll - doch betrachten wir ein Beispiel: Jemand will seinen hochgerüsteten Roboter in eine Höhle fahren und dort eine Sprengladung zünden lassen, wobei Zweites ein Schneiderleinbefehl, Erstes eine normale Vorwärts- und Kurvenfahrt sei. (Ganz ohne Zweifel ein grobes Versäumnis, dass das Zünden von Sprengladungen nicht in die Riege der Standardbefehle aufgenommen wurde.) Wenn nun dort, tief in der Höhle, aber ein Funkloch ist und das Programm nicht ferngesteuert gestartet, geschweige denn geladen werden kann? Dann übertrage der Sprengmeister sein Programm eben im Vorhinein! Denn gerade darauf ist unser System ausgelegt.

## 10.4 Der Server

Wie eingangs erwähnt, haben wir ein klassisches Client-Server-Modell gewählt, das über TCP/IP funktioniert. Der Übersetzungsrechner stellt den Server in diesem System dar. Er nimmt die Befehle des Clients entgegen, die über das Netzwerk übertragen werden, und versucht sie auszuführen.

### 10.4.1 Die Befehlsübertragung

Wie bereits in Kapitel 8 erwähnt, gibt es verschiedene Typen von Nachrichten (momentan System- und Roboternachrichten). Der Ablauf der Befehlsabarbeitung[2] ist wie folgt:

- Sobald der Server eine Roboternachricht empfangen hat, speichert er diese zur späteren Überprüfung und reicht sie an das Roboterkommunikationsmodul weiter (siehe Abschnitt 10.3).
- Dieses wiederum überträgt die Daten (Befehle mit Parametern) vermittels NQC an den Roboter (siehe Abschnitt 9.2.3). Dabei gibt es keine vorgefertigte Möglichkeit, zu überprüfen, ob der Roboter alle Befehle erfolgreich erhalten hat.
- Nachdem alle Daten übertragen wurden, erhält der Server das letzte Startsignal für die Ausführung aller Befehle. Jetzt ruft er das Datalog des Roboters ab und übeprüft es unter Verwendung der Funktion `check_command_datalog` (siehe `main-server.cpp`). Mit dieser Funktion können Übertragungsfehler im Datalog festgestellt werden. Bei Übertragungsfehlern werden nach Möglichkeit nur die „verlorenen“ Daten neu übertragen. Somit ist sichergestellt, dass der Roboter auch die Befehle ausführt, die man ihm geschickt hat.

Während der Befehlsausführung wartet der Server auf den Roboter. Die geschätzte Dauer der Befehlsausführung in Sekunden wurde bereits durch das GUI unter Verwendung von `MSG_EXECUTION_TIME` übertragen. Nach Ablauf dieser Frist versucht der Server, das Datalog des Roboters auszulesen. Anhand der Marke `AUSFUEHRUNGSPROTKOLL_ENDE` (siehe Kapitel 2) kann der Server erkennen, wann

[2] Die Systemnachrichten und ihre Wirkung sind in Abschnitt 8.3.1 erwähnt. Ihre Behandlung ist aber analog zu den Roboternachrichten.

der Roboter die Ausführung der Befehle abgeschlossen hat. Ist dies erst nach Ablauf der vorausberechneten Frist der Fall, wartet der Server.
Nachdem das Datalog erfolgreich ausgelesen wurde, schickt der Server es als Parameter der Nachricht `MSG_EXECUTED_COMMANDS` an den Client (das GUI). Diese Nachricht wird dann verwendet, um die Bewegungen des Roboters im Modell zu berechnen. Siehe dazu auch Kapitel 6.

# Kapitel 11

# Das Roboterhauptprogramm

Jenes Programm, das auf dem RCX läuft und die Befehle unseres Programmes entgegenimmt, ist Gegenstand dieses Kapitels.

## 11.1 Ablauf der Befehlsausführung

Nach dem Aufspielen und Starten des Programms durch das Kompiliermodul (siehe Abschnitt 10.2) ist der Roboter im Bereitschaftsmodus. Hier wartet er auf das Signal `START_PHASE_1` (diese und weitere Roboternachrichten sind in den Abschnitten 8.3.2 und 2.6.1 sowie in Kapitel 2 detailliert beschrieben). Eine gültige „Konversation“ mit dem Roboter läuft hernach wie folgt ab:

- Der Befehl `ROBOTERBEFEHL` wird gesandt.
- Sukzessive werden nun die Kommandos mit Parametern versandt, beispielsweise `VORWAERTS` mit Parameter 100.
- In dieser Phase des Programms erlaubt der Roboter das Auslesen des Datalogs, sodass der Übersetzungsrechner feststellen kann, ob alle Kommandos und alle Parameter erfolgreich empfangen wurden. Siehe hierzu auch die Abschnitte 9.2.3 und 10.4.1.
- Sobald nun die Nachricht `START_PHASE_2` gesendet wird, beginnt der Roboter mit der Ausführung der Befehle. Seitens des Nutzers gibt es jetzt nur noch die Möglichkeit, einen `ABBRUCH`-Befehl zu schicken.

## 11.2 Nach der Befehlsausführung

Während der Roboter alle Befehle ausführt, beschreibt er das Datalog. Für jeden erfolgreich ausgeführten Befehl wird hier die Befehlsnummer notiert. Damit kann das Modell später nachvollziehen, welche Befehle genau abgearbeitet wurden (siehe Kapitel 6). Falls es zu einem Sensorkontakt bei

der Befehlsausführung gab, wird dies zusammen mit Zeit[1] in Dezisekunden, die seit Ausführung des aktuellen Befehls vergangen ist, im Datalog vermerkt.

## 11.3 Erweiterungen

Das kanonische Vorgehen zur Erweiterung des Roboterprogramms ist im Benutzerhandbuch sowie in den Abschnitten 10.2 und 2.4 beschrieben.

[1] Wie in Kapitel 6 beschrieben, rechnet das Modell dann „rückwärts“, wie viel von dem abgeschickten Befehl denn realiter ausgeführt wurde.

# Kapitel 12

# Ausblick

Während der Entwicklung der Software sind uns immer wieder neue Ideen in den Sinn gekommen. Was kann man noch machen? Wozu haben wir noch die Zeit? Was ist unrealistisch? Was sollten wir als Möglichkeit offen lassen? Wo sollten wir unseren Quelltext für eventuelle Änderungen oder Ergänzungen offenhalten? Generell haben wir uns bemüht, unsere Programm möglichst leicht modifizierbar zu halten.
Hier nun eine Liste von Ideen, deren Verwirklichung wir für realistisch und lohnenswert halten.

## 12.1 Andere Legorobotersysteme

Im Hinblick auf die nicht wenigen Einschränkungen, die man bei der Arbeit mit dem RCX 2 zu berücksichtigen hat, erscheint es uns sehr vielversprechend, Eukalyptos fit zu machen für andere Systeme wie NXT. Dies ist, nach allem, was wir bisher erfahren haben, sehr viel leistungsfähiger und entwicklerfreundlicher als das RCX 2-System. Vieles an Quelltext, den wir nur wegen des schwierigen Umgangs mit der eingeschränkten Schnittstelle zum RCX schreiben mussten, würde sicherlich entfallen, und der Code insgesamt noch übersichtlicher und intuitiver aussehen. Aber auch die Abwärtskompatibilität zum RCX 1 wäre sicher machbar und bestimmt eine Herausforderung.

## 12.2 Mehrere Roboter

Sicherlich sehr reizvoll ist es, mit einer grafischen Oberfläche mehrere Roboter in einem oder mehreren Geländen, also auch mit einer oder mehreren Modellkarten in der GUI.

## 12.3 Plattformunabhängigkeit

Für manche Zwecke ist es von Vorteil, wenn das Programm plattformunabhängig funktioniert. Man könnte das Programm für andere Betriebssysteme umschreiben.

## 12.4 Protokolle, Programmiersprachen etc.

Unsere Programme sind, wie eingangs erwähnt, dafür ausgelegt, sehr einfach modifiziert zu werden. Folgende Spezifikationen sollten leicht den eigenen Wünschen und Bedürfnissen angepasst werden können:

- Die Lego-Programmiersprache (bisher NQC)
- Das Netzwerkprotokoll (bisher TCP/IP)
- Die Sprache der Befehle, der GUI (bisher Deutsch)
- Die Format der Konfigurationsdatei
- Die Infrarotschnittstelle (bisher Lego USB-IR-Turm)

## 12.5 Verbindung mit anderen Lego-Praktika

Es ist möglich und unter Umständen sehr nützlich unser Programm für bestehende oder kommende Lego-Praktika zu verwenden. Da vermutlich bald schon keine RCX-Systeme, sondern das neue NXT-System verwendet werden wird, könnte ein Praktikum darin bestehen, unsere Programm hierfür kompatibel zu machen.

## 12.6 Modell, grafische Darstellung

Man könnte das Modell auf die dritte Dimension ausweiten. Um das Modell aufzuwerten, wäre es schön und interessant, einerseits den Roboter durch eine Fototextur realistisch darzustellen und andererseits die Umgebung, die der Roboter erkundet, durch Fotos darzustellen. Dazu könnte der Roboter wie in einem anderen Projekt eine Webcam mit sich führen und dann an das Modell Fotos von der Umgebung oder dem Untergrund schicken.

# Abbildungsverzeichnis

# Listings