
**Universität**
**Siegen**

**FB 12: Elektrotechnik und Informatik**
**FG: Echtzeit Lernsysteme**
Prof. Dr.-Ing. Kuhnert

# Einführung in die Programmiersprache
# Not Quite C

# Inhaltsverzeichnis

# 1 Einleitung

In diesem Umdruck wird die Programmiersprache Not Quite C (kurz: NQC), die von Dave Baum entwickelt wurde, und die Entwicklungsumgebung Bricx Command Center vorgestellt, auf dem das Labor „Mobile Roboter“ basiert. Dazu gehört auch eine Beschreibung der Vorgehensweise bei der Installation des Bricx Command Centers und dessen Bedienung. Das Bricx Command Center ist bereits auf den Laborrechnern installiert worden, kann aber optional auch auf dem Privatrechner installiert werden, da die Software im Internet frei verfügbar ist. Zusätzlich wird noch einer der wichtigsten Ansätze der Roboterprogrammierung nämlich die Subsumtionsarchitektur vorgestellt.

Aber bevor darauf näher eingegangen wird, wird zum besseren Verständnis von NQC und dessen Entwicklungsumgebung der Zusammenhang zwischen der Programmiersprache NQC und dem RCX mit seiner Firmware erläutert. Bevor der RCX in Betrieb genommen werden kann, muss vorher die Firmware zum RCX übertragen worden sein. Danach kann ein NQC-Programm auf dem PC mit dem Bricx Command Center erstellt werden. Nach der Erstellung des NQC-Programms, natürlich im NQC-Code, übersetzt ein Compiler auf dem PC (Host Computer) diesen Code in einen Bytecode. Dieser Bytecode wird dann zum RCX übertragen und dort als Anwenderprogramm gespeichert. Der Mikrocontroller im RCX kann den Bytecode nicht direkt ausführen, dafür gibt es nämlich die Firmware. Die Firmware hat die Aufgabe den Bytecode zu interpretieren, wenn das Anwenderprogramm ausgeführt wird. Genauer gesagt, erzeugt die Firmware für den Mikrocontroller einen Maschinencode; in diesem Fall einen H8-Maschinencode. Damit der Mikrocontroller weiß, wie er auf die Hardware zugreifen kann, sind im System-ROM die Hardware-Routinen für die Steuerung des RCX hinterlegt worden. Sie steuern die Ein- bzw. Ausgänge und auch den IR-Empfänger. Das bedeutet, dass in den Hardware-Routinen festgelegt ist, wie die Firmware vom IR-Empfänger ausgelesen und im RAM gespeichert wird. Außerdem ist dort festgelegt worden, wie der RCX die Motoren ansteuert, und wie er auf die Sensoren zugreifen kann.

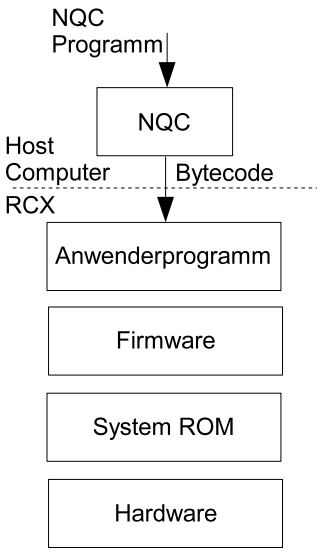


Abbildung 1: RCX Architektur

# 2 Bricx Command Center

Im folgenden wird die Installation und die Bedienung der Entwicklungsumgebung Bricx Command Center noch näher erläutert. Wobei der Abschnitt über die Installation nicht relevant für die Durchführung der Versuche im Labor „Mobile Roboter“ ist.

## 2.1 Installation

Als erstes muss das Bricx Command Center von der Internetseite http://hometown.aol.com/johnbinder/brixcc.htm heruntergeladen werden. Der aktuelle Stand (Februar 2004) ist die Version Bricx Command Center 3.3.7.7 - auf diese Version bezieht sich diese Ausarbeitung. Von der angegebenen Internetseite wird die bricxcc_setup_3377.exe oder eine aktuellere heruntergeladen und in ein beliebiges Verzeichnis gespeichert. Danach erfolgt die eigentliche Installation des Bricx Command Center durch Ausführen der Datei bricxcc_setup_3377.exe. Bei der Installationsroutine des Bricx Command Center handelt es sich um eine menügeführte Installation, deshalb sind die Anweisungen am besten einfach zu befolgen. Bei der Installation wird das Verzeichnis c:\Programme\Bricxcc vorgeschlagen, und auf diese Verzeichnisstruktur bezieht sich die folgende Erklärung.
Nach erfolgreicher Installation des Bricx Command Center fehlt zur einwandfreien Programmierung bzw. Steu-erung des RCX die LEGO-Firmware. Diese kann von der LEGO Mindstorms Internetseite, die auch weitere nützliche Links beinhaltet, http://mindstorms.lego.com/eng/community/resources/default.asp heruntergeladen werden. Um die aktuelle Firmware von der LEGO Mindstorm Internetseite herunterzuladen, muss dazu als erstes die zip-Datei LEGOMinstormsSDK2.5.zip oder eine aktuellere heruntergeladen werden. Danach wird die zip-Datei mit Hilfe eines Archivierungsprogramms geöffnet und die Datei firmXXXX.lgo entpackt. Diese Datei wird am besten in das Unterverzeichnis Bricxcc\Firmware (siehe oben) kopiert.

## 2.2 Starten

Durch Doppelklick auf das Desktop Symbol vom Bricx Command Center wird jenes gestartet. Danach erscheint das Auswahlfenster für den Brick und die Schnittstelle, an der der IR-Tower angeschlossen ist. Dabei ist sicher zustellen, dass der RCX sich gegenüber dem IR-Tower befindet. Wenn die Firmware firm0328.lgo verwendet werden soll, sollte die Einstellung RCX2 gewählt werden. Nach dieser Auswahl erscheint das Command Center Fenster und das Template Fenster, das eine Übersicht der verfügbaren Befehle in NQC anbietet, wenn der RCX gefunden wurde. Sollte jedoch das Bricx Command Center ein Problem bei der Kommunikation mit dem RCX haben, gibt es eine Fehlermeldung heraus. Nach dem wegklicken dieser Fehlermeldung, wird das Bricx Command Center geladen; jedoch mit geringerem Funktionsumfang. Dabei werden die Funktionen, die eine Kommunikation mit dem RCX erfordern, deaktiviert. Diese Beeinträchtigung kann durch eine erneute erfolgreiche suche nach dem RCX behoben werden. Bevor das erste Programm zum RCX übertragen und gestartet werden kann, muss vorher einmal die Firmware zum RCX übertragen werden. Dies erfolgt mit dem Menübefehl **Tools → Download Firmware**. Danach ist der RCX betriebsbereit.

## 2.3 Entwicklungszyklus

Mit dem Bricx Command Center ist es möglich, den RCX zu programmieren. Dazu wird ein Programm erstellt, das aus einzelnen Befehlen besteht, die in einem Textfenster editiert werden. Das Programm wird dann übersetzt und auf den RCX übertragen, bevor es gestartet werden kann. Dieser Entwicklungszyklus wird vom

Bricx Command Center durch entsprechende Werkzeuge unterstützt. Diese Werkzeuge können durch Anklicken des entsprechenden Symbols oder durch ausführen des entsprechenden Menübefehls aufgerufen werden. Im folgenden werden die einzelnen Menübefehle vorgestellt, um den Entwicklungszyklus durchzuführen. Die dazu entsprechenden Symbole werden in der folgenden Abbildung gezeigt.

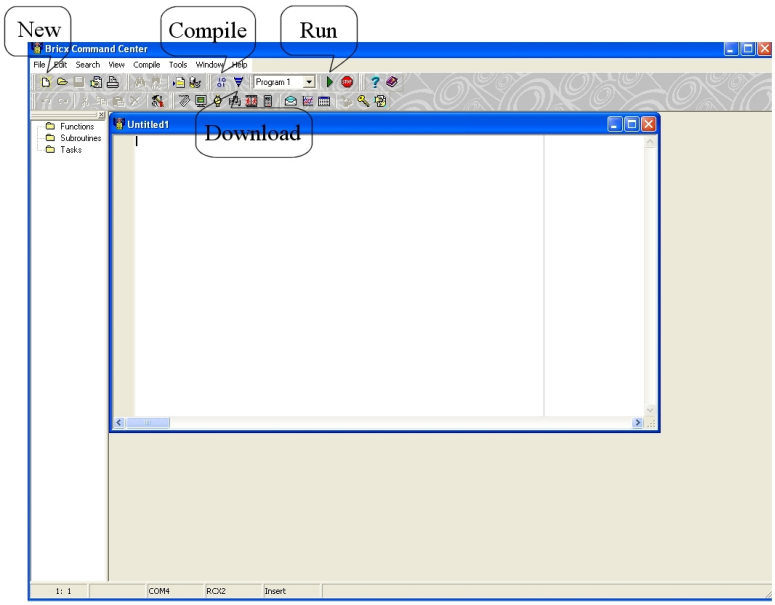


Abbildung 2: Unterstützende Werkzeuge für den Entwicklungszyklus

Um ein Programm neu zu erstellen wird der Menübefehl **File → New** ausgeführt. Danach kann im geöffneten Programmfenster der Text eingegeben werden. Nachdem die Eingabe beendet ist, muss das Programm mit dem Menübefehl **Compile → Download** kompiliert und danach mit dem Menübefehl **Compile → Download** zum RCX übertragen werden. Wenn beim Kompilieren Fehler auftreten, werden diese durch eine entsprechende Fehlermeldung angezeigt. Durch klicken auf die Fehlermeldung werden die fehlerhaften Programmzeilen angezeigt. Sollten beim Kompilieren und übertragen keine Fehler auftreten, wird dies durch ein Signal vom RCX bestätigt. Das Programm kann auch durch das Bricx Command Center durch den Menübefehl **Compile → Run** gestartet werden. Das Übertragen und das Starten des Programms könnte auch durch den Menübefehl **Compile → Download and Run** durchgeführt werden.

# 3 Einführung in NQC

Bei NQC handelt es sich um eine textbasierte Programmiersprache und zudem besitzt dieses Softwarepaket für den RCX einen C-Compiler. Aufgrund des C-Compilers ergeben sich für NQC ähnliche Kontrollstrukturen, sowie einen ähnlichen Präprozessor, wie bei der Programmiersprache C. Aber die Beschränkungen, die sich bei NQC ergeben, sind auf die LEGO Standard-Firmware zurückzuführen. In den folgenden Abschnitten erfolgt die Einführung in die Programmiersprache NQC. Für eine ausführlichere Beschreibung der Programmiersprache NQC wird auf den „NQC Programmer's Guide" von Dave Baum verwiesen, auf dessen diese Einführung in NQC basiert. Eine deutsche Übersetzung dieser Programmieranleitung ist z. B. unter der Internetadresse http://lug.mfh-iserlohn.de/lego/ zu finden. Des Weiteren findet auch in diversen Büchern über den LEGO Mindstorms eine Einführung in die Programmiersprache NQC statt. Auch die Programmierumgebung Bricx Command Center bietet den Anwender eine Beschreibung der Befehle - allerdings in Englisch – an. [NQC-03]

## 3.1 Lexikalische Strukturen

Zu den Lexikalischen Strukturen gehört die Schreibweise von Kommentaren, sowie die Behandlung von Leerzeichen und das Einsetzen von Numerischen Konstanten. Des Weiteren werden auch hier die gültigen Zeichen für Namen festgelegt.

Es wird zwischen zwei verschiedenen Arten von Kommentaren unterschieden. Zum einen sind das die einzeiligen Kommentare, die mit einem doppelten Schrägstrich **//** beginnen, und zum anderen gibt es die mehrzeiligen Kommentare, die mit **/*** beginnen und mit ***/** enden.

```
//Dies ist ein einzeiliger Kommentar
/* Und dies ist ein
   mehrzeiliger Kommentar */
```

Das Einsetzen von Leerzeichen ist nicht unbedingt notwendig, aber es erleichtert das Lesen des Programmcodes ungemein.

```
x=8;        //ohne Leerzeichen
x = 8;      //mit Leerzeichen, deshalb besser zu lesen
```

Die Numerischen Konstanten werden normalerweise vom Compiler als Dezimalwert interpretiert, es sei denn, dass **0x** vorangestellt wird. In diesem Fall wird die Konstante als Hexadezimalwert vom Compiler interpretiert.

```
x = 10;     // setzt x auf dem Dezimalwert 10
x = 0xb;    // setzt x auf dem Dezimalwert 11
```

Namen werden z. B. für Variablen, Tasks und Funktionen verwendet. Dabei ist darauf zu achten, dass die Namen mit einem Buchstaben oder einem Unterstrich **_** beginnen. Des Weiteren muss beachtet werden, dass auch die Programmiersprache NQC reservierte Namen, die so genannten Schlüsselwörter, besitzt. Eine Liste der Schlüsselwörter von NQC ist im Abschnitt „5.6 Schlüsselwörter" zu finden.

## 3.2 Präprozessor

Die wichtigsten Anweisungen des Präprozessors sind die **#define**- und die **#include**-Anweisung. Obwohl der Präprozessor von NQC fast genau wie der Standard C-Präprozessor arbeitet, gibt es jedoch einige Abweichungen bei diesen Anweisungen. Diese Abweichungen machen sich bei der **#include**-Anweisung dadurch bemerkbar, dass der Dateiname in doppelte Anführungszeichen eingeschlossen wird. Des Weiteren gibt es

bei dieser Anweisung keinen include-Pfad für Systemdateien und somit entfällt die Verwendung von spitzen Klammern, die den Dateinamen umschliessen.

```
#include „turn.nqh“     //Erlaubt
#include <turn.nqh>     //Verboten (Fehler)
```

Die #define-Anweisung wird für einfache Makroersetzungen eingesetzt. Bei einer wiederholten Definition wird im Gegensatz zu C ein Fehler erzeugt. Im normalen Fall endet das Makro am Zeilenende, aber durch den Einsatz eines Backslash wird ein mehrzeiliges Makro erzeugt.

```
#define L_Motor   OUT_A             //einzeiliges Makro
#define turn      while(SENNSOR_1 != 1);\
                  Off(OUT_A);       //mehrzeiliges Makro
```

## 3.3 Programmstruktur

Wie jedes andere Programm, besteht auch ein gutes NQC-Programm aus mehreren unterschiedlichen Blöcken. Dabei wird zwischen den drei folgenden verschiedenen Blöcken unterschieden:

- Task
- Inline-Funktion
- Unterprogramm

Jedes NQC-Programm muss mindestens aus einer Task mit dem Namen „main“ bestehen. Diese Task wird automatisch beim Programmstart aktiviert. Zusätzlich zu dieser einen Task kann ein NQC-Programm aus neun weiteren Tasks bestehen. Sie haben die gleiche Syntax wie die Task main, müssen jedoch mit einem eindeutigen Namen ausgezeichnet sein. Außerdem müssen sie vom Anwender mit Hilfe der Anweisung **start** bzw. **stop** gestartet bzw. gestoppt werden. Der Rumpf einer Task besteht aus einer Abfolge von Anweisungen.

```
task Name()
{
  //Rumpf der Task
}
```

Um in einem Programm alle Tasks zu stoppen und somit das Programm vollständig zu beenden, gibt es für diesen Zweck den Befehl **StopAllTasks**.

```
StopAllTasks();         //Beendet alle laufenden Tasks
```

Bei der Erstellung eines Programms mit mehreren Tasks sollte bedacht werden, das mehrere Tasks gleichzeitig aktiv sein können. In diesem Fall schaltet der RCX zwischen den einzelnen Tasks immer hin und her und führt dabei nur einen Teil der Anweisungen der einzelnen Tasks aus. Dadurch entsteht eine simultane Ausführung des Programms. Dies sollte besonders bei der Fehlersuche berücksichtigt werden.

Oft ist es hilfreich die Übersichtlichkeit des Programms zu bewahren, indem einzelne Anweisungen in einer Funktion zusammen gefasst werden, um sie bei Bedarf aufzurufen. Dies ist besonders vorteilhaft, wenn ein bestimmter Anweisungsteil mehrmals ausgeführt werden muss. Dabei ist zu beachten das NQC Funktionen mit Argumenten kennt, aber keine Rückgabewerte.

```
void Name (Argumentenliste)
{
  //Rumpf der Funktion
}
```

Die Argumentenliste kann aus keinem oder aus mehreren Argumenten getrennt durch ein Komma bestehen. Ein Argument wird durch seine Typangabe gefolgt von dem Namen definiert. Von NQC werden vier verschiedene Typen unterstützt, die in folgender Tabelle kurz beschrieben sind.

| ***Typ*** | ***Semantik*** | ***Einschränkung*** |
|---|---|---|
| int | Wertübergabe | keine |
| int & | Verweis (Adressübergabe) | nur Variablen können als Argument verwendet werden |
| const int | Wertübergabe | nur Konstanten können als Argument verwendet werden |
| const int & | Verweis (Adressübergabe) | Funktion kann den Wert des Arguments nicht verändern |

Tabelle 1: Argumenttypen

Zusätzlich kann ein Programm noch bis zu acht Unterprogrammen, den so genannten Subroutinen, beinhalten. Eine Subroutine muss immer von einer Task aufgerufen werden, damit sie ausgeführt werden kann. Sie kann weder sich selbst noch eine andere aufrufen. Eine weitere Einschränkung ist, dass keine Werte übergeben werden können und dass die Subroutine auch keine Werte zurückgibt. In der Zeit, in der die Subroutine ausgeführt wird, muss die Task solange warten, bis die Subroutine ihre Anweisungsliste abgearbeitet hat. Erst danach kann die Task mit der Bearbeitung ihrer eigenen Anweisungen fortfahren. Zudem sollte darauf geachtet werden, dass beim RCX 1.0 keine lokalen Variablen verwendet werden dürfen, wenn mehrere Tasks die Subroutine aufrufen.

```
sub Name()
{
  //Rumpf  des Unterprogramms
}
```

## 3.4 Anweisungen

Die einzelnen Blöcke, z. B. Task, bestehen aus einzelnen Anweisungen. Diese werden immer mit einem Semikolon abgeschlossen.

### 3.4.1 Variablen und Zuweisungen

Der RCX 2.0, der auf die Firmware firm0328.log basiert, stellt dem Anwender insgesamt 48 Variablen zur Verfügung. Diese Variablen teilen sich in 32 globale und 16 lokale Variablen auf. Im Gegensatz dazu stellt der RCX 1.0, der auf die Firmware firm0308.lgo basiert, dem Anwender nur 32 (globale) Variablen zur Verfügung. Globale Variablen sind im gesamten Programm gültig und werden immer außerhalb von Blöcken, z. B. Funk-tionen, Tasks, usw. definiert. Im Gegensatz dazu sind lokale Variablen nur in dem Bereich gültig, in dem sie auch definiert wurden. Lokale Variablen sollten immer am Anfang eines Blocks definiert werden. Damit eine Variable erst einmal verwendet werden kann, muss sie vorher erst mit ihrem Namen deklariert werden. Der RCX unterstützt nur Integer-Variablen, die durch das Schlüsselwort **int** gekennzeichnet werden. Eine Variablendeklaration geschieht nach folgendem Schema: Als erstes kommt das Schlüsselwort **int** gefolgt vom Variablennamen und zuletzt kann optional noch eine Zuweisung erfolgen. Bei Erzeugung einer Liste von Variablen werden die einzelnen Variablennamen durch ein Komma getrennt.

```
int Name;
int Name1, Name2;
int Name3 = 1;
```

Wenn eine Variable einmal definiert worden ist, kann ihr jederzeit über einen Zuweisungsoperator ein neuer Wert zugewiesen werden. Dabei wird zwischen neun verschiedenen Operatoren unterschieden, wobei **=** der einfachste Zuweisungsoperator ist. Eine Auflistung der Zuweisungsoperatoren inklusive der Operatoren, die den Wert der Variable auf unterschiedliche Weise verändern, ist im Anhang unter „5.3 Ausdrücke“ zu finden.

### 3.4.2 Kontrollstrukturen

Kontrollstrukturen kommen immer dann zum Einsatz wenn die sequentielle Abarbeitungsfolge der Anweisungen geändert werden muss. Dies kann durch Schleifen oder durch bedingte bzw. nicht bedingte Anweisungen erreicht werden. In den meisten Fällen wird dazu eine Bedingung verwendet, die stets in Klammern eingeschlossen wird.

Bei der if-Anweisung wird der Ausdruck bewertet. Wenn dieser erfüllt ist, also ungleich Null ist, werden die Anweisungen ausgeführt, ansonsten werden sie einfach übersprungen.

```
if(Ausdruck) Anweisung
```

Die if-Anweisung kann durch eine else-Klausel erweitert werden, die dann ausgeführt wird, wenn der Ausdruck falsch ist.

```
if(Ausdruck) Anweisung1
else Anweisung2
```

Die if-else-Anweisungen können natürlich auch verschachtelt werden. Des Weiteren gibt es auch wie in der Programmiersprache C die else-if-Anweisung, die es erlaubt, mehrere Ausdrücke hintereinander abzufragen.

```
if(Ausdruck1) Anweisung1
else if(Ausdruck2) Anweisung2
else Anweisung3
```

Bei der while-Schleife werden die Anweisungen solange ausgeführt, wie auch der Kontrollausdruck wahr ist, also ungleich Null. Da der Ausdruck schon vor dem ersten Durchlaufen der Schleife überprüft wird, kann es passieren, dass die Anweisungen erst gar nicht ausgeführt werden, wenn der Kontrollausdruck bereits da schon gleich Null ist.

```
while(Ausdruck) Anweisung
```

Bei der do-while-Schleife werden die Anweisungen solange ausgeführt bis der Ausdruck gleich Null ist. Der Unterschied zur while-Schleife besteht darin, dass die Schleife mindestens einmal durchlaufen wird, da der Ausdruck erst nachdem ersten Durchlaufen der Schleife überprüft wird.

```
do Anweisung while(Ausdruck)
```

Die for-Schleife eignet sich besonders gut für Zählvorgänge. Dabei ist der erste Ausdruck der Initialisierungsausdruck und der zweite legt das Abbruchkriterium fest. Das bedeutet, dass die Schleife solange durchlaufen wird, wie auch der zweite Ausdruck wahr ist, während der dritte Ausdruck bei jedem Schleifendurchlauf erneut bewertet wird. Normalerweise ist der dritte Ausdruck eine Variable, die inkrementiert bzw. dekrementiert wird.

```
for (Ausdruck1; Ausdruck2; Ausdruck3) Anweisung
```

Als letztes wird an dieser Stelle die repeat-Anweisung vorgestellt, die entsprechend einer bestimmten Anzahl die Anweisungen wiederholt ausführt.

```
repeat(Anzahl) Anweisung
```

## 3.5 Ausdrücke und Bedingungen

Ab der Version 2.3 und auch der hier verwendeten, wird bei NQC nicht mehr zwischen Ausdrücken und Bedingungen unterschieden. Das bedeutet für den Anwender, dass ihm jetzt auch Vergleichsoperatoren bei Ausdrücken zur Verfügung stehen. Die einfachste Form eines Ausdrucks ist ein Wert, der auch bei Bedarf mit unterschiedlichen Operatoren zu komplexeren Ausdrücken kombiniert werden kann. Wobei ein Wert bei NQC eine numerische Konstante, eine Variable oder ein Sensorwert sein kann. Zudem sind die zwei zusätzlichen Werte **true** (ungleich Null) und **false** (gleich Null) vordefiniert. Eine Liste der zur Verfügung stehen Operatoren ist dem Abschnitt „5.3 Ausdrücke“ zu finden.

Bedingungen kommen immer dann zum Einsatz, wenn Entscheidungen zu treffen sind. Sie bestehen in den meisten Fällen aus einem Vergleich zwischen zwei Ausdrücken. Eine Auflistung der Bedingungen ist im Abschnitt „5.2 Bedingungen“ zu finden.

## 3.6 RCX-API

Im RCX-API sind Konstanten, Werte und Funktionen, die auch zur Steuerung der Aus- und Eingänge des RCX notwendig sind, festgelegt. Die Befehle zur Steuerung der Aus- und Eingänge werden in den folgenden Abschnitten vorgestellt. Wobei sich diese Abschnitte auf die Firmware firm0328.lgo beziehen. Dem Abschnitt „5.4 RCX-Funktionen“ sind weitere RCX spezifische Befehle mit kurzer Beschreibung beigefügt.

### 3.6.1 Ausgangsbefehle

Der RCX besitzt drei Ausgänge, die als **OUT_A**, **OUT_B** und **OUT_C** bezeichnet werden. Um mehrere Ausgänge in einem Befehl anzugeben, werden ihre Namem einfach mit dem Operator **+** verbunden. Zur Steuerung der Ausgänge stehen dem Anwender die drei folgenden unterschiedlichen Optionen zur Verfügung:

- Modus
- Richtung
- Leistung

Der Modus legt fest, in welchem Betriebszustand sich der Ausgang befindet, wobei ein Ausgang drei verschiedene Betriebszustände annehmen kann. Die einfachsten Betriebszustände sind der ein- und ausgeschaltete Ausgang. Diese können durch den Befehl **On(Ausgänge)** und **Off(Ausgänge)** erreicht werden. Ein weiterer Betriebszustand ist der Leerlauf, der durch den Befehl **Float(Ausgänge)** erreicht wird. Der wichtigste Unterschied zwischen den Befehl Off und Float besteht darin, dass bei dem Befehl Off die Motorwelle nur sehr schwer zu drehen ist. Während bei dem Befehl Float die Motorwelle frei zu drehen ist.

```
On(OUT_B+OUT_C);  //Schaltet die Ausgänge B und C ein
Off(OUT_B);       //Schaltet Ausgang B aus
Float(OUT_C);     //Schaltet Ausgang C in den Leerlauf
```

Wenn sich der Ausgang in dem Betriebszustand „eingeschaltet“ befindet, kann er die zwei Bewegungsrichtungen Vor- oder Rückwärts annehmen. Um diese Bewegungsrichtungen einzustellen, gibt es die drei Befehle **Fwd(Ausgänge)**, **Rev(Ausgänge)** und **Toggle(Ausgänge)**.

```
Fwd(OUT_B);       //Setzt für Ausgang B eine Vorwärtsbewegung
Rev(OUT_B);       //Setzt für Ausgang B eine Rückwärtsbewegung
Toggle(OUT_B);    //Wechselt die Bewegungsrichtung von Ausgang B
```

Für das Einschalten der Motoren, sowie die Festlegung der Bewegungsrichtung, gibt es Befehle in NQC, die beides auf einmal bewerkstelligen.

```
OnFwd(Ausgänge); //Schaltet die angegebenen Ausgänge für eine Vorwärtsbewegung
                    ein
OnRev(Ausgänge); //Schaltet die angegebenen Ausgänge für eine Rückwärtsbewegung
                    ein
```

Zusätzlich können die Ausgänge nur für eine bestimmte Zeitdauer mit dem Befehl **OnFor(Ausgänge, Dauer)** eingeschaltet werden. Dabei wird die Zeitdauer in einer Auflösung von 10ms angegeben.

```
OnFor(OUT_B, 200);      //Schaltet den Ausgang B für 2 Sekunden ein
```

Des Weiteren können, wie oben erwähnt, die Leistungsstufen der einzelnen Ausgänge vom RCX geändert werden. Dazu steht dem Anwender der Befehl **SetPower(Ausgänge, Leistungsstufe)** zur Verfügung. Die Leistungsstufe ist eine ganze Zahl zwischen 0 und 7, wobei 0 die niedrigste und 7 die höchste Leistungsstufe ist.

```
SetPower(OUT_B+OUT_C, 2); //Setzt die Ausgänge B und C auf die Leistungsstufe 2
```

### 3.6.2 Eingangsbefehle

Zudem besitzt der RCX noch drei Eingänge, die so genannten Sensoreingänge. Diese Eingänge besitzen die Bezeichnung **SENSOR_1**, **SENSOR_2** und **SENSOR_3**. Aufgrund dessen, dass am RCX der Anschluss von verschiedenen Sensoren möglich ist, ist deren richtige Konfiguration besonders wichtig. Für die richtige Sensorkonfiguration muss der Typ und der Modus konfiguriert werden. Zur Erleichterung der Konfiguration sind in NQC bereits einige Typ-/Moduskombinationen festgelegt. In der folgenden Tabelle werden die Kombinationen für den Berührungs-, Licht- und Rotationssensor aufgelistet.

| ***Sensorkonfiguration*** | ***Typ*** | ***Modus*** |
|---|---|---|
| SENSOR_TOUCH | SENSOR_TYPE_TOUCH | SENSOR_MODE_BOOL |
| SENSOR_PULSE | SENSOR_TYPE_TOUCH | SENSOR_MODE_PULSE |
| SENSOR_EDGE | SENSOR_TYPE_TOUCH | SENSOR_MODE_EDGE |
| SENSOR_LIGHT | SENSOR_TYPE_LIGHT | SENSOR_MODE_PERCENT |
| SENSOR_ROTATION | SENSOR_TYPE_ROTATION | SENSOR_MODE_ROTATION |

Tabelle 2: Sensorkonfiguration

Um die voreingestellten Sensorkonfigurationen zu nutzen gibt, es den Befehl **SetSensor(Sensoranschluss, Sensorkonfiguration)**.

```
SetSensor(SENSOR_1, SENSOR_LIGHT); //Sensor 1 wird als Lichtsensor konfiguriert
```

Natürlich können Sensortyp und -modus auch einzeln konfiguriert werden, dies kann manchmal ziemlich nützlich sein. Die Konfiguration von Sensortyp und -modus wird durch die Befehle **SetSensorType(Sensoranschluss, Sensortyp)** und **SetSensorMode(Sensoranschluss, Sensormodus)** vorgenommen.

| ***Sensortyp*** | ***Sensor*** |
|---|---|
| SENSOR_TYPE_TOUCH | Berührungssensor |
| SENSOR_TYPE_LIGHT | Lichtsensor |
| SENSOR_TYPE_ROTATION | Rotationssensor |

Tabelle 3: Sensortyp

| *Sensormodus* | *Beschreibung* |
|---|---|
| SENSOR_MODE_RAW | Rohwert von 0 bis 1023 |
| SENSOR_MODE_BOOL | Logischer Wert, entweder 0 oder 1 |
| SENSOR_MODE_EDGE | Anzahl der Impulsflanken, also zählt die Übergänge von 0 auf 1 und umgekehrt |
| SENSOR_MODE_PULSE | Anzahl der Impulse, also zählt die Übergänge von 0 auf 1 |
| SENSOR_MODE_PERCENT | Prozentzahl von 0 bis 100 |
| SENSOR_MODE_ROTATION | Anzahl der Achsenumdrehungen, pro voller Umdrehung 16 Impulse |

Tabelle 4: Sensormodi

```
SetSensorType(SENSOR_1, SENSOR_TYPE_LIGHT); //Sensor 1 ist ein Lichtsensor
SetSensorMode(SENSOR_1, SENSOR_MODE_RAW);  //Messwert wird als Rohwert angegeben
```

Je nach eingestellten Sensormodi kann der Sensor durch den Befehl **ClearSensor(Sensoranschluss)** zurückgesetzt werden. Dies ist immer dann der Fall, wenn relative Größen wie bei der Rotation, Flanken- und Impulszählung gemessen werden.

```
SetSensor(SENSOR_2, SENSOR_ROTATION);     //Sensor 2 ist ein Rotationssensor
ClearSensor(SENSOR_2);  //Setzt den Wert des Rotationssensors auf Null zurück
```

### 3.6.3 Klänge

Der RCX kann über seinen Lautsprecher sechs verschiedene Klänge und diverse Töne, die über die Frequenz und Zeitdauer bestimmt werden, abspielen. Mit dem Befehl **PlayTone(Frequenz, Dauer)** wird ein Einzelton mit einer bestimmten Frequenz und Dauer erzeugt. Die Dauer des Tons wird in einer Auflösung von 10ms angegeben.

```
PlayTone(800, 100);     //Spielt einen Ton für eine Sekunde
```

Die Klänge, die im RCX integriert sind, werden mit dem Befehl **PlaySound(Name)** ausgegeben. In der folgenden Tabelle sind die sechs verfügbaren Klänge für den RCX aufgelistet.

| *Name* | *Beschreibung* |
|---|---|
| SOUND_CLICK | Kurzes Klicken |
| SOUND_LOW_BEEP | Tiefer Piepton |
| SOUND_DOUBLE_BEEP | Zwei Pieptöne |
| SOUND_DOWN | Absteigendes Arpeggio |
| SOUND_UP | Aufsteigendes Arpeggio |
| SOUND_FAST_UP | Schnell ansteigendes Arpeggio |

Tabelle 5: Klänge des RCX

```
PlaySound(SOUND_CLICK); //Spielt ein kurzes Klicken
```

### 3.6.4 Timer

Der RCX ist mit vier Timern ausgestattet, die von 0 bis 3 durchnummeriert sind. Sie besitzen standardmäßig eine Auflösung von 100 ms. Wenn eine höhere Auflösung benötigt wird, kann dies durch den Befehl **FastTimer(Timer_Nr)** erreicht werden. In diesem Fall besitzt der angegebene Timer eine Auflösung von 10ms. Zur Steuerung der Timer stellt NQC zwei Befehle zur Verfügung. Mit dem Befehl **Timer(Timer_Nr)** kann der aktuelle Stand des angegebenen Timers abgefragt werden. Mit dem zweiten Befehl **ClearTimer (Timer_Nr)** kann der angegebene Timer auf Null zurückgesetzt werden. Im Gegensatz zu anderen Program-

miersprachen für Mikrocontroller gibt es bei NQC keine direkten Befehle zum starten bzw. stoppen der Timer.

```
Timer(1);         //Gibt den aktuellen Wert von Timer 1 an
ClearTimer(1);    //Setzt Timer 1 auf Null zurück
```

### 3.6.5 Zähler

Mit Einzug der Firmware 0328.lgo stellt der RCX-API dem Anwender zudem noch drei Zähler, die von 0 bis 2 durchnummeriert sind, zur Verfügung. Der Zähler ist praktisch gesehen eine Variable, die inkrementiert, dekrementiert und gelöscht werden kann. Bei der Verwendung eines Zählers sollte immer bedacht werden, dass sich die Speicherplätze von den Zählern mit den Speicherplätzen der globalen Variablen von 0 bis 2 überlappen. Aufgrund dieser Eigenschaft, muss vorher eine Speicherplatzreservierung für die Zähler, die verwendet werden sollen, durch den Befehl **#pragma reverse Anfangsadresse Endadresse** erfolgen. Wenn aber nur ein Speicherplatz reserviert werden soll, entfällt die Angabe der Endadresse.

```
#pragma reserve 0 2     //Reserviert die Speicherplätze 0 bis 2
#pragma reserve 0       //Reserviert den Speicherplatz 0
```

Das Inkrementieren bzw. Dekrementieren des ausgewählten Zählers erfolgt durch den Befehl **IncCounter (Zähler_Nr)** bzw. **DecCounter(Zähler_Nr)**.

```
IncCounter(0);          //Erhöht den Zähler 0 um 1
DecCounter(0);          //Mindert den Zähler 0 um 1
```

Der aktuelle Zählwert eines ausgewählten Zählers kann durch den Befehl **Counter(Zähler_Nr)** ermittelt werden.

```
Counter(0);       //Gibt den aktuellen Zählwert von Zähler 0 an
```

Mit dem Befehl **ClearCounter(Zähler_Nr)** wird der angegebene Zähler auf Null zurückgesetzt.

```
ClearCounter(0);  //Setzt den Zähler 0 auf Null zurück
```

Der Zähler kann natürlich auch - wie es z. B. in der Programmiersprache C üblich ist - durch eine Zählvariable ersetzt werden.

### 3.6.6 Anzeige

Die Anzeige des RCX kann bis zu einem gewissen Grad beeinflusst werden. Dazu stellt NQC dem Anwender drei Befehle zur Verfügung. Der erste Befehl **SelectDisplay(Modus)** stellt den aktuellen Wert der angegebenen Datenquelle, die durch den Modus ausgewählt wird, auf dem Display dar.

| ***Modus*** | ***Beschreibung*** |
|---|---|
| DISPLAY_WATCH | Systemuhr |
| DISPLAY_SENSOR_1 | Eingang 1 |
| DISPLAY_SENSOR_2 | Eingang 2 |
| DISPLAY_SENSOR_3 | Eingang 3 |
| DISPLAY_OUT_A | Ausgang A |
| DISPLAY_OUT_B | Ausgang B |
| DISPLAY_OUT_C | Ausgang C |

Tabelle 6: Modi für die Anzeige

```
SelectDisplay(DISPLAY_SENSOR_1); /*Zeigt die aktuellen Werte von Sensor_1 im
                                   Display an*/
```

Der nächste Befehl **SetUserDisplay(Wert, Genauigkeit)** eignet sich noch besser um einen Fehler im Programm zu suchen bzw. bestimmte Werte zu kontrollieren als der Erste, da durch diesen Befehl die Möglichkeit besteht, die ganzzahligen Werte von Variablen auf dem Display anzuzeigen. Dabei gibt die Genauigkeit die Stellen rechts vom Dezimalpunkt an. Wenn kein Dezimalpunkt erwünscht ist, wird für die Genauigkeit der Wert Null angegeben.

```
int v = 3108;           //Der Integer-Variable v wird der Wert 3108 zugewiesen
SetUserDisplay(v, 0);   //Auf dem Display erscheint der Wert: 3108
SetUserDisplay(v, 2);   //Auf dem Display erscheint der Wert: 31.08
```

Mit dem letzten Befehl **SetWatch(Stunden, Minuten)** kann die Systemuhr konfiguriert werden. Dabei muss die Konstante für die Stunden eine Zahl zwischen 0 und 23 und für die Minuten eine Zahl zwischen 0 und 59 sein.

```
SetWatch(8, 30);  //Setzt die Systemuhr auf 8:30
```

### 3.6.7 Datalog

Der Datalog bietet eine weitere Möglichkeit Fehler im Programm zu suchen, bzw. Werte zu kontrollieren. Er besteht aus einer Liste mit ganzzahligen Werten, die zum PC mit Hilfe des Bricx Command Center übertragen werden können. Kurz gesprochen: ein Datalog ist ein Datenspeicher für Werte. Bevor ein Datalog benutzt wer-den kann, muss ihm vorher erst ein Platz für die abzuspeichernden Elementen auf dem RCX zugewiesen werden. Dies geschieht mit dem Befehl **CreateDatalog(Größe)**, wobei die Größe die Anzahl der abzuspeichernden Elementen angibt.

```
CreateDatalog(30);      //Es wird ein Datalog für 30 Werte erzeugt
```

Zum Löschen des Datalogs wird für die Größe der Wert 0 angegeben. Bei der Arbeit mit dem Datalog sollte immer bedacht werden, dass es nur einen Datalog gibt. Dadurch kann ein bereits bestehender Datalog durch ausführen des Befehls CreateDatalog gelöscht werden.

```
CreateDatalog(0); //Bestehender Datalog wird gelöscht
```

Durch den Befehl **AddToDatalog(Wert)** wird ein neuer Wert dem Datalog hinzugefügt. Dabei sollte darauf geachtet werden, dass nur soviele Werte in den Datalog geschrieben werden, wie auch vorher Platz reserviert wurde. Wenn nämlich der Datalog voll ist, gehen die neuen Werte verloren. Auf der rechten Seite des Displays vom RCX wird mit den Viertelkreisen der Zustand des Datalogs angezeigt.

```
v = 5;
AddToDatlog( v);  //Fügt den Datalog den Wert 5 zu
```

### 3.6.8 IR-Kommunikation

Der RCX ist in der Lage über seine IR-Schnittstelle Nachrichten zu empfangen und zu senden. Eine Nachricht kann aus einer ganzzahligen Zahl zwischen 0 und 255 bestehen, wobei die Verwendung der Zahl 0 nicht empfohlen wird. Die Programmiersprache NQC bietet für die Arbeit mit der IR-Schnittstelle drei Befehle an: Einmal zum Senden von Nachrichten, des Weiteren zum Lesen von Nachrichten und das Löschen der jeweils eingegangen Nachricht, nach dem Beantworten. Das Senden einer Nachricht wird mit dem Befehl **SendMessage(Wert)** ausgeführt und das Lesen der zuletzt eingegangen Nachricht mit dem Befehl **Message ()**. Das Löschen einer eingegangen Nachricht bzw. des Nachrichtenpuffers kann mit dem Befehl **ClearMessage()** erfolgen. Dies kann besonders nützlich sein, wenn auf die nächste eingehende Nachricht gewartet wird.

```
SendMessage(6);   //Sendet den Wert 6 als Nachricht
```

In einigen Situationen ist es notwendig die Sendeleistung der IR-Schnittstelle vom RCX durch den Befehl **SetTxPower(Leistung)** zu verändern. Für niedrige Sendeleistung wird die Konstante TX_POWER_LO verwendet und für hohe TX_POWER_HIGH.

```
SetTxPower(TX_POWER_HIGH); /*IR-Schnittstelle wird auf hohe Sendeleistung konfi-
                              guriert*/
```

### 3.6.9 Verschiedene Befehle

Ein sehr nützlicher Befehl für die Programmierung eines Programms ist der Befehl **Wait(Zeitdauer)**. Beim Ausführen dieses Befehls setzt die Task für die angegebene Zeitdauer aus, wobei die Zeitdauer eine Konstante oder ein Ausdruck sein kann. Die Zeitdauer wird dabei in einer Auflösung von 10ms angegeben.

```
x = 100;
Wait(x);    //Wartet 1 Sekunde
Wait(200);  //Wartet 2 Sekunden
```

Ein weiterer interessanter Befehl ist der Befehl **Random(n)** zur Erzeugung einer Zufallszahl. Dabei wird eine Zufallszahl erzeugt, die zwischen 0 und n liegt.

```
x = Random(10);   //Weist der Variable x einen Wert zwischen 0 und 10 zu
```

# 4 Subsumtionsarchitektur

Die Subsumtionsarchitektur wurde in den späten achtziger Jahren von Professor Rodney Brooks und der Arbeitsgruppe für Mobile Roboter am MIT Artificial Intelligence Laboratory entwickelt. Im Gegensatz zum traditionellen Ansatz, wo die Roboterprogrammierung auf einem Weltmodell basiert, basiert sie bei der Subsumtionsarchitektur nach Brooks auf einer sensorengesteuerten Verhaltensweise. Das bedeutet, dass durch bestimmte anliegende Werte an den Sensoren ein bestimmtes Verhalten ausgelöst wird, wobei die einzelnen Verhaltensweisen parallel laufen. Damit es dabei zu keiner Überlappung der einzelnen Verhaltensweisen kommt, werden sie nach Prioritätsstufen geordnet. Somit kann ein höherwertiges Verhalten bei entsprechenden Sensorenwerten die volle Kontrolle über den Roboter übernehmen und damit das niederwertige Verhalten subsumieren (ersetzen).

Um dieses Konzept besser zu verdeutlichen, wird folgendes Beispiel eingeführt: Ein Roboter ist mit Berührungssensoren (Bumper) zur Erkennung von Hindernissen ausgestattet. Er fährt also solange umher, was hier dem Verhalten „cruise“ entspricht, bis ein Berührungssensor aktiviert wird. Durch das Aktivieren des Berührungssensors wird das Verhalten „avoid“, also das Ausweichen vor dem Hindernis ausgelöst. Damit diese Verhaltensweise einwandfrei funktionieren kann, muss das Verhalten „avoid“ eine höhere Priorität bekommen als das Verhalten „cruise“. Die Entscheidung welches Verhalten letztendlich den Roboter steuert und somit die Kontrolle über die Motoren übernimmt, wird durch den Kreis mit dem „S“ symbolisiert.

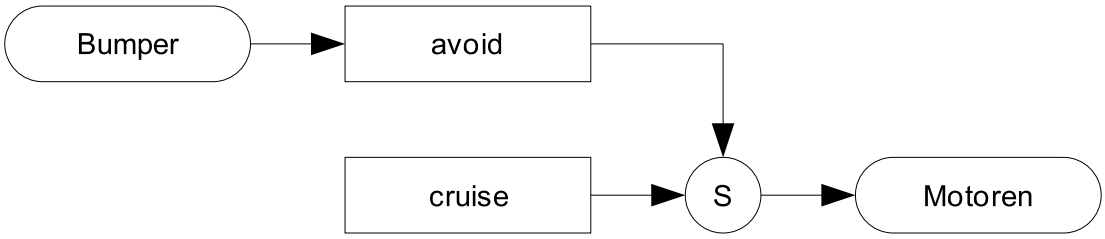


Abbildung 3: Bei Aktivierung der Bumper übernimmt das Verhalten "avoid" die Kontrolle über die Motoren, ansonsten übernimmt das Verhalten „cruise“ die Kontrolle.

Natürlich gibt es nicht nur immer eine Eins-zu-Eins-Relation zwischen den Sensoreingängen und den Verhaltensweisen. Es kann genauso gut eine Kombination von mehreren Sensoreingängen ein bestimmtes Verhalten oder auch ein einziger Sensoreingang, abhängig von seinem Sensorwert, verschiedene Verhaltensweisen auslösen. [INO-00]

## 4.1 Implementierung

Da der RCX über ein präemptives Multitasking verfügt, ist es relativ einfach, die Subsumtionsarchitektur auf diesem System zu implementieren. Die Grundlage dieser Realisierung besteht darin, dass jedes Verhalten eine eigene Task ist. Hinzu kommt noch eine weiterer Task, die die Entscheidung trifft, welches Verhalten gerade die höchste Priorität besitzt und somit seine Befehle an die Motoren mittels eines Unterprogramms schicken kann und die Task-main, wo die einzelnen Tasks bzw. Verhaltensweisen gestartet werden. Wenn eine Verhaltensweise nicht aktiv ist, wird dieser Zustand durch eine eigene Konstante gekennzeichnet. Dadurch ergibt sich die folgende allgemeine Lösung zur Implementierung der Subsumtionsarchitektur: [INO-00]

```
//Definition der Konstanten (z.B. Motorbefehle)
#define COMMAND_NONE    -1
#define COMMAND_FORWARD 1
#define COMMAND_STOP    2
⋮
```

```
//Definition der Variablen
int Verhalten1Command;
int Verhalten2Command;
int motorCommand;

//Verhaltensweisen des Roboters
task Verhalten1()
{
  while(true)
  {
   if(Sensor_aktiviert)                  //Verhaltensweise 1 wird aktiviert
   {
    Verhalten1Command = COMMAND_FORWARD; //z. B. Reaktionen, Motorbefehle
   }
   else Verhalten1Command = Command_None; // Verhaltensweise 1 nicht aktiv
  }
}
task Verhalten2()
{
  while(true)
  {
   if(Sensor_aktiviert)                  // Verhaltensweise 2 wird aktiviert
   {
    Verhalten2Command = COMMAND_STOP;    //z. B. Reaktionen, Motorbefehle
   }
   else Verhalten2Command = Command_None; //Verhaltensweise 2 nicht aktiv
  }
}
⋮

/*Arbiter trifft die Entscheidung, welche Verhaltensweise ihre Befehle an die
Motoren schicken kann*/
task arbitrate()
{
  while(true)
  {
  if(Verhalten1Command != Command_None) motorCommand = Verhalten1Command;
  if(Verhalten2Command != Command_None) motorCommand = Verhalten2Command;
  ⋮
  motorControl();
  }
}
```

```
//Hauptprogramm
task main()
{
  initialize();                          // Aufrufen der Initialisierung
  Verhalten1Command = Command_None;      //Anfangswerte zuweisen
  Verhalten2Command = Command_None;
  ⋮
  start Verhalten1;                      //Starten der einzelnen Task
  start Verhalten2;
  ⋮
  start arbitrate;
}

//Unterprogramme
sub initialize()
{                                        //Initialisierung; z. B. der Sensoren
}
sub motorContol()
{
  if(motorCommand == COMMAN_FORWARD) OnFwd(OUT_A);   //Schaltet Ausgang A ein
  else if(motorCommand == COMMAND_STOP) Off(OUT_A);  //Schaltet Ausgang A aus
  ⋮
}
```

# 5 Kurzübersicht über NQC

Auf den folgenden Seiten befindet sich eine Kurzübersicht der wichtigsten NQC-Elemente für das Laborpraktikum „Mobile Roboter“. Diese Seiten sollen eine zusätzliche Erleichterung und auch ein Nachschlagewerk für die Programmieraufgaben darstellen.

## 5.1 Anweisungen

Jedes NQC-Programm besteht wie ein C-Programm aus Anweisungen (statements), die in dem Programm auszuführenden Aktionen festlegen. Die Anweisungen werden durch ein Semikolon abgeschlossen. Sie werden prinzipiell sequentiell abgearbeitet. Durch Kontrollstrukturen wie z. B. Schleifen kann die sequentielle Abarbeitungsreihenfolge geändert werden.

| ***Anweisung*** | ***Beschreibung*** |
|---|---|
| while (Bedingung) Anweisung | Führt Anweisung solange aus, wie die Bedingung wahr ist |
| do Anweisung while (Bedienung) | Führt Anweisung mindestens einmal aus und danach solange, bis die Bedienung wahr ist |
| for(Initialisierung; Bedingung; Inkrementierung) Anweisung | Führt Anweisungen solange aus, wie die Bedingung wahr ist (Zählschleife) |
| until (Bedingung) Anweisung | Anweisung solange ausführen, bis die Bedingung wahr ist |
| break | Beendet while-, do- oder until-Anweisung |
| continue | Springe zum nächsten Durchlauf der while-, do- oder until-Anweisung |
| repeat (Wert) Anweisung | Wiederholt die Anweisung so oft, wie angegeben (Wert) |
| if (Bedingung) Anweisung1 | Führt Anweisung1 aus, wenn Bedingung wahr ist |
| if (Bedingung) Anweisung1 else Anweisung2 | Führt Anweisung1 aus, wenn Bedingung wahr ist, sonst wird Anweisung2 ausgeführt |
| start task_name | Startet die angegebene Task |
| stop task_name | Stoppt die angegebene Task |
| Funktionsname (Argument) | Ruft die angegebene Funktion auf und übergibt die Argumente (Werte) |
| Variable = Ausdruck | Wertet Ausdruck aus und weist ihn anschließend der Variable zu |
| Variable += Ausdruck | Wertet Ausdruck aus und addiert ihn anschließend zur Variable |
| Variable -= Ausdruck | Wertet Ausdruck aus und subtrahiert ihn anschließend von der Variable |
| Variable *= Ausdruck | Wertet Ausdruck aus und multipliziert diesen Wert dann mit der Variable und weist ihn anschließend der Variable zu |
| Variable /= Ausdruck | Wertet Ausdruck aus und dividiert die Variable dann durch diesen Wert und weist ihn anschließend der Variable zu |
| Variable &= Ausdruck | Wert der Variable und Wert des Ausdrucks werden bitweise UND-verknüpft, Ergebnis wird der Variable zugewiesen |
| Variable \|= Ausdruck | Wert der Variable und Wert des Ausdrucks werden bitweise ODER-verknüpft, Ergebnis wird der Variable zugewiesen |

| *Anweisung* | *Beschreibung* |
|---|---|
| return | Beendet Funktion und kehrt zur aufrufenden Anweisung zurück |
| Ausdruck | Wertet Ausdruck aus |

Tabelle 7: Anweisungen

## 5.2 Bedingungen

Mit Hilfe von Bedingungen kann der Ablauf eines Programms durch gewisse Entscheidungen beeinflusst werden. Dazu wird in der Regel ein Ausdruck mit einem anderen Ausdruck verglichen. Das Ergebnis dieses Vergleichs bestimmt dann den weiteren Ablauf des Programms.

| *Bedingung* | *Bedeutung* |
|---|---|
| true | wahr; entspricht der logischen 1 |
| false | falsch; entspricht der logischen 0 |
| Ausdruck1 == Ausdruck2 | wahr, wenn beide Ausdrücke gleich sind |
| Ausdruck1 != Ausdruck2 | wahr, wenn Ausdruck1 nicht gleich Ausdruck 2 ist |
| Ausdruck1 < Ausdruck2 | wahr, wenn Ausdruck1 kleiner als Ausdruck2 ist |
| Ausdruck1 <= Ausdruck 2 | wahr, wenn Ausdruck1 kleiner gleich Ausdruck2 ist |
| Ausdruck1 > Ausdruck2 | wahr, wenn Ausdruck1 größer als Ausdruck2 ist |
| Ausdruck1 >= Ausdruck2 | wahr, wenn Ausdruck1 größer gleich Ausdruck2 ist |
| ! Bedingung | Logische Negation einer Bedingung |
| Bedingung1 && Bedingung2 | Logische UND-Verknüpfung zweier Bedingungen |
| Bedingung1 \|\| Bedingung2 | Logische ODER-Verknüpfung zweier Bedingungen |

Tabelle 8: Bedingungen

## 5.3 Ausdrücke

Ein Ausdruck wird immer aus mindestens einem Operanden und einem oder mehren Operatoren, die auf die Operanden angewendet werden, gebildet. Dabei kann ein Operand aus einer Konstante, einem Sensorwert oder einer Variable bestehen.

| *Operand* | *Beschreibung* |
|---|---|
| Zahl | konstanter Wert; z. B.: 5 |
| Variable | benannte Variable; z. B.: i |
| Timer(Timer_Nr) | aktuelle Zeit des gewählten Timers (0 bis 3) |
| Random(n) | erzeugt Zufallszahl zwischen 0 und n |
| Watch() | aktuelle Zeit der Systemuhr in Minuten |
| Message() | Letzte erhaltene Nachricht |
| SensorValue(Sensor) | aktueller Wert des angegebenen Sensors n, wobei n zwischen 0 und 2 liegt |

Tabelle 9: Operanden

Bei der folgenden Tabelle der Operatoren sind die Operatoren mit fallender Priorität geordnet, also von der höchsten zur niedrigsten Priorität.

| *Operator* | *Beschreibung* | *Assoziativität* | *Einschränkung* |
|---|---|---|---|
| abs() | Absolutwert | keine | keine |
| sign() | Vorzeichen des Operanden | keine | keine |
| ++ | Inkrement | links | nur Variablen |
| -- | Dekrement | links | nur Variablen |
| - | Unäres Minus (Vorzeichen) | rechts | keine |
| ~ | Bitweise Negation | rechts | nur Konstanten |
| ! | Logische Negation | rechts | keine |
| << | Links-Shift | links | nur Konstanten |
| >> | Rechts-Shift | links | nur Konstanten |
| < > | kleiner als, größer als | links | keine |
| <= >= | kleiner gleich, größer gleich | links | keine |
| == | Gleichheit | links | keine |
| != | Ungleichheit | links | keine |
| & | Bitweises UND | links | keine |
| ^ | Bitweises XOR | links | nur Konstanten |
| \| | Bitweises ODER | links | keine |
| && | Logisches UND | links | keine |
| \|\| | Logisches ODER | links | keine |

Tabelle 10: Operatoren

## 5.4 RCX-Funktionen

Für die Steuerung des RCX gibt es spezielle Funktionen, die in der Regel als Argument einen konstanten Ausdruck erfordern. Die Ausnahme bilden die Funktionen, die ein Sensor als Argument verwenden und diejenigen, die einen beliebigen Ausdruck verwenden können. Bei den Funktionen für die Sensoren muss das Argument ein Sensorname wie z. B. SENSOR_1 sein.

| *Funktion* | *Beschreibung* |
|---|---|
| SetSensor(Sensoranschluss, Sensorkonfiguration) | Konfiguriert den angegeben Sensor |
| SetSensorType(Sensoranschluss, Sensortyp) | Stellt den Sensortyp des angegeben Sensors ein |
| SetSensorMode(Sensoranschluss, Sensormodus) | Stellt den Sensormodus des angegeben Sensors ein |
| ClearSensor(Sensoranschluss) | Löscht den Messwert des angegeben Sensors |
| On(Ausgänge) | Schaltet die angegebenen Ausgänge ein |
| Off(Ausgänge) | Schaltet die angegebenen Ausgänge aus |
| Float(Ausgänge) | Schalten den Freilauf für die angegebenen Ausgänge ein |
| Fwd(Ausgänge) | Versetzt die angegebenen Ausgänge in Vorwärtsrichtung |
| Rev(Ausgänge) | Versetzt die angegebenen Ausgänge in Rückwärtsrichtung |
| Toggle(Ausgänge) | Kehrt in Bewegungsrichtung der angegebenen Ausgänge um |
| OnFwd(Ausgänge) | Schaltet die angegebenen Ausgänge in Vorwärtsrichtung ein |
| OnRev(Ausgänge) | Schaltet die angegebenen Ausgänge in Rückwärtsrichtung ein |
| OnFor(Ausgänge, Dauer) | Schaltet die angegebenen Ausgänge für die festgelegte Zeitdauer mit einer Auflösung von 10ms ein. |
| SetPower(Ausgänge, Leistung) | Stellt die Leistungsstufe von 0 (niedrig) bis 7 (hoch) für die angegebenen Ausgänge ein |
| Wait(Zeitdauer) | Wartet die angegebene Zeitdauer mit einer Auflösung von 10ms ab. |
| PlaySound(Name) | Spielt den angegebenen Klang (Name) ab |
| PlayTone(Frequenz, Zeitdauer) | Spielt einen Ton mit der angegebenen Frequenz für die angegebene Zeitdauer (Hundertstelsekunden) ab |
| ClearTimer(Timer_Nr) | Löscht den aktuellen Wert vom angegebenen Timer |
| StopAllTask() | Stoppt alle zur Zeit laufenden Tasks |
| SelectDispaly(Modus) | Stellt den ausgewählten Anzeigemodus ein |
| SendMessage(Nachricht) | Sendet eine IR-Nachricht. Nachricht besteht aus einer ganzzahligen Zahl zwischen 1 und 255 oder aus einem Ausdruck |
| ClearMessage(Nachricht) | Löscht den Nachrichtenpuffer |
| CreateDatalog(Größe) | Erzeugt einen neuen Datenspeicher der angegebenen Größe |
| AddToDatalog(Wert) | Fügt den angegebenen Wert dem Datenspeicher zu. Wert kann ein Ausdruck sein. |
| SetWatch(Stunden, Minuten) | Stellt die Systemuhr nach den angegebenen Werten ein |
| SetTxPower(Leistung) | Stellt die Sendeleistung der IR-Schnittstelle ein |

Tabelle 11: RCX-Funktionen

## 5.5 RCX-Konstanten

Für einige Argumente der RCX-Funktionen gibt es anstatt der Zahlenwerte benannte Konstanten, die die Lesbarkeit des Programmcodes ungemein erhöhen. Aufgrund dessen wurde bei dieser Ausarbeitung so weit wie möglich auf die Zahlenwerte verzichtet. Daher wird in der folgenden Tabelle nur der Zusammenhang zwischen den RCX-Funktionen und den dazugehörigen Konstanten hergestellt.

| ***Funktion*** | ***Konstante*** |
|---|---|
| SetSensor() | SENSOR_TOUCH, SENSOR_LIGHT, SENSOR_ROTATION, SENSOR_PULSE, SENSOR_EDGE |
| SetSensorMode() | SENSOR_MODE_RAW, SENSOR_MODE_BOOL, SENSOR_MODE_EDGE, SENSOR_MODE_PULSE, SENOSR_MODE_PERCENT, SENSOR_MODE_ROTATION |
| SetSensorType() | SENSOR_TYPE_TOUCH, SENSOR_TYPE_LIGHT, SENSOR_TYPE_ROTATION |
| On(), Off() | OUT_A, OUT_B, OUT_C |
| SetPower() | OUT_LOW, OUT_HALF, OUT_FULL |
| PlaySound() | SOUND_CLICK, SOUND_DOUBLE_BEEP, SOUND_DOWN, SOUND_UP, SOUND_LOW_BEEP, SOUND_FAST_UP |
| SelectDisplay() | DISPLAY_WATCH, DISPLAY_SENSOR_1, DISPLAY_SENSOR_2, DISPLAY_SENSOR_3, DISPLAY_OUT_A, DISPLAY_OUT_B, DISPLAY_OUT_C |
| SetTxPower() | TX_POWER_LO, TX_POWER_HI |

Tabelle 12: RCX-Konstanten

## 5.6 Schlüsselwörter

In NQC gibt es wie bei jeder anderen Programmiersprache auch eine Reihe von Wörtern, die als Schlüsselwörter definiert sind. Die Schlüsselwörter sind Programmierbefehle der Programmiersprache und dürfen deshalb nicht als Bezeichner verwendet werden, da sie vom NQC-Compiler für die Programmiersprache selbst reserviert sind. Deshalb erfolgt an dieser Stelle eine Auflistung dieser Schlüsselwörter.

| | | | |
|---|---|---|---|
| _event_src | break | for | start |
| _nolist | case | goto | stop |
| _res | catch | if | sub |
| _sensor | const | inline | switch |
| _taskid | continue | int | task |
| _type | default | monitor | true |
| abs | do | repeat | void |
| acquire | else | return | while |
| asm | false | sign | |

# Literaturverzeichnis

NQC-03: Dave Baum, NQC Programmer`s Guide, 2003, Dr. - Ing- Fritz Mehner (Übersetzer),
Stand: 22.06.2004, http://lug.mfh-iserlohn.de/lego

INO-00: Knudsen & Noga, Das Inoffizielle Handbuch für LEGO MINDSTORMS Roboter, 2000, 1. Auflage, O`Reilly