Programmierung LEGO NXT Roboter mit NXC

(beta3.0 oder höher)

(Version 2.2, 7. Juni 2007)

Von Daniele Benedettelli

mit Änderungen von John Jansen

Übersetzung (und Ergänzungen) ins Deutsche von Thorsten Leimbach, Sebastian Trella

(Version-DE 1.0, 24. Juli 2009)

Vorwort

Was bereits für die guten alten Mindstorms RIS, Cybermaster und Spybotics galt, gilt nun auch für den Mindstorms NXT Stein, um die volle Leistung ausnutzen zu können, wird eine Programmierumgebung benötigt, die handlicher ist als die, im Einzelhandel erhältliche, grafische Programmierumgebung NXT-G, welche ähnlich der Software LabVIEW der Firma National Instruments ist.

NXC ist eine Programmiersprache, die von John Hansen, speziell für die Lego-Roboter entwickelt wurde. Sollten Sie noch nie ein Programm zuvor geschrieben haben, keine Sorge! NXC ist wirklich einfach zu erlernen und anzuwenden. Dieses Tutorial wird Sie bei Ihren ersten Schritten zur Erreichung dieses Ziels leiten.

Um textbasierte Programme zu erleichtern, gibt es das Bricx Command Center (BricxCC). Diese Umgebung hilft Ihnen, Ihre Programme zu schreiben, auf den Roboter zu übertragen, diese zu starten und zu stoppen, den Speicher des NXT zu durchsuchen, Sound-Dateien für die Verwendung auf dem NXT-Stein zu konvertieren und vieles mehr. BricxCC funktioniert fast wie ein Text-Prozessor, aber mit einigen Extras. Dieses Tutorial basiert auf BricxCC (Version 3.3.7.16 oder höher) als integrierte Entwicklungsumgebung (IDE).

Sie können es kostenlos aus dem Internet unter folgender Adresse herunterladen:
http://bricxcc.sourceforge.net/

BricxCC läuft auf Windows-PCs (95, 98, ME, NT, 2K, XP, Vista[1]). Die NXC Sprache kann auch auf anderen Plattformen verwendet werden. BricxCC kann unter folgender Web-Seite herunterladen werden:
http://bricxcc.sourceforge.net/nxc/

Der Großteil dieses Tutorial sollte auch für andere Plattformen zutreffend sein, es sei denn, einige der Werkzeuge, die in BricxCC und in der Farbcodierung (Text Highlighting) enthalten sind, gehen verloren.

Dieses Tutorial wurde aktualisiert, um mit Beta 30 von NXC und höhere Versionen arbeiten zu können. Einige der Programmbeispiele lassen sich nicht mit älteren Versionen als Beta-30 kompilieren.

Hinweis in Eigener Sache: Meine Web-Seite ist voll von Lego Mindstorms RCX und NXT Inhalten, darunter auch ein PC-Tool zur Kommunikation mit NXT:
http://daniele.benedettelli.com

Danksagungen

Vielen Dank an John Hansen, dessen Arbeit unbezahlbar ist!

Vorwort zur übersetzen Version

In der Übersetzung wurde, wo es sinnvoll erschien Sachverhalte ergänzt (meistens in Fußnoten zu finden) bzw. korrigiert. Die übersetzte Version bezieht sich allerdings überwiegend auf die Version 2.2 von Daniele Benedettelli, wobei die beschriebenen Sachverhalte sinngemäß, frei übersetzt wurden. Die Screenshots und die Programme wurden bis auf wenige Ausnahmen unverändert übernommen.

[1] Es gibt mittlerweile (2009) auch eine Beta-Version für MacOS, Linux und andere Unix Systeme

1 Das erste selbstgeschriebene Programm

In diesem Kapitel wird Ihnen gezeigt, wie man ein sehr einfaches Programm schreiben kann. Es wird ein Roboter programmiert, der 4 Sekunden vorwärts fahren soll, dann für weitere 4 Sekunden rückwärts fährt um anschließend anzuhalten. Nicht sehr spektakulär, aber es wird Sie in die Grundidee der Programmierung einführen und Ihnen zeigen, wie einfach die Programmierung ist. Aber bevor wir ein Programm schreiben, benötigen wir einen Roboter.

Konstruktion eines Roboters

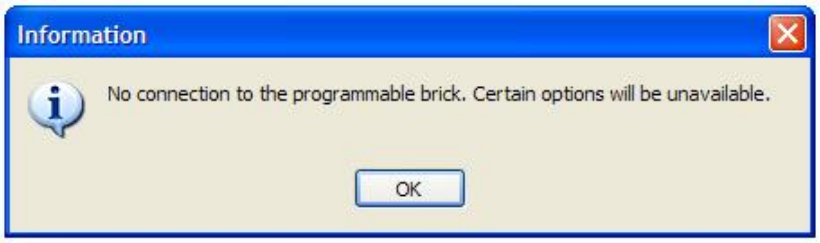
Den Roboter, den wir in diesem Tutorial verwenden nennt sich Tribot. Dies ist der erste Roboter den Sie bauen sollten, nachdem Sie ihr NXT-Set ausgepackt haben.
Der einzige Unterschied ist (zur original LEGO Bauanleitung a. d. Übersetzers), Sie müssen den rechten Motor mit Port (Ausgang[2]) A, den linken Motor mit Port C und den Greifer mit Port B verbinden. Ihr Roboter sollte nun wie folgt aussehen:[3]

Bitte vergewissern Sie sich, dass Sie die Mindstorms NXT Fantom Treiber, welche mit dem NXT-Set mitgeliefert werden, korrekt installiert haben.

Das Bricx Command Center

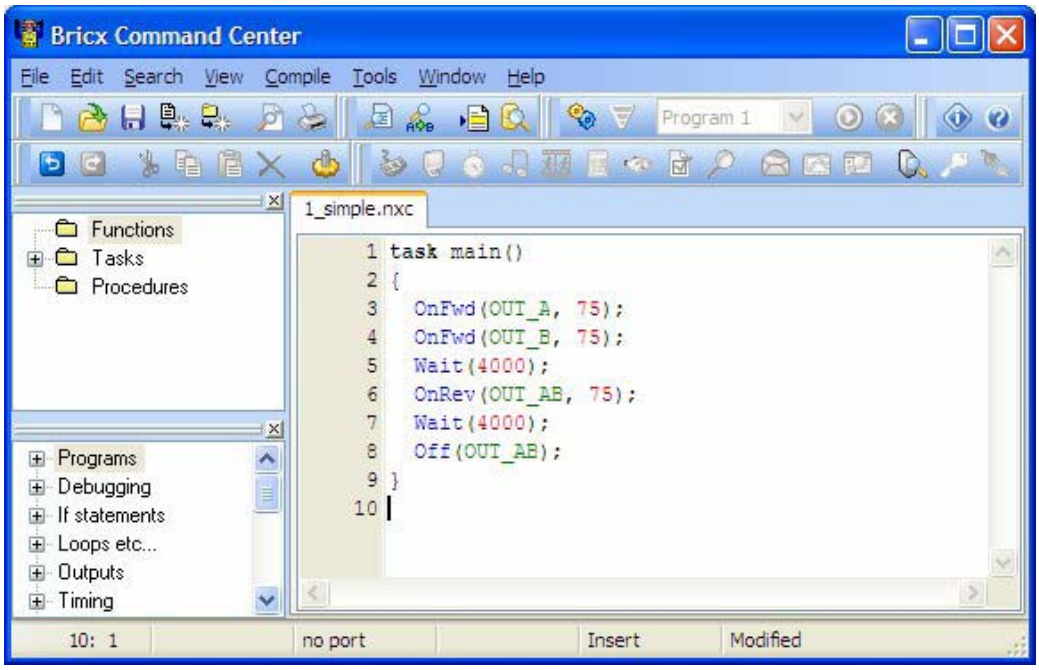
Sie schreiben die Programme mit dem Bricx Command Center. Starten Sie es durch einen Doppelklick auf das Symbol BricxCC. (Es wird davon ausgegangen, dass BricxCC bereits installiert ist. Wenn nicht, laden Sie es von der Webseite (siehe oben), und installieren Sie es in ein beliebiges Verzeichnis. Das Programm wird Sie fragen, wie der Roboter mit Ihrem PC verbunden ist.

(Sollt Ihr Roboter nicht verbunden bzw. nicht angeschaltet sein erscheint die folgende Fehlermeldung a. des Übersetzers)



Schalten Sie den Roboter ein und drücken Sie **OK**. BricxCC wird (wahrscheinlich) den Roboter automatisch finden. Nun erscheint die Benutzeroberfläche, wie unten angezeigt (ohne die Registerkarten).

[2] Die Anschlüsse für die Motoren können auch als Ausgang und die Anschlüsse für Sensoren als Eingang bezeichnet werden.
[3] Sollten Sie eine Education Version des LEGO MINDSTORMS NXT Sets besitzen, kann der Tribot abweichen.



Die Oberfläche sieht aus wie ein Standard-Text-Editor, mit den üblichen Menüs und Schaltflächen zum Öffnen, Speichern, Drucken und Editieren von Dateien usw. Aber es gibt auch einige spezielle Menüs für die Erstellung (kompilieren) und den Download der Programme auf den Roboter, sowie für das Anzeigen von Informationen des Roboter. Dies können Sie aber für diesen Moment vorerst ignorieren.

Wir gehen nun daran ein neues, erstes Programm zu schreiben. Drücken Sie das Icon **New File** (leeres weißes Blatt) oder drücken Sie die Tastenkombination *Strg + N* um ein neues, leeres Fenster anzulegen.

Ein Programm schreiben

Tippen Sie nun das folgende Programm ab:

```
task main()
{
  OnFwd(OUT_A, 75);
  OnFwd(OUT_C, 75);
  Wait(4000);
  OnRev(OUT_AC, 75);
  Wait(4000);
  Off(OUT_AC);
}
```

Speichern Sie das Programm indem Sie Strg + S drücken. Sollten Sie keinen eigenen Namen eingeben, wird das Programm als Untiteled1.nxc gespeichert. (Achten Sie darauf, dass Ihr Programm den Zusatz **.nxc** hat a. d. Übersetzers)

Es könnte auf den ersten Blick etwas kompliziert aussehen, lassen Sie es uns deshalb analysieren:

Programme in NXC bestehen aus Tasks (`task`). Unser Programm hat nur einen Task, namens `main`. Jedes Programm muss einen main-Task besitzen, dieser wird immer vom Roboter ausgeführt. Sie erfahren mehr über Tasks in Kapitel 6. Ein Task besteht aus einer Reihe von Befehlen, auch Anweisung genannt. Es gibt Klammern um diese Befehlsblöcke, sodass klar ist, zu welchem Task sie gehören. Jede Anweisung endet mit einem Semikolon. Auf diese Weise wird deutlich, wo ein Befehl endet und wo der nächste beginnt[4].

[4] Sie könnten auch alle Befehle nur durch Komma getrennt in eine Zeile schreiben. Hierdurch würden Sie allerdings schnell die Übersicht über Ihr Programm verlieren. Es empfiehlt sich daher immer, Befehle und Anweisungen klar zu strukturieren.

Ein Task sieht in der Regel wie folgt aus:

```
task main()
{
  statement1;
  statement2;
  …
}
```

Das Programm hat sechs Anweisungen. Werfen wir einen Blick auf darauf:

```
OnFwd(OUT_A, 75);
```

Diese Anweisung „sagt“ dem Roboter den Motor auf Ausgang A zu starten (Ausgang A wird der *Port* am NXT genannt, der mit A beschriftet ist.) `OnFwd` schaltet den Motor Ein und setzt den Motor dabei gleichzeitig auf „Vorwärts“. Die nachfolgende Zahl (`75`) setzt die Geschwindigkeit des Motors auf 75% der maximalen Geschwindigkeit.

```
OnFwd(OUT_C, 75);
```

Die gleiche Anweisung, jetzt aber für den Motor C. Nach diesen beiden Befehlen, drehen sich beide Motoren, und der Roboter bewegt sich vorwärts.

```
Wait(4000);
```

Jetzt ist es Zeit, eine Weile zu warten. Der Befehl (`Wait`) weist an, 4 Sekunden (`4000`) zu „warten“. Das Argument, die Zahl zwischen den Klammern, gibt die Zeit in 1 / 1000 Sekunde an. Damit können Sie sehr genau bestimmen, wie lange das Programm zu warten hat. Hier hält das Programm exakt für 4 Sekunden an, ehe es mit der Abarbeitung des nächsten Befehls weiter voranschreiten bzw. der Roboter die nächste Aktion ausführt.

```
OnRev(OUT_AC, 75);
```

Der nächste Befehl, weist den Roboter an in umgekehrter Richtung, also rückwärts zu fahren. Beachten Sie, dass beide Motoren auf einmal mit einem einzigen Argument `OUT_AC` angesteuert werden können. Auch die ersten beiden Befehle hätten auf diese Weise geschrieben werden können.

```
Wait(4000);
```

Es wird wieder 4 Sekunden gewartet.

```
Off(OUT_AC);
```

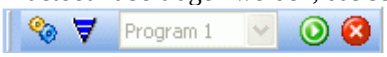
Zum Schluss werden beide Motoren mit `Off` ausgeschaltet.

Das Programm in Kurzfassung: Es bewegt beide Motoren für 4 Sekunden vorwärts, dann rückwärts für 4 Sekunden, und schließlich schaltet es sie aus.

Wahrscheinlich haben Sie die Farbgebung bemerkt während Sie das Programm eingegeben haben. Diese erscheinen automatisch. Die Farben und Ausdrucksweisen, die vom Editor für das sogenannte Syntax-Highlighting benutzt werden, können individuell angepasst werden.

Das Programm starten

Sobald Sie ein Programm geschrieben haben, muss es kompiliert werden (das heißt, es wird in Binär-Code übertragen, damit der Roboter es „verstehen“ und ausführen kann) und an den Roboter via USB-Kabel oder Bluetooth übertragen werden, das so genannte "Downloaden" des Programms.



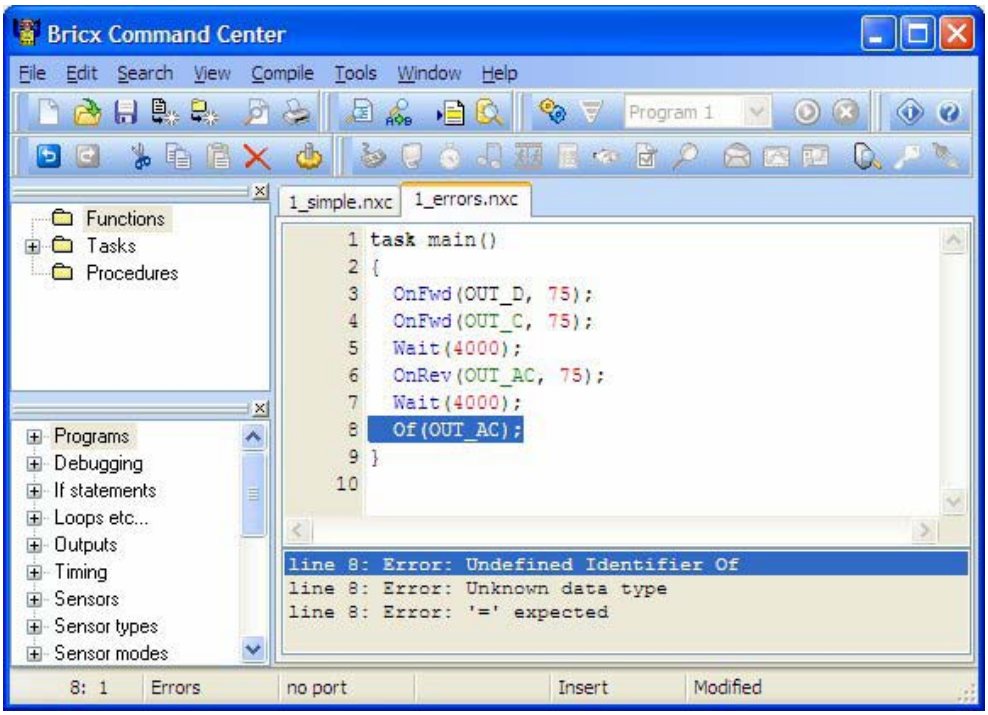
Hier sehen Sie die Knöpfe (von links nach rechts), welche es Ihnen ermöglichen ein Programm zu
(1) kompilieren, (2) herunterzuladen, zu (3) starten und zu (4) beenden

Drücken Sie den zweiten Knopf, unter der Annahme, dass Sie keine Fehler bei der Eingabe des Programms gemacht haben, wird es richtig kompiliert und auf den Roboter geladen. (Wenn es Fehler in Ihrem Programm gibt, werden Sie darüber benachrichtigt (siehe unten). Jetzt können Sie Ihr Programm starten. Hierfür nehmen Sie Ihren NXT. Im Menu, *Software files* führen Sie Ihr Programm aus. Denken Sie daran: Software-Dateien im Dateisystem des NXT haben den gleichen Namen wie die NXC-Quelldatei die Sie mit BricxCC erstellt haben. Um das Programm automatisch auszuführen, können Sie die Tastenkombination Strg + F5 wählen oder nach dem Herunterladen des Programms den grünen Knopf (mit mittigem weißem Pfeil) drücken.

Verhält sich der Roboter wie erwartet? Wenn nicht überprüfen Sie die Verbindung zwischen Roboter und PC.

Fehler in Ihrem Programm

Bei der Eingabe Ihres Programms gibt es eine realistische Chance, dass sich einige Fehler eingeschlichen haben. Der Compiler zeigt die Fehler und erläutert diese am unteren Rand des Fensters, wie in der folgenden Abbildung dargestellt:



Der Compiler wählt automatisch den ersten Fehler (hier: Vertippen beim Namen des Motors Of statt Off). Bei mehreren Fehlern im Programm, klicken Sie auf die Fehlermeldungen um zur fehlerhaften Stelle im Programm zu gelangen. Beachten Sie, dass oft Fehler am Anfang des Programms, anderen Fehlermeldungen an anderen Orten nach sich führen können. Es ist also besser nur den ersten „richtigen“ Fehler zu korrigieren und anschließend das Programm erneut zu kompilieren. Beachten Sie außerdem, dass das Syntax-Highlighting sehr hilfreich bei der Vermeidung von Fehlern ist. Schreiben Sie beispielsweise Of anstatt Off, wird der Befehl nicht wie `Off` , also farblich Blau hervorgehoben (highlighted).

Es gibt auch Fehler, die nicht vom Compiler gefunden werden können. Hätten wir z.B. `OUT_B` getippt würde dies dazu führen, dass der falsch Motor angesprochen wird. Der Compiler kann solche „logischen“ Fehler nicht finden. Wenn Ihr Roboter also ein unerwartetes Verhalten an den Tag legt, ist es wahrscheinlich dass etwas an Ihrer Programm-Logik falsch ist.

Ändern der Geschwindigkeit

Wie Sie bemerkt haben, bewegt sich der Roboter ziemlich schnell. Um die Geschwindigkeit zu ändern müssen Sie lediglich den zweiten Parameter innerhalb der Klammern ändern. Die Geschwindigkeit ist eine Zahl zwischen 0 und 100. Die Zahl 100 bedeutet maximale Geschwindigkeit, 0 bedeutet Stopp (NXT-Servomotoren halten die Position). Hier ist eine neue Version des Programms, das den Roboter veranlasst sich langsam zu bewegen.

```
task main()
{
  OnFwd(OUT_AC, 30);
  Wait(4000);
  OnRev(OUT_AC, 30);
  Wait(4000);
  Off(OUT_AC);
}
```

Zusammenfassung

In diesem Kapitel haben Sie Ihr erstes Programm in NXC, mit der Programmierumgebung BricxCC geschrieben. Sie sollten jetzt wissen, wie man ein Programm schreibt, wie Sie es auf den Roboter laden und wie Sie den Roboter dazu veranlassen das Programm zu starten. BricxCC kann noch viel mehr. Um herauszufinden welche Möglichkeiten Ihnen BricxCC zur Verfügung stellt lesen Sie bitte die Dokumentation, die Ihnen mit dem Download mitgeliefert wurde. Dieses Tutorial wird sich in erster Linie mit der Sprache NXC befassen. Es wird nur dann auf Merkmale von BricxCC eingegangen, wenn Sie diese wirklich brauchen.

Sie haben zudem auch einige wichtige Aspekte der Sprache NXC gelernt. Zu aller erst haben Sie gelernt, dass jedes Programm einen Task namens `main` benötigt und dass dieser immer vom Roboter ausgeführt wird. Auch haben Sie die vier grundlegenden Motor-Befehle: `OnFwd` (), `OnRev` () und `Off` () kennengelernt. Schließlich haben Sie gelernt, wie der `Wait` () Befehl verwendet werden kann.

Tipp: Da oftmals beide Motoren des NXT nicht exakt gleich stark sind, kann es vorkommen, dass Ihr Roboter beim Geradeausfahren einen Drift (der Roboter zieht leicht nach link oder rechts) aufweist. Um dem entgegenzusteuern, sprechen Sie die Motoren einzeln an

OnFwd(OUT_A, 40;
OnFwd(OUT_C, 60

und variieren Sie dabei die Geschwindigkeit.

2 Ein viel interessanteres Programm

Unser erstes Programm war nicht wirklich beeindruckend. Lassen Sie uns versuchen, es noch interessanter zu gestalten. Wir werden Schritt für Schritt komplexere Programme schreiben und dabei einige wichtige Merkmale der Programmiersprache NXC einbauen.

Kurven fahren

Sie können Ihren Roboter durch Stillstand oder Umkehr der Richtung einer der beiden Motoren eine Kurve fahren lassen. Schreiben Sie das Beispielprogramm ab, laden Sie dies auf Ihren Roboter und starten Sie es. Der Roboter sollte ein Stück geradeaus fahren und dann eine 90-Grad Rechtskurve vollziehen.

```
task main()
{
  OnFwd(OUT_AC, 75);
  Wait(800);
  OnRev(OUT_C, 75);
  Wait(360);
  Off(OUT_AC);
}
```

Vielleicht müssen Sie den Parameter des zweiten `Wait`-Befehls anpassen, um eine 90-Grad Kurve zu fahren. Dies hängt von der Oberflächenbeschaffenheit ab, auf der der Roboter fährt. Anstatt derartige Änderungen im Wait-Befehl vorzunehmen, empfiehlt es sich einen Namen für diesen Parameterwert zu nutzen. In NXC können Sie konstante Werte festlegen, wie folgendes Beispielprogramm zeigt:

```
#define MOVE_TIME 1000    // Konstante definieren
#define TURN_TIME 360

task main()
{
  OnFwd(OUT_AC, 75);
  Wait(MOVE_TIME);        // Einsetzen der Konstanten
  OnRev(OUT_C, 75);
  Wait(TURN_TIME);
  Off(OUT_AC);
}
```

Die ersten beiden Zeilen definieren zwei Konstanten. Diese können als Parameter nun im Programm verwendet werden. Definieren von Konstanten ist aus zwei Gründen empfehlenswert:

1. Das Programm wird besser lesbar, und
2. Die Werte im Programm können leichter geändert werden.

Beachten Sie, dass BricxCC den define-Anweisungen eine eigene Farbe zuweist. Wie wir in Kapitel 6 sehen werden, können neben Konstanten auch weitere Programm-Konstrukte definiert werden.

Schleifen

Lassen Sie uns nun versuchen, ein Programm zu schreiben, mit dem Sie den Roboter ein Quadrat fahren lassen. Ein Quadrat zu fahren bedeutet:
Vorwärts fahren – 90 Grad Drehung – Vorwärts fahren – 90 Grad Drehung – Vorwärts fahren – 90 Grad Drehung usw.
Um ein Quadrat zu programmieren, können wir einfach die obigen Code-Sequenz vier Mal wiederholen, aber dies kann auch viel einfacher, mit der sogenannten repeat-Anweisung, programmiert werden.

```
#define MOVE_TIME 500
#define TURN_TIME 500

task main()
{
  repeat(4)               // Wiederholung der Befehle
  {
    OnFwd(OUT_AC, 75);
    Wait(MOVE_TIME);
    OnRev(OUT_C, 75);
    Wait(TURN_TIME);
  }
  Off(OUT_AC);
}
```

Die Zahl (4) innerhalb der Klammern der **repeat**-Anweisung gibt an, wie oft der Code zwischen den geschweiften Klammern wiederholt wird. Beachten Sie, dass im obigen Programmbeispiel die define-Anweisungen wieder benutzt wurden. Dies ist zwar nicht notwendig, trägt aber zur Lesbarkeit des Programms bei.

Als letztes Beispiel, lassen Sie uns den Roboter 4-mal ein Quadrat fahren. Hier das Beispielprogramm:

```
#define MOVE_TIME 500
#define TURN_TIME 500

task main()
{
  repeat (4)
  {
    repeat(4)
    {
      OnFwd(OUT_AC, 75);
      Wait(MOVE_TIME);
      OnRev(OUT_C, 75);
      Wait(TURN_TIME);
    }
    Off(OUT_AC);
  }
}
```

Das Beispielprogramm zeigt zwei repeat-Anweisungen. Eine repeat-Anseiung befindet sich innerhalb der anderen. Dies wird auch eine "verschachtelte" repeat-Anweisung bzw. allgemein –Verschachtelung- genannt. Sie können die Verschachtelung beliebig oft wiederholen. Verwenden Sie Zeit und Sorgfalt auf das setzen der Klammern und das Einrücken der Programm-Befehle.

- Der Task beginnt bei der ersten (geschweiften) Klammer und endet bei der letzten.
- Die erste repeat-Anweisung beginnt bei der zweiten (geschweiften) Klammer und endet bei der Fünften.
- Die geschachtelte repeat-Anweisung beginnt mit der dritten (geschweiften) Klammer und endet mit der Vierten.

Wie Sie erkennen, werden Klammern immer paarweise gesetzt. Das Code-Stück (Programmblock) zwischen den Klammern wird (zur besseren Übersicht) eingerückt.

Kommentare einfügen

Um das Programm noch lesbarer zu gestalten, empfiehlt es sich das Programm mit Bemerkungen (Kommentare) zu versehen. Wenn Sie / / in einer Zeile einfügen, wird der Rest der Zeile ignoriert und kann zur Kommentierung genutzt werden. Ein langer Kommentar kann zwischen / * und * / geschrieben werden. Kommentare werden in BricxCC hervorgehoben (Syntax Highlighting). Das vollständige Programm könnte somit wie folgt aussehen:

```
/*
  Dieses Programm lässt den Roboter 10 Quadrate fahren
*/

#define MOVE_TIME 500      // Zeit für geradeaus Fahrt
#define TURN_TIME 360      // Zeit für 90Grad-Drehung
degrees

task main()
{
  repeat(10)               // Zehn Quadrate fahren
  {
    repeat(4)
    {
      OnFwd(OUT_AC, 75);
      Wait(MOVE_TIME);
      OnRev(OUT_C, 75);
      Wait(TURN_TIME);
    }
  }
  Off(OUT_AC);             // Motoren ausschalten
}
```

Zusammenfassung

In diesem Kapitel haben Sie die richtige Anwendung der repeat-Anweisung und die Verwendung von Kommentaren gelernt. Auch haben Sie verschachtelte Anweisungen und die Nutzung von Einrückungen gesehen. Mit dem bisher Gelernten, können Sie einen Roboter entlang aller möglichen Wege fahren lassen. Eine gute Übung für Sie: versuchen Sie die Programme dieses Kapitels zu variieren, bevor Sie mit dem nächsten Kapitel fortfahren!

3 Nutzung von Variablen

Variablen bilden einen sehr wichtigen Aspekt jeder Programmiersprache. Variablen sind eine Art Behälter/Container - in denen Werte gespeichert werden können. Wir können Variablen an verschiedenen Stellen im Programm nutzen, und wir können Variablen ändern. Lassen Sie uns die Verwendung von Variablen anhand eines Beispiels veranschaulichen.

Eine Spirale fahren

Angenommen, wir wollen, dass das obige Programm abändern, sodass der Roboter eine Spirale fährt. Dies können wir erreichen, indem wir den Wert von MOVE_TIME, also der Zeit in der der Roboter eine Drehung macht, zwischen jeder Bewegung erhöhen. Aber wie können wir das tun? Wir haben MOVE_TIME also eine Konstante definiert, deren Wert daher nicht verändert werden kann. Statt einer Konstante benötigen wir nun eine Variable. Variablen können einfach in NXC definiert werden. Hier ist das Beispielprogramm:

```
#define TURN_TIME 360

int move_time;            // Variablendefinition

task main()
{
  move_time = 200;        // Initialisierung der Variable
  repeat(50)
  {
    OnFwd(OUT_AC, 75);
    Wait(move_time);      // Nutzen der Variable
    OnRev(OUT_C, 75);
    Wait(TURN_TIME);
    move_time += 200;     // Erhöhung des Variablenwerts
  }
  Off(OUT_AC);
}
```

Die interessanten Zeilen wurden mit Kommentaren versehen. Zuerst definieren wir eine Variable, indem wir das Stichwort **`int`** gefolgt von einem Namen wählen. (In der Regel werden Kleinbuchstaben für Variablennamen und Großbuchstaben für Konstanten vergeben). Der Name der Variablen muss mit einem Buchstaben beginnen, nachfolgend können auch Ziffern und Unterstriche verwendet werden. Andere Symbole sind nicht erlaubt. (Das gleiche gilt für Namen von Konstanten, Tasks, usw.) Das Wort **`int`** steht für Ganzzahl. Nur ganze Zahlen können in **`int`** Variablen gespeichert werden[5]. Mit Initialisierung einer Variable wird die Wertzuweisung (move_time = 200) bezeichnet.

In der dritten Zeile der Task `main` wird der Wert der Variable auf 200 gesetzt (die Variable wir initialisiert). Wird nun die Variable benutzt (wie in Zeile 6 und Zeile 8) besitzt diese den Wert 200. Jetzt folgt die repeat-Schleife in der die Variable benutzt wird um den Roboter Vorwärts fahren zu lassen. Am Ende der Schleife erhöhen wir den Wert der Variable um 200 (move_time += 200[6]). Beim ersten Schleifendurchlauf fährt der Roboter 200ms geradeaus. Beim zweiten Durchlauf 400ms, beim dritten Durchlauf 600ms usw.

Neben dem Hinzufügen von Werten zu einer Variable kann man Variablen auch

- multiplizieren *=
- subtrahieren -=
- teilen / =

(Beachten Sie, dass bei der Teilung einer int-Ganzzahl, das Ergebnis auf die nächste ganze Zahl gerundet wird z.B. 3,4 ist 3; 3,5 ist 4).

Sie können auch mehrere Variablen miteinander verrechnen, sowie kompliziertere Ausdrücke berechnen. Das nächste Beispiel hat keine Wirkung auf den Roboter, es dient lediglich der Veranschaulichung, wie Variablen verwendet werden können.

[5] In NXC gibt es keine Kommazahlen, alle Divisionen werden deshalb ab- bzw. aufgerundet.
[6] Statt des angegebenen Ausdrucks hätte man auch: move_time = move_time + 200; schreiben können

```
int aaa;
int bbb,ccc;
int values[];

task main()
{
  aaa = 10;
  bbb = 20 * 5;
  ccc = bbb;
  ccc /= aaa;
  ccc -= 5;
  aaa = 10 * (ccc + 3);          // aaa is now equal to 80
  ArrayInit(values, 0, 10);      // allocate 10 elements = 0
  values[0] = aaa;
  values[1] = bbb;
  values[2] = aaa*bbb;
  values[3] = ccc;
}
```

Hinweis zu den ersten beiden Zeilen: es können auch mehrere Variablen in einer Zeile definiert werden. Wir hätten auch alle drei Variablen hintereinander schreiben und somit definieren können. Die Variable namens `values` stellt ein sogenanntes Array dar. Als Array werden Variablen bezeichnet, die mehre Zahlen bzw. Werte aufnehmen können. In NXC werden Integer-Arrays wie folgt deklariert:

`int name[];`

Der nachfolgende Ausdruck allokiert 10 Elemente und initialisiert diese mit dem Wert 0 (alle 10 Variablen werden mit 0 initialisiert).

```
ArrayInit(values, 0, 10);
```

Das obige Programmbeispiel zeigt wie einzelnen Elemente eines Arrays über Indizes direkt angesprochen werden. Wir haben eine Zeile noch mal explizit herausgestellt um dies zu verdeutlichen.

```
values[1] = bbb; // Der zweiten Variable wird der Wert bbb zugewiesen
```

Zufallszahlen

In allen bisher aufgeführten Programmen haben wir genau definiert, was der Roboter tun soll. Aber die Dinge werden oftmals viel interessanter, wenn man nicht genau weiß was der Roboter tun wird. Wir wollen einige Zufälligkeiten in den Bewegungen des Roboters einbauen.
Mit NXC können Sie zufällige Zahlen erzeugen. Das folgende Programm verwendet Zufallszahlen, um den Roboter „beliebig“ umher fahren zu lassen. Der Roboter fährt eine zufällige Zeit geradeaus und macht anschließend eine zufällige Drehung[7].

```
int move_time, turn_time;
task main()
{
  while(true)
  {
    move_time = Random(600);
    turn_time = Random(400);
    OnFwd(OUT_AC, 75);
    Wait(move_time);
    OnRev(OUT_A, 75);
    Wait(turn_time);
  }
}
```

Das Programm definiert zwei Variablen, und weist diesen Zufallszahlen zu. Die Funktion `Random` (`600`) erzeugt eine Zufallszahl zwischen 0 und 600 (die maximal Werte 0 und 600 sind nicht im Zufallsbereich enthalten). Jedes Mal wenn die Funktion `Random` aufgerufen wird unterscheiden sich die Zahlen.

[7] Die Richtung der Kurve; also Links- bzw. Rechtskurve ist allerdings durch die Ansteuerung der Motoren festgelegt und somit nicht zufällig.

Beachten Sie, dass wir bei der Veränderung des Ausdrucks: Wait (Random (600)) die Verwendung von Variablen hätten vermeiden können.

Im soeben eingeführten Beispielprogramm, haben Sie auch einen neuen Schleifen-Typ kennen gelernt. Anstatt der Verwendung von **`repeat`** haben wir die `while(true)` Anweisung benutzt. Die while-Schleife wird so lange wiederholt, wie die Bedingung zwischen den Klammern ist wahr ist. Eine while-Schleife mit der Bedingung true (=wahr) bricht nie ab[8], man spricht hier auch von einer Endlosschleife. Mehr über while-Schleifen erfahren Sie in Kapitel 4.

Zusammenfassung

In diesem Kapitel haben Sie etwas über die Verwendung von Variablen und Arrays erfahren. Sie können auch noch andere Datentypen wie int deklarieren; z.B. short, long, byte, bool und String.
Sie haben auch gelernt, wie man Zufallszahlen erzeugt und verwendet, so dass Sie dem Roboter ein gewisses Maß an unvorhersehbarem Verhalten verleihen. Schließlich haben wir die Verwendung der while-Anweisung, behandelt um eine Endlosschleife zu programmieren.

[8] Eine Abbruchbedingung kann durch drücken des dunkelgraue NXT-Knopfs erzeugt werden.

4 Kontrollstrukturen

In den vorangegangenen Kapiteln haben wir die Schleifenkonstrukte repeat und while kennen gelernt. Diese Anweisungen kontrollieren die Art und Weise, wie andere Befehle/Anweisungen innerhalb eines Programms ausgeführt werden. Derartige Konstrukte bezeichnen wir als "Kontrollstrukturen". In diesem Kapitel werden wir weitere Kontrollstrukturen kennen lernen.

Die If-Anweisung

Manchmal möchte man, dass ein Teil des Programms nur in bestimmten Situationen ausgeführt wird. In diesem Fall wird die if-Anweisung verwendet. Lassen Sie uns ein konkretes Beispiel nehmen. Wir werden wieder das bisherige Programm verwenden, und um die if-Anweisung erweitern.
Wir wollen, dass der Roboter einer geraden Linie entlang fährt und dann entweder links oder rechts abbiegt. Dazu brauchen wir wieder zufällige Zahlen. Wir nehmen eine Zufallszahl, die entweder positiv oder negativ sein kann. Ist die Zahl kleiner oder gleich 0 (<=0) biegen wir rechts ab, anderenfalls biegen wir links ab. Hier nun das Beispielprogramm:

```
#define MOVE_TIME 500
#define TURN_TIME 360

task main()
{
  while(true)
  {
    OnFwd(OUT_AC, 75);
    Wait(MOVE_TIME);

    if (Random() <= 0)    // if-Verzweigung im Programm
    {
      OnRev(OUT_A, 75);
    }
    else
    {
      OnRev(OUT_C, 75);
    }
    Wait(TURN_TIME);
  }
}
```

Die if-Anweisung ähnelt auf dem ersten Blick der while-Anweisung. Wenn die Bedingung zwischen den Klammern wahr ist, wird der Teil zwischen den geschweiften Klammern ausgeführt. Andernfalls wird der Teil zwischen den geschweiften Klammern nach dem Wort `else` ausgeführt.
Was wird eigentlich gemacht? Die Funktion `Random` () <= 0 bedeutet, dass der mit Random erzeugte Zufallswert kleiner oder gleich 0 sein muss, damit die Bedingung *wahr* wird. Andernfalls (*else*) „überspringt“ das Programm die zur if-Anweisung gehörigen Klammern und führt den else-Teil aus. Werte und Zahlen können auf unterschiedlicher Weise miteinander verglichen werden. Hier die wichtigsten:

==	Vergleich ob zwei Werte exakt gleich sind[9]
<	Kleiner als
<=	Kleiner gleich
>	Größer als
>=	Größer gleich
!=	ungleich

Es können auch Bedingungen wie &&, was soviel bedeutet wie "und" oder | |, was soviel bedeutet wie "oder". Benutzt werden. Hier sind einige Beispiele:

true	immer wahr
false	nie wahr
xyz !=3	wahr wenn xyz ungleich 3 ist
(xyz >=5) && (xyz <= 10)	wahr wenn xyz kleiner 5 und kleiner 10 ist
(thc == 10) || (mfg == 10)	wahr wenn entweder thc und/oder mfg gleich 10 ist

[9] Diesen Ausdruck nicht mit = verwechseln. Der Ausdruck = weist einer Variablen einen neuen Wert zu.

Beachten Sie bitte, dass die if-Anweisung aus zwei Teilen besteht. Der erste Teil, unmittelbar nach der Bedingung, wird ausgeführt wenn die Bedingung wahr ist. Der zweite Teil (else-Teil) wird ausgeführt, wenn die Bedingung falsch ist. Das Schlüsselwort else und der sich daran anschließende Teil ist optional. Soll beispielsweise der Roboter keine Aktion durchführen, falls die if-Anweisung falsch ist, kann der else-Teil einfach entfallen.

Die do-Anweisung (do-while Schleife)

Es gibt noch andere Kontrollstrukturen, z.B. die do-Anweisung. Diese weist folgende Form auf:

```
do
{
  statements;
}
while (condition);
```

Die Anweisungen zwischen den geschweiften Klammern wird solange ausgeführt, wie die Bedingung der while-Schleife wahr ist. Die do-Anweisung hat die gleiche Form wie in der if-Anweisung oben beschrieben. Hier ein Beispielprogramm. Der Roboter fährt 20 Sekunden (zufällig) umher. Sind 20 Sekunden (`while (total_time < 20000);`) erreicht stoppt er.

```
int move_time, turn_time, total_time; //Variablendeklaration

task main()
{
  total_time = 0;         // Initialisierung von total_time
  do                      // Beginn der do-Anweisung
  {
    move_time = Random(1000);
    turn_time = Random(1000);
    OnFwd(OUT_AC, 75);
    Wait(move_time);
    OnRev(OUT_C, 75);
    Wait(turn_time);
    total_time += move_time;
    total_time += turn_time;
  }
  while (total_time < 20000); // Prüfen der Bedingung
  Off(OUT_AC);
}
```

Beachten Sie bitte, dass sich die do-Anweisung ähnlich der while-Anweisung verhält. Während allerdings in der while-Anweisung die Vorraussetzung vor dem Ausführen der Anweisungen geprüft wird, geschieht dies bei der do-Anweisung erst nach dem ersten „Durchlauf“. Die Anweisungen innerhalb der geschweiften Klammern werden somit mindestens einmal abgearbeitet.

Zusammenfassung

In diesem Kapitel haben wir zwei neue Strukturen kennen gelernt: die if-Anweisung und die do-Anweisung. Zusammen mit der repeat-Anweisung und der while-Anweisung, kennen wir somit mittlerweile vier Kontrollstrukturen. Es ist sehr wichtig, dass Sie verstehen, was diese Kontrollstrukturen wirklich tun. Also versuchen Sie, sich selbst noch einige weitere Beispiele auszudenken, bevor Sie mit dem nächsten Kapitel fortfahren.

5 Sensoren

Natürlich können Sie auch Sensoren an Ihrem NXT anschließen, um den Roboter auf seine Umwelt reagieren zu lassen. Bevor wir damit loslegen, müssen Sie aber noch ein paar Veränderungen an Ihrem Roboter vornehmen. Als erstes bauen Sie den Berührungssensor an Ihrem NXT an. Wie schon beim Zusammenbau des Tribot folgen Sie bitte der LEGO MINDSTORMS Bauanleitung. Ihr Roboter sollte nun wie folgt aussehen:

Schließen Sie den Touch-Sensor an Port (Eingang) 1 am NXT an.

Auf Sensor(-Werte) warten

Beginnen wir mit einem sehr einfachen Programm, in dem der Roboter vorwärts fährt, bis er auf einen Gegenstand trifft. Hier ist es:

```
task main()
{
  SetSensor(IN_1,SENSOR_TOUCH);//Initialisierung des Sensors
  OnFwd(OUT_AC, 75);
  until (SENSOR_1 == 1);
  Off(OUT_AC);
}
```

Es gibt zwei wichtige Zeilen. Die erste Zeile des Programms sagt dem Roboter, welche Art von Sensor wir verwenden. `IN_1` ist die Nummer des Eingangs, an dem wir unseren Berührungssensor angeschlossen haben. Die anderen Sensoreingänge sind `IN_2, IN_3` und `IN_4`. `SENSOR_TOUCH` weist darauf hin, dass es sich hierbei um einen Berührungssensor (engl. Touch-Sensor) handelt. Für den Lichtsensor würden wir `SENSOR_LIGHT` schreiben. Nachdem wir den Sensor-Typ spezifiziert haben, schaltet das Programm die Motoren an und der Roboter beginnt sich zu bewegen. Die nächste Anweisung ist ein sehr nützlicher Befehl, es handelt sich hierbei um die until-Anweisung. Diese weist das Programm an, solange zu warten (der Roboter bewegt sich währenddessen weiter vorwärts) bis der Ausdruck innerhalb der Klammern wahr ist. Diese Bedingung sagt, dass der Wert des Sensors (`SENSOR_1`) =1 sein muss, was dann erreicht wird, wenn der Tastsensor gedrückt wird bzw. gedrückt ist. Solange der Sensor nicht gedrückt wird, liefert der Tastsensor den Wert 0. Die until-Anweisung wartet also solange, bis der Tastsensor gedrückt wird. Anschließend werden beide Motoren ausgeschaltet und das Programm wird beendet.

Auf den Tastsensor reagieren

Versuchen wir nun, den Roboter Hindernissen ausweichen zu lassen. Wenn der Roboter gegen ein Objekt stößt, lassen wir ihn ein wenig rückwärts fahren und eine Drehung vollziehen. Anschließend setzen wir die Fahrt fort. Hier ist das Beispielprogramm:

```
task main()
{
  SetSensorTouch(IN_1);   //Initialisierung des Sensors
  OnFwd(OUT_AC, 75);
  while (true)            //Endlosschleife
  {
    if (SENSOR_1 == 1)    //Abfragen des Sensors
    {
      OnRev(OUT_AC, 75); Wait(300);
      OnFwd(OUT_A, 75); Wait(300);
      OnFwd(OUT_AC, 75);
    }
  }
}
```

Wie in den vorherigen Beispielen haben wir zunächst die Art des Sensors initialisiert. Anschließend fährt der Roboter vorwärts. In der Endlosschleife prüfen wir, ob der Tastsensor (SENSOR_1) berührt wurde. Wenn ja fährt der Roboter für 300ms zurück, biegt danach für 300ms rechts ab und fährt dann wieder geradeaus.

Lichtsensor

Neben dem Tastsensor enthält das LEGO MINDSTORM NXT Set auch einen Lichtsensor, einen Soundsensor und einen (digitalen) Ultraschallsensor. Der Lichtsensor kann in einen aktiven und einen passiven Modus versetzt werden. Im aktiven Modus emittiert eine LED-Diode rotes Licht. Im passiven Modus ist die LED-Diode ausgeschaltet. Der aktive Modus kann bei der Messung des reflektierten[10] Lichts nützlich sein, wenn der Roboter beispielsweise einer Linie auf dem Boden folgt. Dies werden wir im nächsten Beispielprogramm versuchen. Um das folgende Experimente angehen zu können, statten Sie Ihren Tribot mit einem Lichtsensor aus. Verbinden Sie den Lichtsensor mit Eingang 3, Soundsensor mit Eingang 2 und den Ultraschallsensor auf Eingang 4, wie in der mitgelieferten Bauanleitung beschrieben.

[10] Das rote Licht der LED-Diode wird von Oberflächen reflektiert. Je näher ein Objekt bzw. die Oberfläche desto sinnvoller kann es sein, den Lichtsensor im aktiven Modus zu nutzen.

Für das Linienfolger-Experiment benötigen wir die Testunterlage des NXT-Sets, auf dieser ist eine schwarze Linie aufgedruckt. Das Grundprinzip des Linienfolgens ist es, dass der Roboter immer versucht an der Grenze/Kante der schwarzen Linie zu bleiben. Ist der Roboter bspw. rechts dieser Kante (also über einer weisen Fläche) dreht er solange nach links bis der Lichtsensor die schwarze Linie bzw. schwarz detektiert. Anschließend dreht der Roboter wieder nach rechts und „sucht" die weiße Fläche. Durch diese Drehbewegungen hangelt sich der Roboter entlang der (schwarz(weißen) Kante, nach vorne. Hier ist ein sehr einfaches Beispielprogramm zum Folgen einer Linie. Hinweis: das Programm nutzt einen Schwellenwert für das Linienfolgen.

```
#define SPEED 60
#define motoren OUT_BC
#define THRESHOLD 45
task main ()
{
  SetSensorLight(IN_3);
  OnFwd(motoren, SPEED);
  while(true)
  {
    if(SENSOR_3 < THRESHOLD)
    {
      Off(OUT_B);
      OnFwd(OUT_C, SPEED);
    }
    else
    {
      Off(OUT_C);
      OnFwd(OUT_B, SPEED);
    }
  }
}
```

Zuerst wird Eingang 3 als Lichtsensor konfiguriert. Als nächstes werden die Motoren des Roboters ein und auf Vorwärts gestellt. Innerhalb der Endlosschleife wird der Lichtsensor abgefragt. Ist der Wert des Lichtsensors größer als der Schwellenwert (hier 40)[11] dreht sich der Roboter für 100ms `Wait(100);` nach rechts `OnRev(OUT_C, 75);` Diese beiden Befehle werden so lange ausgeführt, bis der Lichtsensor einen niedrigeren Wert als den in der Konstanten THRESHOLD angegebenen Schwellenwert misst `until(Sensor(IN_3) <= THRESHOLD);` Wird nun ein Wert unterhalb des Werts `40` gemessen, dreht der Roboter nach links, bis der gemessene Wert wieder über den als THRESHOLD definierten Wert ist. Anschließend beginnt das „Spiel" von vorne.

Wie Sie sehen können, sind die Bewegungen des Roboters nicht sehr flüssig. Sie können es zusätzlich noch mit einem `Wait(100);` Befehl nach `OnFwd(OUT_A, 75);` ergänzen.

Zum messen des Umgebungslicht im passiven Modus, mit ausgeschaltem LED Licht – konfigurieren Sie den Sensor wie folgt:

```
SetSensorType(IN_3,IN_TYPE_LIGHT_INACTIVE);
SetSensorMode(IN_3,IN_MODE_PCTFULLSCALE);
ResetSensor(IN_3);
```

Sound Sensor

Mit dem Soundsensor[12] können Sie Ihren NXT auf Geräusche reagieren lassen! Wir werden ein Programm schreiben, das auf ein lautes Geräusch wartet. Der Roboter bewegt sich anschließend und stoppt falls der Soundsensor erneut ein lautes Geräusch wahrnimmt. Schließen Sie bitte den Soundsensor, wie in der Tribot-Bauanleitung beschrieben, an den Eingang 2 an.

[11] Wir verwenden hier eine Konstante, diese kann somit leicht an das Umgebungslicht angepasst werden. Jedes Mal wenn sich die Lichtverhältnisse ändern z.B. Deckenbeleuchtung, Sonneneinwirkung etc. muss ggf. der Schwellenwert angepasst werden.

[12] Soundsensor ist streng genommen nicht richtig übersetzt, da der Sensor selbst keine Töne erzeugen kann, sondern nur Geräusche detektiert. Der Soundsensor ist demnach eher ein „db-Meter"

Beispielprogramm: Wenn ein lautes Geräusch auftritt, fährt der Roboter geradeaus, bis er von einem weiteren Geräusch gestoppt wird.

```
#define THRESHOLD 40
#define MIC SENSOR_2

task main()
{
  SetSensorSound(IN_2);
  while(true)
  {
    until(MIC > THRESHOLD);
    OnFwd(OUT_AC, 75);
    Wait(300);
    until(MIC > THRESHOLD);
    Off(OUT_AC);
    Wait(300);
  }
}
```

Wir definierten zunächst einen Schwellenwert (THRESHOLD Konstante) und einen Platzhalter für `SENSOR_2`. In der `main` Task initialisieren wir den Soundsensor auf Port/Ausgang 2, anschließend starten wir eine Endlosschleife.

Mit der until-Anweisung, wird das Programm angewiesen so lange zu warten, bis die gemessene Lautstärke größer ist als der von uns festgelegte Schwellenwert. Hinweis: `SENSOR_2`, ist hier nicht nur ein Name, sondern auch ein Makro, der die Werte des Soundsensor zurück liefert.

Der `Wait` –Befehl wurde eingefügt, da ansonsten der Roboter ständig starten und stoppen würde. Der Prozessor des NXT ist in der Tat so schnell, dass das abarbeiten der Befehle zwischen den beiden until-Anweisungen nur wenige Millisekunden in Anspruch nimmt. Sie können gerne versuchen, die beiden `Wait` –Befehle auszukommentieren, Versuchen Sie es! Eine Alternative zu der Verwendung der until-Anweisung um auf das Eintreten eines Ereignisses (hier lautes Geräusch) zu warten, wäre die while-Anweisung. Hierfür würde es genügen, die Bedingung innerhalb der Klammern folgendermaßen zu ändern:

```
while(MIC <= THRESHOLD)
```

Viel mehr gibt es über die analogen NXT Sensoren nicht zu wissen. Sie sollten sich nur daran erinnern, dass die Sensoren für Licht und Sound Werte zwischen 0 und 100 zurückgeben.

Ultraschallsensor

Der Ultraschallsensor arbeitet ähnlich einem Sonar: Man könnte es folgendermaßen beschreiben: der Sensor schickt (kegelförmige) Schallwellen aus und misst die Zeit, die sie benötigen um von einem Objekt zurückgeworfen zu werden. Der Ultraschallsensor ist ein digitaler Sensor, dies bedeutet, dass er direkt die Sensorwerte analysiert und diese an den NXT weitergibt. Mit diesem Sensor kann Ihr Roboter „sehen“ und somit Hindernisse erkennen ehe er dagegen fährt, wie dies bspw. mit dem Tastsensor der Fall ist.

```
#define NEAR 15     //Definieren ab welchem Abstand auf ein
                    //Hindernis reagiert wird; in cm
task main()
{
  SetSensorLowspeed(IN_4);
  while(true)
  {
    OnFwd(OUT_AC,50);
    while(SensorUS(IN_4)>NEAR);
    Off(OUT_AC);
    OnRev(OUT_C,100);
    Wait(800);
  }
}
```

Wir initialisieren den Ultraschallsensor auf Ausgang 4. Innerhalb der Endlosschleife lassen wir den Roboter so lange gerade aus fahren, bis der Ultraschallsensor ein Hindernis unterhalb des Abstands von 15cm erkennt. Anschließend stoppt der Roboter kurz, führt eine Ausweichbewegung durch und fährt weiter.

Zusammenfassung

In diesem Kapitel haben wir gesehen, wie wir die im NXT-Set enthaltenen Sensoren benutzen und einsetzen können. Ebenso haben wir den sinnvollen Einsatz von until- und while-Anweisungen im Zusammenhang mit Sensoren kennen gelernt.
Ich empfehle Ihnen mehrere unterschiedliche Programme zu schreiben. Sie haben nun alle nötigen Kenntnisse, um Ihrem Roboter ein komplexes Verhalten zu verleihen.

6 Tasks und Subroutinen

Bis jetzt bestanden alle unsere Programme aus einem einzigen Task. NXC Programme können aber auch aus mehreren Tasks bestehen. Es ist ebenso möglich, Programmteile in so genannten Subroutinen „auszulagern“, die wiederum an unterschiedlichen Stellen innerhalb eines Programms verwendet werden können. Die Verwendung von Task und Subroutinen macht Ihr Programm übersichtlicher, verständlicher und auch kompakter. In diesem Kapitel werden wir uns mit den verschiedenen Einsatzmöglichkeiten von Tasks und Subroutinen befassen.

Tasks

Ein NXC Programm kann maximal 255 Tasks beinhalten. Jede dieser Tasks muss ihren eigenen Namen haben. Der main-Task muss immer vorhanden sein, da es der Task ist, der vom Roboter als erstes ausgeführt wird. Die anderen Tasks können nur von einem „laufenden“ Task, aufgerufen werden. Der main-Task muss ausgeführt werden bevor andere Tasks starten können. Tasks können auch parallel/gleichzeitig, ausgeführt werden.

Lassen Sie uns die Verwendung von Tasks näher betrachten. Wir wollen ein Programm schreiben, in dem der Roboter ein Quadrat fährt. Dieses Mal soll der Roboter aber auch auf Hindernisse reagieren können. Diese beiden Eigenschaften in einer einzigen Task zu programmieren ist nicht gerade einfach, da der Roboter zwei Sachen zur gleichen Zeit machen muss: Ein Quadrat fahren (das heißt, Motoren An/Aus Vor und Zurück schalten) und die Sensoren abfragen. Es empfiehlt sich deshalb, zwei Tasks zu verwenden. Der erste Task für das Quadrat, der zweite Task für das Abfragen der Sensoren. Hier ist das Beispielprogramm:

```
mutex moveMutex;

task move_square()
{
  while (true)
  {
    Acquire(moveMutex);
    OnFwd(OUT_AC, 75); Wait(1000);
    OnRev(OUT_C, 75); Wait(500);
    Release(moveMutex);
  }
}

task check_sensors()
{
  while (true)
  {
   if (SENSOR_1 == 1)
   {
     Acquire(moveMutex);
     OnRev(OUT_AC, 75); Wait(500);
     OnFwd(OUT_A, 75); Wait(500);
     Release(moveMutex);
   }
  }
}

task main()
{
  Precedes(move_square, check_sensors);
  SetSensorTouch(IN_1);
}
```

Der main-Task initialisiert den Sensor-Typ, startet die Tasks `check_sensors` und `move_square` und fügt beide Tasks in die Warteschlange des Prozessors, danach endet die main-Task. Der Task `move_square` veranlasst den Roboter endlos ein Quadrat zu fahren. Der Task `check_sensors` kontrolliert ob der Tastsensor betätigt wurde und wenn ja, macht der Roboter eine Ausweichbewegung.

Es ist sehr wichtig, sich daran zu erinnern, dass beide Aufgaben im selben Augenblick beginnen und dass dies, zu einem unerwarteten Verhalten des Roboters führen kann, wenn beide Tasks versuchen, die Motoren zu steuern.

Um diese Probleme zu vermeiden, wird eine etwas „merkwürdige“ Arte einer Variablen deklariert - Mutex (mutex ist ein Akronym und steht für eine gegenseitige (**mut**ual) Ausgrenzung (**ex**klsuion)). Auf diese Variablenart kann nur mit den Funktionen *Acquire* und *Release* zugegriffen werden. Durch diese beiden Funktionen wird sichergestellt, dass auf „kritische“ Programmzeilen, die zwischen *Acquire* und *Release* stehen, nur von einem Task (zu einer Zeit) zugegriffen werden kann, beispielsweise Motorenbefehle.

Diese Mutex-Variablen sind sogenannte Semaphore[13]. Die Programmierung dieser Technik wird auch nebenläufige Programmierung genannt. Auf diese Art der Programmierung wird in Kapitel 10 im Detail eingegangen.

Subroutinen

Manchmal benötigen Sie das gleiche Codefragment, z.B. das Fahren einer 90° Kurve - an mehreren Stellen in Ihrem Programm. In diesem Fall können Sie das Codefragment in eine Subroutine (engl. für Unterprogramm) auslagern und diesem Unterprogramm einen Namen geben. Jetzt können Sie diese Programmzeilen einfach durch Aufruf, des (Namens) Unterprogramms, an einer beliebigen Stelle im Programm ausführen. Ein Beispielprogramm:

```
sub turn_around(int pwr)        //Anlegen einer Subroutine
{
  OnRev(OUT_C, pwr); Wait(900);
  OnFwd(OUT_AC, pwr);
}

task main()
{
  OnFwd(OUT_AC, 75);
  Wait(1000);
  turn_around(75);               //Aufruf der Subroutine
  Wait(2000);
  turn_around(75);               //Aufruf der Subroutine
  Wait(1000);
  turn_around(75);               //Aufruf der Subroutine
  Off(OUT_AC);
}
```

In diesem Programm haben wir eine Subroutine Namens `turn_around` definiert. Diese veranlasst den Roboter sich um seinen Mittelpunkt zu drehen. In der main-Task wird die Subroutine dreimal aufgerufen. Beachten Sie, dass der Aufruf der Subroutine durch schreiben des Namens (hier `turn_around`) gefolgt von einer Übergabe eines Parameters (dieser steht in Klammern; hier `75`) erfolgt. Wenn einer Subroutine keine Parameterübergabe benötigt, enthält die Klammer keine Argumente z.B. `turn_around()`.

Der große Vorteil von Subroutinen ist, dass sie nur einmal im (Flash-) Speicher des NXT gespeichert werden und somit Speicherplatz gespart wird. Ist die Subroutine sehr kurz, ist es empfehlenswert sogenannte Inline-Funktionen zu benutzen. Diese werden nicht separat gespeichert sondern (während des Kompilierens) an die benötigten Programmstellen kopiert. Dies benötigt zwar mehr Speicherplatz, dafür gibt es aber keine Begrenzung, wie viele Inline-Funktionen ein Programm enthalten kann. Inline-Funktionen werde wie folgt deklariert:

```
inline int Name( Args )
{
  //body;
  return x*y;
}
```

Definition und Aufruf von Inline-Funktionen ist die gleiche wie bei Subroutinen. Das obige Beispielprogramm nun mit einer Inline-Funktion:

[13] Laut Wikipedia: Semaphor (Informatik), eine Datenstruktur zur Synchronisation von Prozessen [Juli, 2009]

```
inline void turn_around() //Deklaration der Inline-Funktion
{                         // ohne Parameter
  OnRev(OUT_C, 75);
  Wait(900);
  OnFwd(OUT_AC, 75);
}

task main()
{
  OnFwd(OUT_AC, 75);
  Wait(1000);
  turn_around();      //Aufruf der Inline-Funktion
  Wait(2000);
  turn_around();      //Aufruf der Inline-Funktion
  Wait(1000);
  turn_around();      //Aufruf der Inline-Funktion
  Off(OUT_AC);
}
```

Das obige Beispiel, können wir erweitern, indem wir die Zeit für die Drehung (`turntime`), und die Geschwindigkeit (`pwr`) als Argumente beim Aufruf der Inline-Funktion übergeben, wie im folgenden Beispielprogramm:

```
inline void turn_around(int pwr, int turntime)
{
  OnRev(OUT_C, pwr);
  Wait(turntime);
  OnFwd(OUT_AC, pwr);
}

task main()
{
  OnFwd(OUT_AC, 75);
  Wait(1000);
  turn_around(75, 2000);  //Aufruf der Inline-Funktion
  Wait(2000);
  turn_around(75, 500);
  Wait(1000);
  turn_around(75, 3000);
  Off(OUT_AC);
}
```

Beachten Sie, dass in der Klammer hinter dem Namen der Inline-Funktion, das/die Argument (e) der Funktion stehen. Im Beispielprogramm sind diese Parameter als Integer (Ganzzahlen) deklariert worden. Es können aber auch andere Variablen-Typen (z.B. char, long etc.) deklariert werden. Wenn mehrere Parameter übergeben werden, müssen diese durch Komma getrennt werden. Beachten Sie, dass in NXC, **sub** das gleiche ist wie **void**. Funktionen können aber auch andere Rückgabetypen z.B. integer, string haben.

```
inline int turn_around(int pwr, int turntime)
{
  int turn = int pwr + int turntime
  return turn;
}

task main()
{
  OnFwd(OUT_AC, 75);
  Wait(turn_around(75, 200);)  //Aufruf der Inline-Funktion
  Off(OUT_AC);
}
```

Weitere Informationen finden Sie im Leitfaden NXC.

Makros Definieren

Es gibt noch einen weiteren Weg Codefragmenten einen Namen zu geben. In NXC können Makros (nicht zu verwechseln mit den Makros in BricxCC) definiert werden. Wie wir bereits gesehen haben, können Konstanten wie folgt definiert werden: #KONSTANTE Nun können wir aber jedes beliebige Codefragment wie folgt definieren:

```
#define turn_around \
  OnRev(OUT_B, 75); Wait(3400);OnFwd(OUT_AB, 75);

task main()
{
  OnFwd(OUT_AB, 75);
  Wait(1000);
  turn_around;
  Wait(2000);
  turn_around;
  Wait(1000);
  turn_around;
  Off(OUT_AB);
}
```

Nach der #define-Anweisung `turn_around` steht das Wort `turn_around` für die darauf folgende Zeile. Wann immer jetzt `turn_around`, geschrieben wird, wird dies durch
`OnRev(OUT_B, 75); Wait(3400);OnFwd(OUT_AB, 75);`
ersetzt. Beachten Sie, dass der Text in eine Zeile geschrieben wird. (Es gibt auch Möglichkeiten, eine # define-Anweisung über mehrere Zeilen zu schreiben, dies wird aber nicht empfohlen.)

Define-Anweisungen sind tatsächlich sehr viel mächtiger. Ihnen können auch Parameter übergeben werden. Zum Beispiel können wir die Drehzeit als Parameter übergeben. Hier ist ein Beispiel, in dem wir vier Makros, für vorwärts fahren, für rückwärts fahren, für eine Links- und eine Rechtsdrehung definieren. Jeder dieser Define-Anweisung hat zwei Argumente: die Geschwindigkeit (s) und die Zeit (t).

```
#define turn_right(s,t) \
  OnFwd(OUT_A, s);OnRev(OUT_B, s);Wait(t);
#define turn_left(s,t) \
  OnRev(OUT_A, s);OnFwd(OUT_B, s);Wait(t);
#define forwards(s,t) OnFwd(OUT_AB, s);Wait(t);
#define backwards(s,t) OnRev(OUT_AB, s);Wait(t);

task main()
{
  backwards(50,10000);
  forwards(50,10000);
  turn_left(75,750);
  forwards(75,1000);
  backwards(75,2000);
  forwards(75,1000);
  turn_right(75,750);
  forwards(30,2000);
  Off(OUT_AB);
}
```

Zusammenfasssung

In diesem Kapitel haben Sie die Verwendung von Tasks, Subroutinen, Inline-Funktionen und Makros kennen gelernt. Tasks werden üblicherweise verwendet, wenn verschiedene Prozesse (Aufgaben, Berechnungen, Abfragen etc), zur gleichen Zeit erfolgen sollen. Subroutinen sind nützlich, wenn größere Teile des Codes an verschiedenen Stellen im Programm verwendet werden. Inline-Funktionen sind nützlich, wenn kleinere Teile des Codes an vielen verschiedenen Stellen und in verschiedenen Tasks verwendet werden sollen, allerdings benötigen diese mehr Speicher als Subroutinen. Schließlich Makros, diese sind, ebenso wie Inline-Funktionen nützlich, wenn kleinere Codefragmente Verwendung finden. Wenn Sie dieses Tutorial bis hierher durchgearbeitet haben, können Sie Ihrem Roboter schon sehr komplexe Sachen machen lassen. Die weiteren Kapitel gehen nun auf spezielle Anwendungsfälle ein.

7 Musik komponieren

Der NXT besitzt einen eingebauten Lautsprecher, dieser kann sowohl Töne, als auch Musikdateien wiedergeben. Dies ist vor allem dann nützlich, wenn der NXT Ihnen mitteilen soll, dass etwas (spezielles z.B. das Eintreten eines bestimmten Ereignisses) eingetreten ist. Es kann aber auch sonst witzig sein, wenn Ihr Roboter Musik macht oder spricht, während er gerade umher fährt.

Abspielen von Musikdateien

BricxCC besitzt ein Konvertierungsprogamm, welches .wav-Dateien in für den NXT verwendbare .rso-Dateien konvertiert. Dies kann über das Menü *Tools* → *Sound conversion* ausgeführt werden.

Mit einem weiteren Programm (NXT memory browser) können die .rso-Dateien auf den NXT übertragen und dort gespeichert werden. Dieses kann über das Menü *Tools* → *NXT explorer* ausgeführt werden.
Mit dem folgenden Befehl können die .rso-Dateien aus einem Programm heraus abgespielt werden:

```
PlayFileEx(filename, volume, loop?)
```

Die Parameter sind filename (Dateiname), volume (Lautstärke), von 0 bis 4 und loop (Schleife). Der letzte Parameter wird auf 1 (TRUE) gesetzt, falls die Datei erneut abgespielt werden soll oder auf 0 (FALSE), falls die Musikdatei ein einziges Mal gespielt werden soll.

```
#define TIME 200
#define MAXVOL 7
#define MINVOL 1
#define MIDVOL 3
#define pause_4th Wait(TIME)
#define pause_8th Wait(TIME/2)
#define note_4th \
  PlayFileEx("! Click.rso",MIDVOL,FALSE); pause_4th
#define note_8th \
  PlayFileEx("! Click.rso",MAXVOL,FALSE); pause_8th

task main()
{
  PlayFileEx("! Startup.rso",MINVOL,FALSE);
  Wait(2000);
  note_4th;
  note_8th;
  note_8th;
  note_4th;
  note_4th;
  pause_4th;
  note_4th;
  note_4th;
  Wait(100);
}
```

Dieses nette Programm spielt zuerst die Start-Melodie, dann werden die anderen Standard-Klick-Töne gespielt. Versuchen Sie Änderungen der Lautstärke-Einstellungen etc. vorzunehmen.

Spielen von Musik

Um einen Ton zu erzeugen, können Sie folgenden Befehl verwenden
`PlayToneEx(frequency, duration, volume, loop?)`

Dieser Befehl hat vier Parameter. Der erste ist die Frequenz in Hertz, der zweite bestimmt die Dauer des Tons (in 1/1000 einer Sekunde, wie die Wartezeit) letzen beiden Parameter kennen Sie ja bereits.
`PlayToneEx(frequency, duration)` kann auch verwendet werden, falls die Lautstärke über das NXT-Menü bestimmt werden soll und falls keine Wiederholung erfolgt.

Hier sehen Sie eine Tabelle nützlicher Frequenzen:

Sound	3	4	5	6	7	8	9
H	247	494	988	1976	3951	7902	
A#	233	466	932	1865	3729	7458	
A	220	440	880	1760	3520	7040	14080
G#		415	831	1661	3322	6644	13288
G		392	784	1568	3136	6272	12544
F#		370	740	1480	2960	5920	11840
F		349	698	1397	2794	5588	11176
E		330	659	1319	2637	5274	10548
D#		311	622	1245	2489	4978	9956
D		294	587	1175	2349	4699	9398
C#		277	554	1109	2217	4435	8870
C		262	523	1047	2093	4186	8372

Wie im Falle von `PlayFileEx`, wartet der NXT nicht darauf, bis ein Ton „fertig“ ist. Sollten Sie also mehrere Töne in eine Zeile nutzen, dann fügen Sie am besten den wait-Befehl ein. Hier ein Programmbeispiel:

```
#define VOL 3

task main()
{
  PlayToneEx(262,400,VOL,FALSE); Wait(500);
  PlayToneEx(294,400,VOL,FALSE); Wait(500);
  PlayToneEx(330,400,VOL,FALSE); Wait(500);
  PlayToneEx(294,400,VOL,FALSE); Wait(500);
  PlayToneEx(262,1600,VOL,FALSE); Wait(2000);
}
```

Sie können „Musikstücke“ leicht selbst komponieren, indem Sie das *Brick Piano*, welches ebenfalls in BricxCC enthalten ist nutzen. Wenn Sie möchten, dass Ihr Roboter während er umher fährt Musik spielt, empfiehlt es sich separate Tasks zu nutzen. Hier ein Programmbeispiel eines Roboters, der vor und zurück fährt und währenddessen Musik macht:

```
task music()
{
  while (true)
  {
    PlayTone(262,400); Wait(500);
    PlayTone(294,400); Wait(500);
    PlayTone(330,400); Wait(500);
    PlayTone(294,400); Wait(500);
  }
}

task movement()
{
  while(true)
  {
     OnFwd(OUT_AC, 75); Wait(3000);
     OnRev(OUT_AC, 75); Wait(3000);
  }
}

task main()
{
  Precedes(music, movement);
}
```

Zusammenfassung

In diesem Kapitel haben Sie gelernt, wie man mit dem NXT Musik macht bzw. Musikdateien abspielt. Ebenfalls haben Sie gesehen, wie separate Tasks hierfür genutzt werden können.

8 Mehr über Motoren

Es gibt eine Reihe von zusätzlichen Motor-Befehlen, die Sie verwenden können, um die Motoren noch genauer zu steuern. In diesem Kapitel werden wir folgende Befehle näher betrachten: ResetTachoCount, Coast (Float), OnFwdReg, OnRevReg, OnFwdSync, OnRevSync, RotateMotor, RotateMotorEx, und die grundlegenden Konzepte eines PID-Reglers.

Sachtes bremsen

Wenn Sie den `Off` ()-Befehl verwenden, stoppt der Servomotor sofort, der Motor bremst und hält die Position. Es ist aber auch möglich, um die Motoren sanft abzubremsen, quasi auslaufen zu lassen. Hierfür können Sie die Befehle `Float()` oder `Coast()` verwenden, beide Befehle lassen den Motor auslaufen. Hier ist ein einfaches Beispiel, dass die Verwendung des Float-Befehls veranschaulicht. Zuerst stoppt (bremst) der Roboter, nach dem anschließenden weiterfahren, lässt er die Motoren auslaufen. Beachten Sie den Unterschied, dieser ist je nach Art und Bauweise des Roboters beträchtlich.

```
task main()
{
  OnFwd(OUT_AC, 75);
  Wait(500);
  Off(OUT_AC);
  Wait(1000);
  OnFwd(OUT_AC, 75);
  Wait(500);
  Float(OUT_AC);
}
```

Fortgeschrittene Befehle

Die Befehle OnFwd() und OnRev() sind die einfachsten Routinen um die Motoren zu steuern.

Die NXT-Servomotoren haben einen eingebauten Encoder, der es erlaubt, die Position der Achse und die Geschwindigkeit des Motors sehr genau anzusteuern.

Wenn Sie möchten, dass Ihr Roboter perfekt gerade aus fährt, können Sie die Motoren synchronisieren. NXC besitzt eine Funktion, die es ermöglicht, dass zwei vorher ausgewählte Motoren „gemeinsam“ laufen und aufeinander warten, wenn einer etwas langsamer oder gar blockiert ist. In ähnlicher Weise können Sie zwei Motoren synchronisieren um eine prozentgenaue Lenkung nach links oder rechts zu vollziehen. Es gibt viele Befehle um die Möglichkeiten der NXT-Servomotoren voll zu Nutzen.

Der Befehl: `OnFwdReg('ports','speed','regmode')` steuert die Motoren über die Parameter „ports“ „speed“ und den (Regulierungsmodus) „regmode“.

Es gibt drei Regulierungsarten:
`OUT_REGMODE_IDLE`, → keine PID-Reglung
`OUT_REGMODE_SPEED` → die/der Motor wird über die Geschwindigkeit reguliert
`OUT_REGMODE_SYNC`. → zwei Motoren werden mit einander synchronisiert

Der Befehl: `OnRevReg()` fungiert auf dieselbe Art wie `OnFwdReg`, nur für die umgekehrte Richtung.

```
task main()
{
  OnFwdReg(OUT_AC,50,OUT_REGMODE_IDLE);
  Wait(2000);
  Off(OUT_AC);
  PlayTone(4000,50);
  Wait(1000);
  ResetTachoCount(OUT_AC);
  OnFwdReg(OUT_AC,50,OUT_REGMODE_SPEED);
  Wait(2000);
  Off(OUT_AC);
  PlayTone(4000,50);
  Wait(1000);
  OnFwdReg(OUT_AC,50,OUT_REGMODE_SYNC);
  Wait(2000);
  Off(OUT_AC);
}
```

Dieses Programm veranschaulicht ziemlich gut wie sich die verschiedenen Ansteuerungsarten auswirken. Zur Demonstration, nehmen Sie Ihren Roboter am besten in die Hand:

1. (IDLE mode), stoppen Sie ein Rad – Sie werden nichts merken.
2. (SPEED mode), versuchen Sie ein Rad mit der Hand zu verlangsamen – sie werden merken, wie der NXT versucht, diese Verlangsamung durch mehr Erhöhung der Geschwindigkeit auszugleichen. Dieser Modus versucht die Geschwindigkeit konstant zu halten.
3. (SYNC mode), wenn Sie ein Rad verlangsamen, werden Sie bemerken, wie sich das andere Rad anpasst und auch seine Geschwindigkeit verringert.

Der Befehl: `OnFwdSync('ports','speed','turnpct')` ist der gleiche Befehl wie `OnFwdReg`() nur eben im Synchronisationsmodus, dies bedeutet, dass der „Steuerungs-„ Parameter „turnpct“ Prozentual (von -100 bis +100) angegeben werden kann.

Der Befehl: `OnRevSync()`ist der gleiche wie `OnFwdSync` nur in umgekehrter Richtung.

Das folgende Programmbeispiel veranschaulicht diese drei Befehle, verändern Sie die Parameter und sehen Sie welche Auswirkungen dies mit sich bringt.

```
task main()
{
  PlayTone(5000,30);
  OnFwdSync(OUT_AC,50,0);
  Wait(1000);
  PlayTone(5000,30);
  OnFwdSync(OUT_AC,50,20);
  Wait(1000);
  PlayTone(5000,30);
  OnFwdSync(OUT_AC,50,-40);
  Wait(1000);
  PlayTone(5000,30);
  OnRevSync(OUT_AC,50,90);
  Wait(1000);
  Off(OUT_AC);
}
```

Schließlich, können die Motoren so eingestellt werden, dass sich nur einer begrenzten Gradzahl drehen (eine volle Radumdrehung entspricht 360°).

Die beiden folgenden Befehle ändern die Motorrichtung durch die Änderung des Vorzeichens der Geschwindigkeit bzw. durch die Änderung des Vorzeichens des Winkels: dies bedeutet, wenn das Vorzeichen der Geschwindigkeit und des Winkels gleich sind, dreht sich der Motor vorwärts, unterscheiden sich die beiden Vorzeichen, dreht sich der Motor in die entgegen gesetzte Richtung.

Der Befehl: `RotateMotor('ports','speed','degrees')` benötigt die Parameter „ports“, (Anschluss des Motors) „speed“ (Geschwindigkeit auf einer Skala von 0 bis +100) „degrees“.

```
task main()
{
  RotateMotor(OUT_AC, 50,360);
  RotateMotor(OUT_C, 50,-360);
}
```

`RotateMotorEx('ports','speed','degrees','turnpct','sync','stop')` ist eine Erweiterung des vorherigen Befehls, mit dem Sie zwei Motoren synchronisieren können (z. B. OUT_AC). Zusätzlich zur Angabe des „turnpct“ kann nun ein boolean[14] Flag (true oder false) für die Synchronisierung gesetzt werden und Sie können bestimmen (ebenfalls mit einem boolean Flag), ob der Motor nach der in „degree“ angegebenen Rotation abgebremst (gestoppt) werden soll.

```
task main()
{
RotateMotorEx(OUT_AC, 50, 360, 0, true, true);
RotateMotorEx(OUT_AC, 50, 360, 40, true, true);
RotateMotorEx(OUT_AC, 50, 360, -40, true, true);
RotateMotorEx(OUT_AC, 50, 360, 100, true, true);
}
```

PID-Regler

Die NXT Firmware implementiert einen digitalen PID (proportional–integral–derivative controller) Regler um präzise die Position und die Geschwindigkeit der Servomotoren zu steuern. Dieser Regler ist einer der einfachsten, zugleich aber auch wirksamsten Regler.

Einfach ausgedrückt funktioniert der PID-Regler folgendermaßen:

Ihr Programm gibt dem Regler einen festen Punkt (R(t) vor. Der Servomotor wird nun (mithilfe des eingebauten Encoders) mit dem „Befehl“ U(t) veranlasst, seine Position Y(t) zu messen. Der Encoder kalkuliert ebenfalls den Fehler E(t) = R(t) – Y(t). Aufgrund dieser Vorgehensweise wird es Regelung (im Gegensatz zur Steuerung) genannt, da die ermittelte Position (der Ausgangswert) Y(t) als Eingangswert für die Berechnung des Fehlers dient. Der Regler wandelt den Fehler E(t) folgendermaßen zum Motorbefehl U(t) um:

$U(t) = P(t) + I(t) + D(t)$, wobei
$P(t) = K_p * E(t)$,
$I(t) = K_I * (I(t-1) + E(t))$
And $D(t) = K_D * (E(t) - E(t-1))$.

Für Anfänger kann dies etwas kompliziert aussehen, versuchen wir es aufzuschlüsseln:

Der Befehl ist die Summe der drei Anteile, der proportionale Anteil P(t), der integrale Anteil I(t) und der abgeleitete Anteil D(t).

- P(t) regelt die Veränderung (der Position) über die Zeit, gewährleistet aber nicht den sog. Null-Fehler auszugleichen
- I(t) verleiht dem Regler eine Art Gedächtnis, die Fehler werden über die Zeit (t) aufsummiert und kompensiert. Der Null-Fehler wird somit ausgeglichen.
- D(t) verleiht dem Regler ein Art Vorhersage-Möglichkeit (ähnlich der Ableitung aus der Mathematik) je größer der Fehler, desto mehr steuert der Regler dieser Entwicklung entgegen.

Dies mag immer noch verwirrend klingen, angesichts der Tatsache wie viel Akademische Abhandlungen bereits über dieses Thema geschrieben wurden. Aber wir können immer noch versuchen sie „online“, mit unserem NXT auszuprobieren! Am besten wir sehen uns einfach folgendes Programmbeispiel an:

[14] Laut wikipedia: Eine Boolesche Variable kann immer einen von zwei Werten annehmen. Dieses Wertepaar wird je nach Anwendung u. a. als „wahr/falsch“, „true/false“ oder „1/0“ bezeichnet. [Juli, 2009]

```
#define P 50
#define I 50
#define D 50

task main()
{
  RotateMotorPID(OUT_A, 100, 180, P, I, D);
  Wait(3000);
}
```

Der Befehl: RotateMotorPID(port,speed, angle, Pgain,Igain,Dgain) ermöglicht es Ihnen, den Motor mit verschiedenen, von den Standardwerten abweichenden, PID-Einstellungen zu regeln. Versuchen Sie, die folgenden Werte:

(50, 0, 0)	der Motor dreht sich nicht exakt um 180 °, solange ein unkompensierter Fehler bleibt
(0, x, x)	ohne den Proportional-Anteil bleibt der Fehler sehr hoch
(40,40,0)	dies ist eine Überschreitung, dies bedeutet, dass die Motorachse über den gesetzten Punkt hinaus dreht und sich anschließend zurück bewegt
(40,40,90)	gute Präzision und Erhöhung der Zeiteinheiten (Zeit bis zum Erreichen der festgelegten Punkt)
(40,40,200)	die Achse oszilliert, solange wie der abgeleitete Anteil zu hoch bleibt.

Setzen Sie andere Werte ein, um herauszufinden wie die verschiedenen Anteile die Steuerung des Motors beeinflussen.

Zusammenfassung

In diesem Kapitel haben Sie gelernt welche Weiterführenden Motorbefehle es sonst noch gibt. Wie `Float()`, `Coast()` die den Motor sanft abbremsen oder die Befehle: `OnXxxReg()`und `OnXxxSync()` die die Motoren mittels der eigenen Encoderwerte Geschwindigkeit und Synchronisation steuern. `RotateMotor()`und `RotateMotorEx()` Befehle die es erlauben den Motor gradgenau zu steuern. Ebenso haben sie etwas über den PID-Regler erfahren, es war zwar keine erschöpfende Erklärung, aber vielleicht hat es Sie neugierig auf mehr gemacht. Durchsuchen Sie das Internet!

9 Mehr über Sensoren

In Kapitel 5 haben wir die grundlegenden Aspekte der Verwendung von Sensoren kennen gelernt. Aber es gibt noch viel mehr Möglichkeiten, Sensoren einzusetzen. In diesem Kapitel diskutieren wir den Unterschied zwischen Sensor-Modus und Sensor-Typ, wir werden sehen, wie die alten, mit dem RCX kompatiblen, Sensoren verwendet werden und wie diese mit einem (im LEGO MINDSTORMS Set mitgelieferten) Konverter-Kabel am NXT angeschlossen werden.

Sensor-Modus und Sensor-Typ

Der SetSensor ()-Befehl den wir aus Kapitel 5 kennen erledigt zwei Dinge: Er bestimmt den Typ des Sensors und er setzt den Modus in dem der Sensor arbeiten soll. Durch die separate Festelegung des Modus und des Typs kann das Verhalten des Sensors präziser gesteuert werden, was für bestimmte Anwendungen sehr nützlich sein kann.

Der Sensor-Typ wird durch den Befehl `SetSensorType()` bestimmt. Es gibt viele verschiedene Sensorarten hier die wichtigsten:

`SENSOR_TYPE_TOUCH`	→ Tastsensor
`SENSOR_TYPE_LIGHT_ACTIVE`	→ (aktiver) Lichtsensor, mit eingeschalteten LED
`SENSOR_TYPE_SOUND_DB`	→ Geräuschsensor
`SENSOR_TYPE_LOWSPEED_9V`	→ Ultraschallsensor

Das Einstellen des Sensor-Typs ist im speziellen wichtig, um anzugeben welche Leistung der Sensor benötigt (z.B. um die LED des Lichtsensors zu betreiben), oder um dem NXT-Stein mitzuteilen, dass es sich um einen digitalen Sensor handelt, der über das Serielle Protokoll I^2C gelesen werden muss.

Es ist ebenso möglich, alte RCX Sensoren mit dem NXT zu nutzen (bzw. unter NXC zu programmieren):

`SENSOR_TYPE_TEMPERATURE`	→ RCX-Temperatursensor
`SENSOR_TYPE_LIGHT`	→ RCX-Lichtsensor
`SENSOR_TYPE_ROTATION`	→ RCX-Rotationssensor

Der Modus des Sensors wird mit dem Befehl `SetSensorMode()` bestimmt. Es gibt acht verschiedene Modi. Der wichtigste Modus ist `SENSOR_MODE_RAW`.
Befindet sich der Sensor in diesem Modus befindet sich der Wert, den der Sensor liefert zwischen 0 und 1023. Es ist der sogenannte Rohwert des Sensors. Die Interpretation dieses Wertes ist abhängig vom jeweiligen Sensor. Zum Beispiel der Tastsensor: Ist der Tastsensor nicht gedrückt, liefert er einen Wert, in der Nähe von 1023. Ist der Sensor stark gedrückt liefert er einen Wert um die 50. Wird der Sensor nur leicht gedrückt, gibt er einen Wert zwischen 50 und 1000 zurück. Wenn Sie also den Berührungssensor auf den Raw-Mode setzen, können Sie herauszufinden, ob der Tastsensor nur leicht gedrückt wurde[15]. Wird der Lichtsensor im Raw-Mode betrieben, liefert er Werte zwischen 300 (sehr hell) bis 800 (sehr dunkel). Dadurch können wesentlich genauere Werte als mit dem SetSensor ()-Befehl ermittelt werden. Weitere Informationen finden Sie im NXC Programming Guide.

Der zweite Sensor-Modus ist `SENSOR_MODE_BOOL`. Dieser Modus liefert die Rückgabewerte 0 oder 1. Sollte der gemessene Rohwert über 562 sein, wird eine 0 zurückgegeben, ansonsten eine 1. Der Modus SENSOR_MODE_BOOL ist der Standardmodus für den Tastsensor. Er kann aber auch für andere (analoge) Sensoren verwendet werden. Die Modi `SENSOR_MODE_CELSIUS` und `SENSOR_MODE_FAHRENHEIT` sind nur für Temperatursensoren sinnvoll und liefern Werte als Celsius oder Fahrenheit zurück. `SENSOR_MODE_PERCENT` konvertiert den Rohwert in einen Prozentwert zwischen 0 und 100. SENSOR_MODE_PERCENT ist der Standardwert für den Lichtsensor. `SENSOR_MODE_ROTATION` ist nur sinnvoll für den Rotationssensor – siehe unten.

Es gibt zwei weitere interessante Modi: `SENSOR_MODE_EDGE` und `SENSOR_MODE_PULSE` Diese zählen Übergänge von einem niedrigen Rohwert zu einem höheren, bzw. in umgekehrter Richtung. Zum Beispiel wenn ein Tastsensor gedrückt wird, hat dies einen Übergang - von einem hohen Rohwert zu einem niedrigeren Rohwert zur Folge. Wird der Tastsensor wieder losgelassen, erfolgt ein Übergang in die andere Richtung. Wird der Sensor in den Modus SENSOR_MODE_PULSE gesetzt, werden nur Übergänge von niedrig zu hoch gezählt. Damit würde für das drücken und loslassen des Tastsensors als ein Übergang gezählt. Wir der Sensor dagegen im SENSOR_MODE_EDGE gesetzt, werden beide Übergänge gezählt. Dieser Modus kann z.B. benutzt werden, um zu zählen wie oft der Tastsensor gedrückt wurde, oder der Modus kann zusammen mit dem

[15] Befindet sich der Sensor nicht im Raw-Mode liefert er 0 für nicht gedrückt und 1 für gedrückt, weitere Unterscheidungsmöglichkeiten gibt es nicht.

Lichtsensor benutz werden, um zu zählen wie oft eine Lampe Ein- und Ausgeschaltet wurde. Natürlich sollte man beim zählen der Übergänge auch in der Lage sein, den „Zähler“ wieder auf 0 zu setzen. Hierfür kann der Befehl `ClearSensor()`genutzt werden.

Das folgende Programmbeispiel steuert einen Roboter. Der Tastsensor wurde mit einem langen Kabel auf Eingang 1 gelegt. Wenn der Tastsensor schnell zweimal hintereinander gedrückt wurde, fährt der Roboter vorwärts. Wird der Tastsensor einmal gedrückt, hält der Roboter an.

```
task main()
{
  SetSensorType(IN_1, SENSOR_TYPE_TOUCH);
  SetSensorMode(IN_1, SENSOR_MODE_PULSE);
  while(true)
  {
    ClearSensor(IN_1);
    until (SENSOR_1 > 0);
    Wait(500);
    if (SENSOR_1 == 1) {Off(OUT_AC);}
    if (SENSOR_1 == 2) {OnFwd(OUT_AC, 75);}
  }
}
```

Bitte beachten Sie, dass wir zuerst den Typ des Sensors und anschließend den Modus definiert haben. Dies ist wichtig, da der Typ des Sensors auch den jeweiligen Modus beeinflussen kann.

Der Rotationssensor

Der alte RCX-Rotationssenor wird hier nicht beschrieben.

Zusammenfassung

In diesem Kapitel haben Sie zusätzliche Eigenschaften der Sensoren kennen gelernt. Sie haben gesehen, wie unabhängig voneinander der Sensortyp und Sensormode bestimmt werden können und wie diese die vom Sensor zurückgegebenen Daten beeinflussen.

10 Parallele Tasks

Wie bereits erwähnt wurde, können Tasks in NXC gleichzeitig oder parallel ,wie üblicherweise auch dazu gesagt werden kann, abgearbeitet werden. Diese Art der Verarbeitung kann z.B. sehr nützlich sein, um mit einer Task die Sensoren abzufragen, während eine andere Task den Roboter steuert und eine dritte Task gleichzeitig dazu Musik erzeugt. Aber auch parallele Task können zu Problemen führen. Eine Aufgabe kann eine andere stören/beeinflussen.

Ein falsches Programm

Betrachten Sie das folgende Programm. Eine Task veranlasst den Roboter ein Quadrat zu fahren (wie wir es schon oft programmiert haben) und die zweite Task kontrolliert den Tastsensor. Wurde der Sensor berührt, soll er sich ein wenig zurück bewegen und eine 90-Grad-Wende machen.

```
/*Falsches Programm*/

task check_sensors()
{
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      OnRev(OUT_AC, 75);
      Wait(500);
      OnFwd(OUT_A, 75);
      Wait(850);
      OnFwd(OUT_C, 75);
    }
  }
}

task submain()
{
  while (true)
  {
    OnFwd(OUT_AC, 75);
    Wait(1000);
    OnRev(OUT_C, 75);
    Wait(500);
  }
}

task main()
{
  SetSensor(IN_1,SENSOR_TOUCH);
  Precedes(check_sensors, submain);
}
```

Dies mag auf den ersten Blick vielleicht wie ein gut funktionierendes Programm aussehen. Wenn Sie es aber ausführen, werden Sie wahrscheinlich einige unerwartete Verhaltensweisen entdecken. Versuchen Sie folgendes: Berühren Sie den Tastsensor, während der Roboter gerade die 90-Grad-Wende ausführt. Er wird etwas zurück fahren, aber dann unverzüglich wieder Vorwärts fahren. Der Grund hierfür ist, dass sich die Tasks gegenseitig beeinflussen.
Folgendes geschieht: Der Roboter dreht sich nach rechts, der erste Task befindet sich im zweiten wait-Befehl. Nun wird der Tastsensor berührt. Der Roboter beginnt Rückwärts zu fahren, in diesem Moment ist der submain-Task fertig mit dem wait-Befehl und veranlasst den Roboter (bzw. die Motoren) sich wieder Vorwärts zu bewegen. Der zweite Task (`check_sensors`) befindet sich derzeit im wait-Befehl und kann eine weitere Berührung nicht feststellen. Dies ist natürlich nicht das Verhalten, welches wir erwarten. Das Problem ist, dass während der zweite Task „schläft“ (`Wait`), das Programm nicht realisiert, dass der erste Task immer noch läuft und das dieses Verhalten, das Verhalten der zweiten Task beeinflusst.

Kritische Abschnitte und Mutex Variablen

Eine Möglichkeit zur Lösung dieses Problems ist, sicherzustellen, dass zu jedem Zeitpunkt nur eine Task die Motoren des Roboters ansteuert. Dieses Konzept haben wir in Kapitel 6 kennen gelernt. Lassen Sie es uns nochmals wiederholen.

```
mutex moveMutex;          //Deklaration der mutex-Variable

task move_square()
{
  while (true){
    Acquire(moveMutex);
    OnFwd(OUT_AC, 75); Wait(1000);
    OnRev(OUT_C, 75); Wait(850);
    Release(moveMutex);
  }
}

task check_sensors()
{
  while (true){
    if (SENSOR_1 == 1)
    {
      Acquire(moveMutex);
      OnRev(OUT_AC, 75); Wait(500);
      OnFwd(OUT_A, 75); Wait(850);
      Release(moveMutex);
    }
  }
}

task main()
{
  SetSensor(IN_1,SENSOR_TOUCH);
  Precedes(check_sensors, move_square);
}
```

Der springende Punkt bei diesem Programm ist, dass sowohl die Task `check_sensors` als auch die Task `move_square` nur dann auf die Motoren zugreifen können, wenn die jeweils andere Task die Motoren nicht anspricht. Dies geschieht durch die **Acquire**-Anweisung die darauf wartet, dass die **mutex**-Variable `moveMutex` die Motoren wieder freigibt. Der Gegenpart der **Acquire**-Anweisung ist die **Release**-Anweisung. Diese gibt die mutex-Variable wieder „frei“, so dass andere Task wieder auf die Motoren zugreifen können. Der Code innerhalb der Acquire-Release-Anweisung wird kritische Region bzw. kritischer Programmblock genannt. Kritisch bedeutet in diesem Zusammenhang, dass auf gemeinsamen Ressourcen zugegriffen wird. Wird die mutex, Acquire und Release auf diese Weise verwendet, können sich Task gegenseitig nicht stören.

Das Nutzen von Semaphoren[16]

Es gibt eine „handgemachte“ Alternative zur Nutzung von **mutex**-Variablen, dies ist die explizite Implementierung der Befehle **Acquire** und **Release**.
Eine Standard-Technik, hierfür ist; die Verwendung einer „Signal-„ Variable, deren Aufgabe es ist, anzuzeigen welche Task die Motoren (gerade) kontrolliert. Einer Task ist es solange nicht gestattet, auf die Motoren zuzugreifen, bis die „Signal-„ Variable anzeigt, dass die Motoren (frei) wieder bereit sind. Eine solche Variable wird oft als Semaphor bezeichnet. Bezeichnen wir nun `sem`[17] als ein Semaphor (wie mutex). Wir definieren nun die Variable `sem` wie folgt: Hat `sem` den Wert 0, bedeutet dies, dass aktuell keine Task auf die Motoren zugreift. Beansprucht ein Task nun die „Ressource“ Motor für sich, nimmt die Variable `sem` den Wert 1 ein.

Folgendes Programmbeispiel veranschaulicht, wie dies umgesetzt werden kann:

[16] Semaphore bedeutet wörtlich übersetzt: Signalmast, Flaggensignal – es ist aber durchaus üblich diesen begriff ohne Übersetzung zu verwenden.
[17] Die Variable sem wird im folgenden Beispiel als sogenannte „globale-Variable“ deklariert. Auf globale-Variablen kann von überall aus (Task, Subroutinen, etc.) zugegriffen werden.

```
until (sem == 0);
sem = 1;     //Acquire(sem);

// Do something with the motors
// critical region

sem = 0;     //Release(sem);
```

Die erste Programmzeile `until (sem == 0)` veranlasst einen Task (definieren wir diesen als *task_1*) solange zu warten, bis die Motoren freigegeben wurden. Ist dies der Fall, kann *task_1* nun seinerseits auf die Motoren zugreifen. Dabei wird der Semaphore `sem` der Wert 1 zugewiesen. Anschließend wird der kritische Programmblock ausgeführt. Danach wird der `sem` der Wert 0 zugewiesen und auf die Motoren kann nun wieder zugegriffen werden. Nun lässt sich das zuvor „fehlerhaft“ programmierte Programm, mit Hilfe der Semaphore korrigieren, damit das unvorhersehbare Verhalten des Roboters nicht länger auftaucht.

```
int sem;            //Deklaration der globalen Variable sem

task move_square()
{
  while (true)
  {
    until (sem == 0); sem = 1;
    OnFwd(OUT_AC, 75);
    sem = 0;
    Wait(1000);
    until (sem == 0); sem = 1;
    OnRev(OUT_C, 75);
    sem = 0;
    Wait(850);
  }
}

task submain()
{
  SetSensor(IN_1, SENSOR_TOUCH);
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      until (sem == 0); sem = 1;
      OnRev(OUT_AC, 75); Wait(500);
      OnFwd(OUT_A, 75); Wait(850);
      sem = 0;
    }
  }
}

task main()
{
  sem = 0;
  Precedes(move_square, submain);
}
```

Es kann hier argumentiert werden, dass es nicht notwendig ist, in der Task `move_square` die Semaphore auf 1 bzw. 0 zu setzen. Dem kann allerdings widersprochen werden, da der Befehl `OnFwd` faktisch aus zwei Befehlen besteht (siehe auch Kapitel 8). Es ist nicht gewollt, dass diese Befehlssequenz von einer anderen Task unterbrochen wird. Semaphoren sind sehr nützlich und manchmal auch notwendig, wenn komplexere Programme mit parallelen Task geschrieben werden[18].

[18] Deadlock-Problem: Bei der Verwendung paralleler Programmstrukturen kann das sogenannte Deadlock-Problemm auftreten. Hierbei warten zwei Abläufe (Tasks) gegenseitig aufeinander (hier: auf die Freigabe der Resource Motor). Zur Behebung dieser Problematik können verschiedene Techniken angewendet werden. Die einfachste Lösung wäre, jede Task darf nur eine Resource gleichzeitig verwenden.

Zusammenfassung

In diesem Kapitel haben wir uns mit den Problemen befasst, die auftauchen können wenn verschiedenen bzw. mehrere Task benutzt werden. Derartige Seiteneffekte sollten immer, bei unvorhergesehenen Verhaltensweisen des Roboters, Berücksichtigung bei der Fehleranalyse finden. Zwei verschiedene Lösungsansätze wurden aufgezeigt. Der erste stoppt und startet die Task, um sicher zu stellen, dass immer nur eine Task (zu einer Zeit) aktiv ist. Der zweite Ansatz nutzt Semaphoren um die zugriffe einer Task auf kritische Programmblöcke (Ressourcen) zu kontrollieren.

11 Kommunikation zwischen Robotern

Sollten Sie mehr als einen NXT besitzen wird dieses Kapitel sehr interessant für Sie sein. Aber auch ohne einen zweiten NXT können Sie immer noch Daten zwischen Ihrem NXT und dem PC austauschen. Die NXTs (Roboter) können via Bluetooth Daten austauschen. Beispielweise kann mit Hilfe der Bluetooth-Kommunikation mehrere Roboter zusammenarbeiten bzw. könnten Sie einen Roboter konstruieren, der zwei NXTs nutzt.

Das Vorgängermodell, der RCX nutzte Infrarot-Strahlen zur Kommunikation, diese konnten von allen weiteren in der Umgebung befindlichen RCX empfangen werden. Dies hatte den Vorteil, dass die Zahl der untereinander kommunizierenden Roboter (theoretisch) nicht begrenzt war. Allerdings war die Infrarot-Kommunikation des RCX sehr anfällig auf externe Lichtquellen und bei der Kommunikation über großen Reichweiten. Der NXT nutzt Radiowellen (Bluetooth) zur Kommunikation. Diese Art der Kommunikation ist sehr unterschiedlich zu der beim RCX genutzten. Zuerst müssen zwei oder mehr NXTs (bzw. NXT mit PC) miteinander (drahtlos) verbunden werden. Dies erfolgt über das NXT-Menü. Nur wenn die NXTs über das NXT-Menü verbunden wurden lassen sich Nachrichten/Daten zwischen Ihnen austauschen.
Der NXT, der die Verbindung zu (einem) anderen NXT(s) aufbaut wird Master genannt. Der NXT, der diese Verbindungsaufforderung annimmt wird Slave genannt. Ein Master kann mit bis zu 3 Slaves gleichzeitig Verbunden sein. Hierfür stehen die Kanäle 1,2 und 3 zur Verfügung. Der Slave wird immer auf Kanal 0 mit dem Master verbunden. Zusätzlich können die versendeten Nachrichten an 10 verschiedene Mailboxen (Briefkästen) adressiert werden.

Master – Slave Kommunikation

Im Folgenden werden zwei Programmbeispiele gezeigt. Das erste Beispiel ist das Programm des Masters (dieses muss auch auf den Master-NXT übertragen werden). Diese grundlegenden Programmbeispiele sollen Ihnen zeigen, wie eine kontinuierliche, drahtlose Nachrichtenverbindung zwischen zwei NXTs aufgebaut wird. Als Datenformat für den Austausch der Nachrichten wurde „string“[19] gewählt.
Das Programm des Masters kontrolliert zuerst, ob der Slave korrekt auf Kanal 1 (`BT_CONN (constant)`) mit dem Master verbunden ist. Dafür wird folgende Funktion verwendet: `BluetoothStatus(conn)` Anschließend wird die Nachricht „erstellt“ und versendet mit: `SendRemoteString(conn,queue,string)`. Eingehende Nachrichten vom Slave werden mit: `ReceiveRemoteString(queue,clear,string)` empfangen und auf dem Display ausgegeben.

```
//MASTER
#define BT_CONN 1
#define INBOX 1
#define OUTBOX 5

sub BTCheck(int conn){
  if (!BluetoothStatus(conn)==NO_ERR){
    TextOut(5,LCD_LINE2,"Error");
    Wait(1000);
    Stop(true);
  }
}

task main(){
  string in, out, iStr;
  int i = 0;
  BTCheck(BT_CONN); //check slave connection
  while(true){
    iStr = NumToStr(i);
    out = StrCat("M",iStr);
    TextOut(10,LCD_LINE1,"Master Test");
    TextOut(0,LCD_LINE2,"IN:");
    TextOut(0,LCD_LINE4,"OUT:");
    ReceiveRemoteString(INBOX, true, in);
    SendRemoteString(BT_CONN,OUTBOX,out);
    TextOut(10,LCD_LINE3,in);
    TextOut(10,LCD_LINE5,out);
    Wait(100);
    i++;
  }
}
```

[19] Als „string“ wird eine Sammlung von Zeichen (Datentyp char) bezeichnet.

Das Programm des Slaves ist ähnlich dem des Masters, mit der Ausnahme, dass es die `SendResponseString(queue,string)`nutzt, anstatt des `SendRemoteString` Befehls, da der Slave nur an den Master auf Kanal 0 – Nachrichten schicken kann.

```
//SLAVE
#define BT_CONN 1
#define INBOX 5
#define OUTBOX 1

sub BTCheck(int conn){
  if (!BluetoothStatus(conn)==NO_ERR){
    TextOut(5,LCD_LINE2,"Error");
    Wait(1000);
    Stop(true);
  }
}

task main(){
  string in, out, iStr;
  int i = 0;
  BTCheck(0); //check master connection
  while(true){
    iStr = NumToStr(i);
    out = StrCat("S",iStr);
    TextOut(10,LCD_LINE1,"Slave Test");
    TextOut(0,LCD_LINE2,"IN:");
    TextOut(0,LCD_LINE4,"OUT:");
    ReceiveRemoteString(INBOX, true, in);
    SendResponseString(OUTBOX,out);
    TextOut(10,LCD_LINE3,in);
    TextOut(10,LCD_LINE5,out);
    Wait(100);
    i++;
  }
}
```

Sollte eines der Programme beendet werden, gehen die Nachrichten des anderen verloren. Die Beendigung eines Programms (egal ob Master oder Slave) wird dem anderen Programm/Roboter nicht mitgeteilt. Um dieses Problem zu vermeiden, können wir ein „eleganteres“ Protokoll schreiben, dass eine Empfangsbestätigung beinhaltet.

Versenden von Nummern mit Empfangsbestätigung

Als nächstes betrachten wir zwei Programme; der Master sendet Nummern mit: `SendRemoteNumber(conn,queue,number)`und wartet auf die Bestätigung des Slaves, der dies innerhalb einer until-Anweisung mit `ReceiveRemoteString`) macht. Nur wenn der Slave die eingehenden Nachrichten bestätigt, sendet der Master die nächste Nachricht. Der Slave empfängt die Nachrichten mit `ReceiveRemoteNumber(queue,clear,number)` und sendet die Bestätigungen mit `SendResponseNumber`. Das hier beschriebene Master-slave-Programm muss eine gemeinsame Codierung der Bestätigung vereinbaren. Hierfür wurde der Hexadezimalwert 0xFF gewählt.

Der Master schickt Zufallszahlen und wartet auf die Bestätigung, jedes Mal wenn eine Bestätigung mit dem korrekten Hex-Wert eingegangen ist, muss die Bestätigungs-Variable zurückgesetzt werden. Ansonsten würde der Master kontinuierlich Nachrichten senden, da die Bestätigungs-Variable fälschlicherweise eine (bzw. die letzte) Empfangsbestätigung anzeigt. Der Eingang einer korrekten Nachricht würde somit genügen um den Master zu veranlassen, ständig weitere Nachrichten/Nummern zu senden.

Der Slave seinerseits kontrolliert ständig seine Mailbox und gibt, vorausgesetzt die Mailbox ist nicht leer, die erhaltenen Zahlen auf seinem Display aus. Ebenso sendet der Slave den Bestätigungs-Hex-Code. Beim Programmstart sendet der Slave als erstes eine Bestätigung, um den Bestätigungsmechanismus des Masters zu aktivieren. Würde ohne diesen Trick der Master vor dem Slave gestartet werden, immer auf die Bestätigung seiner Nachricht warten. Die hier beschriebene Programmierung geht das Risiko ein, dass die erste Nachricht verloren geht, der Master aber nicht in einer „warte-auf-Bestätigungsschleife“ hängen bleibt.

```
//MASTER
#define BT_CONN 1
#define OUTBOX 5
#define INBOX 1
#define CLEARLINE(L) \
TextOut(0,L," ");

sub BTCheck(int conn){
  if (!BluetoothStatus(conn)==NO_ERR){
   TextOut(5,LCD_LINE2,"Error");
   Wait(1000);
   Stop(true);
  }
}

task main(){
  int ack;
  int i;
  BTCheck(BT_CONN);
  TextOut(10,LCD_LINE1,"Master sending");
  while(true){
    i = Random(512);
    CLEARLINE(LCD_LINE3);
    NumOut(5,LCD_LINE3,i);
    ack = 0;
    SendRemoteNumber(BT_CONN,OUTBOX,i);
    until(ack==0xFF) {
    until(ReceiveRemoteNumber(INBOX,true,ack) == NO_ERR);
  }
  Wait(250);
}
```

```
//SLAVE
#define BT_CONN 1
#define OUT_MBOX 1
#define IN_MBOX 5

sub BTCheck(int conn){
  if (!BluetoothStatus(conn)==NO_ERR){
    TextOut(5,LCD_LINE2,"Error");
    Wait(1000);
    Stop(true);
  }
}

task main(){
  int in;
  BTCheck(0);
  TextOut(5,LCD_LINE1,"Slave receiving");
  SendResponseNumber(OUT_MBOX,0xFF); //unblock master
  while(true){
    if (ReceiveRemoteNumber(IN_MBOX,true,in) != STAT_MSG_EMPTY_MAILBOX){
      TextOut(0,LCD_LINE3," ");
      NumOut(5,LCD_LINE3,in);
      SendResponseNumber(OUT_MBOX,0xFF);
    }
    Wait(10); //take breath (optional)
  }
}
```

Direkte Befehle

Es gibt ein weiteres, sehr interessante Eigenschaft der Bluetooth Kommunikation: der Master kann den Slave direkt steuern.

Das nachfolgende Programmbeispiel zeigt, wie der Master dem Slave direkte Befehle zum Abspielen von Musik und zur Ansteuerung des Motors sendet. Hierfür wird kein Slave-Programm benötigt, da die Firmware des NXT das Erhalten und Handhaben der Befehle übernimmt.

```
//MASTER
#define BT_CONN 1
#define MOTOR(p,s) RemoteSetOutputState(BT_CONN, p, s, \
OUT_MODE_MOTORON+OUT_MODE_BRAKE+OUT_MODE_REGULATED, \
OUT_REGMODE_SPEED, 0, OUT_RUNSTATE_RUNNING, 0)

sub BTCheck(int conn){
   if (!BluetoothStatus(conn)==NO_ERR){
     TextOut(5,LCD_LINE2,"Error");
     Wait(1000);
     Stop(true);
   }
}

task main(){
  BTCheck(BT_CONN);
  RemotePlayTone(BT_CONN, 4000, 100);
  until(BluetoothStatus(BT_CONN)==NO_ERR);
  Wait(110);
  RemotePlaySoundFile(BT_CONN, "! Click.rso", false);
  until(BluetoothStatus(BT_CONN)==NO_ERR);
  //Wait(500);
  RemoteResetMotorPosition(BT_CONN,OUT_A,true);
  until(BluetoothStatus(BT_CONN)==NO_ERR);
  MOTOR(OUT_A,100);
  Wait(1000);
  MOTOR(OUT_A,0);
}
```

Zusammenfassung

In diesem Kapitel haben wir uns mit den grundlegenden Aspekten der Bluetooth-Kommunikation – Verbindung zweier NXT, Senden und Empfangen der Variablentypen strings und numbers sowie mit dem Senden von Empfangsbestätigungen befasst. Der letzte Aspekt ist wichtig, wenn ein sicheres Nachrichtenprotokoll implementiert werden soll. Zusätzlich haben Sie gesehen, wie Sie mittels direkter Befehle den Slave steuern können.

12 Weitere Befehle

NXC besitzt eine Vielzahl zusätzlicher Befehle. In diesem Kapitel werden drei Typen vorgestellt: Timer-Befehle; Display-Befehle und Befehle zur NXT-Ordnerstruktur

Timer

Der NXT hat einen internen Timer (Zähler) der kontinuierlich läuft. Dieser Timer wird schrittweise um 1/1000 Sekunde erhöht. Den aktuellen Wert des Timers erhält man mit folgendem Befehl: `CurrentTick()` Hier ein Programmbeispiel, dass den Roboter 10 Sekunden lang zufällig umher fahren lässt.

```
task main()
{
  long t0, time;    //Variablendeklaration vom Typ long
  t0 = CurrentTick();
  do
  {
    time = CurrentTick()- t0;
    OnFwd(OUT_AC, 75);
    Wait(Random(1000));
    OnRev(OUT_C, 75);
    Wait(Random(1000));
  }
  while (time<10000);
  Off(OUT_AC);
}
```

Evtl. wollen Sie dieses Programm mit dem aus Kapitel 4 vergleichen, welches exakt dasselbe getan hat. Allerdings ist dieses, mit Nutzung des Timers – wesentlich einfacher.

Timer sind sehr nützlich um den `Wait`-Befehl zu ersetzen. Sie können das Programm für eine gewisse Zeit „anhalten“ indem Sie den Timer zurücksetzen (auf 0 setzen) und anschließend das Programm warten lassen, bis ein bestimmter Wert (hier: `while (time < 10000);`) erreicht wurde. Ebenso kann der Timer auch im Zusammenhang mit Sensoren eingesetzt werden. Das folgende Programm zeigt dies, es lässt einen Roboter umherfahren, bis 10 Sekunden um sind oder bis (währenddessen) der Tastsensor berührt wurde.

```
task main()
{
  long t3;
  SetSensor(IN_1,SENSOR_TOUCH);
  t3 = CurrentTick();
  OnFwd(OUT_AC, 75);
  until ((SENSOR_1 == 1) || ((CurrentTick()-t3) > 10000));
  Off(OUT_AC);
}
```

Vergessen Sie bitte nicht, dass der Timer in 1/1000 Sekunden arbeitet, genau wie der wait-Befehl.

Punkt-Matrix-Display

Der NXT-Stein besitzt ein schwarz-weiß LCD Display mit einer Auflösung von 100x64 Bildpunkten. Es gibt zahlreiche API-Funktionen um Text, Zahlen, Punkte, Linien, Rechtecke, Kreise und sogar Bitmap Bilder (.ric-Format) auf dem Display anzuzeigen. Das nachfolgende Beispiel versucht all diese Optionen abzudecken. Hinweis: der Bildpunkt (0,0) stellt den der äußerste Bildpunkt links unten dar.

```
#define X_MAX 99
#define Y_MAX 63
#define X_MID (X_MAX+1)/2
#define Y_MID (Y_MAX+1)/2

task main(){
  int i = 1234;
  TextOut(15,LCD_LINE1,"Display", true);
  NumOut(60,LCD_LINE1, i);
  PointOut(1,Y_MAX-1);
  PointOut(X_MAX-1,Y_MAX-1);
  PointOut(1,1);
  PointOut(X_MAX-1,1);
  Wait(200);
  RectOut(5,5,90,50);
  Wait(200);
  LineOut(5,5,95,55);
  Wait(200);
  LineOut(5,55,95,5);
  Wait(200);
  CircleOut(X_MID,Y_MID-2,20);
  Wait(800);
  ClearScreen();
  GraphicOut(30,10,"faceclosed.ric"); Wait(500);
  ClearScreen();
  GraphicOut(30,10,"faceopen.ric");
  Wait(1000);
}
```

All diese Funktionen sollten eigentlich selbsterklärend sein, hier aber ein Beschreibung der dazugehörigen Parameter:

`ClearScreen()`	`Löscht den Bildschirm`
`NumOut(x, y, number);`	Spezifizieren der Koordinaten und Nummer
`TextOut(x, y, string)`	wie NumOut, mit Textausgabe
`GraphicOut(x, y, filename)`	zeigt ein bitmap .ric Bild an
`CircleOut(x, y, radius)`	Ausgabe eines Kreises, unter Angabe der Koordinaten des Kreismittelpunkts und des Radius
`LineOut(x1, y1, x2, y2)`	Zeichnet eine Line von den Punkten (x1, x2) zu (x2, y2)
`PointOut(x, y)`	“Zeichnet” einen Punkt auf das Display
`RectOut(x, y, width, height)`	Zeichnet ein Rechteck mit dem Eckpunkt (x,y) und unter Angabe der Breite und Höhe
`ResetScreen()`	Setzt den Bildschirm zurück

Dateisystem

Der NXT kann gespeicherte Dateien lesen und schreiben. Somit können Sensordaten (Datalog) aufgezeichnet werden oder Zahlen während des Ausführens eines Programms gelesen werden. Einziges Limit ist die Größe des NXT-Speichers. Mit Hilfe der Funktionen der NXC-API können Dateien manipuliert (erzeugt, umbenannt, gelöscht, gesucht) werden, ebenso können *strings*, *numbers* und einzelne Bytes innerhalb von Dateien gelesen bzw. geschrieben werden.

Das erste Programm löscht Dateien mit dem von uns gewählten Namen. Üblicherweise sollte die Existenz dieser Datei vorher geprüft werden und ggf. manuell gelöscht werden bzw. umbenannt werden. In unserem einfachen Beispiel ist dies aber kein Problem Mit `CreateFile("Danny.txt", 512, fileHandle)`erzeugen wir eine Datei, geben dieser einen Namen (hier Danny.txt), eine Größe (512) und eine Handhabung für die NXT-Firmware, damit dieser der Datei einen eigene Nummer zuweisen kann.

Im Anschluss daran, werden strings erzeugt und diese dann mit einem Zeilenrücklauf: `WriteLnString(fileHandle,string,bytesWritten)` in die Datei geschrieben. Alle im Befehl WriteLnString angegebenen Parameter müssen Variablen sein. Abschließend wird die Datei geschlossen und umbenannt. Hinweis: Eine Datei muss erst geschlossen werden, damit andere Operationen (löschen, umbenennen etc.) auf dieser Datei stattfinden können. Bevor eine Datei gelesen werden kann, muss diese mit: `OpenFileRead()` geöffnet werden.

```
#define OK LDR_SUCCESS

task main(){
  byte fileHandle;
  short fileSize;
  short bytesWritten;
  string read;
  string write;
  DeleteFile("Danny.txt");
  DeleteFile("DannySays.txt");
  CreateFile("Danny.txt", 512, fileHandle);
  for(int i=2; i<=10; i++ ){
    write = "NXT is cool ";
    string tmp = NumToStr(i);
    write = StrCat(write,tmp," times!");
    WriteLnString(fileHandle,write, bytesWritten);
  }
  CloseFile(fileHandle);
  RenameFile("Danny.txt","DannySays.txt");
}
```

Das Ergebnis kann unter BricxCC → Tools → NXT Explorer upload (hochladen der Datei) DannySays.txt eingesehen werden.
Im nächsten Beispiel wird eine ASCII –Tabelle erstellt.

```
task main(){
  byte handle;
  if (CreateFile("ASCII.txt", 2048, handle) == NO_ERR) {
    for (int i=0; i < 256; i++) {
      string s = NumToStr(i);
      int slen = StrLen(s);
      WriteBytes(handle, s, slen);
      WriteLn(handle, i);
    }
    CloseFile(handle);
  }
}
```

Dieses einfach Beispiel erzeugt eine Datei, tritt dabei kein Fehler auf, schreibt es eine Zahl von 0 bis 255 (vorher wird diese zu einem *string* konvertiert) mit `WriteBytes(handle, s, slen)` – dies ist eine weitere Möglichkeit einen *string* ohne Zeilenrücklauf zu erzeugen – anschließend wird mit `WriteLn(handle, value)` eine Zahl geschrieben und ein Zeilenrücklauf hinzugefügt. Das Ergebnis kann mit einem Texteditor (Windows Notepad) eingesehen werden. Das Ergebnis lässt sich wie folgt erläutern: die Zahlen werden als *string* in ein für Menschen lesbares Format geschrieben, während die Nummer als Hexadezimalwert als ASCII-Code interpretiert und angezeigt wird.

Zwei wichtige Funktionen sollen nun noch erläutert werden:
`ReadLnString` → um *strings* aus der Datei lesen zu können
`ReadLn` → um Zahlen lesen zu können

Das folgende Programmbeispiel erläutert beide Befehle und zeigt deren Anwendung: In der main-Task wird die Subroutine `CreateRandomFile` aufgerufen. Diese Subroutine erzeugt eine Datei und beschreibt diese mit Zufallszahlen (als *string*). Anschließend wird diese Datei zum lesen geöffnet, liest mit: `ReadLnString` eine Zeile bis zum Zeilenende und gibt diesen Text auf dem Display aus.

Die Subroutine `CreateRandomFile` generiert eine vordefinierte Anzahl Zufallszahlen, konvertiert sie zu einem *string* und schreibt diese anschließend in die Datei.

Die Funktion benötigt ein *file handle* und einen *string* als Übergabeparameter, nach dem Funktionsaufruf beinhaltet der *string* eine Textzeile und die Funktion gibt ggf. einen Rückgabe „Fehlercode“ an, um das Dateiende anzuzeigen.

```
#define FILE_LINES 10

sub CreateRandomFile(string fname, int lines){
  byte handle;
  string s;
  int bytesWritten;
  DeleteFile(fname);
  int fsize = lines*5;
  //create file with random data
  if(CreateFile(fname, fsize, handle) == NO_ERR) {
    int n;
    repeat(FILE_LINES) {
      int n = Random(0xFF);
      s = NumToStr(n);
      WriteLnString(handle,s,bytesWritten);
    }
    CloseFile(handle);
  }
}

task main(){
  byte handle;
  int fsize;
  string buf;
  bool eof = false;
  CreateRandomFile("rand.txt",FILE_LINES);
  if(OpenFileRead("rand.txt", fsize, handle) == NO_ERR) {
    TextOut(10,LCD_LINE2,"Filesize:");
    NumOut(65,LCD_LINE2,fsize);
    Wait(600);
    until (eof == true){ // read the text file till the end
      if(ReadLnString(handle,buf) != NO_ERR) eof = true;
      ClearScreen();
      TextOut(20,LCD_LINE3,buf);
      Wait(500);
    }
  }
  CloseFile(handle);
}
```

Das letzte Programmbeispiel soll zeigen, wie Zahlen aus deiner Datei gelesen werden können.
Zusätzlich veranschaulicht dieses Programm ein Beispiel bedingter Kompilierung. Am Programmanfang befindet sich eine Definition, die weder ein Marko noch einen Platzhalter darstellt, es wird einfach `INT` definiert.

Anschließend folgt eine Präprozessor-Anweisung

```
#ifdef INT
…Code…
#endif
```

Diese teilt dem Compiler mit, die Programmzeilen zwischen den beiden Anweisungen zu kompilieren, wenn `INT` zuvor definiert wurde. Sprich wenn `INT` vorab definiert wurde, wird die main-Task (zwischen den `#ifdef` und `#endif` Anweisungen) kompiliert, ebenso wenn LONG vor Abfragen der `#ifdef LONG` Anweisung definiert wurde, wird die zweite main-Task kompiliert.

Diese Methode zeigt, wie in einem Programm beide Typen: int (16 bit) und long (32 bit) mit dem gleichen Dateiaufruf `ReadLn(handle,val)` gelesen werden können.

Diese Funktion akzeptiert die Parameter *handle* und eine numerische Variable, als Rückgabewert ist ein Error-Code.

Diese Funktion liest 2 Bytes aus der Datei, falls der Übergabeparameter als `int` deklariert wurde, wurde der Parameter als `long` deklariert werden 4 Bytes gelesen. Ebenso können boolsche-Variablen geschrieben und gelesen werden.

```
#define INT // INT or LONG

#ifdef INT
task main () {
  byte handle, time = 0;
  int n, fsize,len, i;
  int in;
  DeleteFile("int.txt");
  CreateFile("int.txt",4096,handle);
  for (int i = 1000; i<=10000; i+=1000){
    WriteLn(handle,i);
  }
  CloseFile(handle);
  OpenFileRead("int.txt",fsize,handle);
  until (ReadLn(handle,in)!=NO_ERR){
    ClearScreen();
    NumOut(30,LCD_LINE5,in);
    Wait(500);
  }
  CloseFile(handle);
}
#endif

#ifdef LONG
task main () {
  byte handle, time = 0;
  int n, fsize,len, i;
  long in;
  DeleteFile("long.txt");
  CreateFile("long.txt",4096,handle);
  for (long i = 100000; i<=1000000; i+=50000){
     WriteLn(handle,i);
  }
  CloseFile(handle);
  OpenFileRead("long.txt",fsize,handle);
  until (ReadLn(handle,in)!=NO_ERR){
    ClearScreen();
    NumOut(30,LCD_LINE5,in);
    Wait(500);
  }
  CloseFile(handle);
}
#endif
```

Zusammenfassung

Dieses letzte Kapitel hat Sie in die weiterführenden Eigenschaften des NXT eingeführt; wie hochauflösende TIMER, NXT-Display und Dateistruktur des NXT.

13 Schlussbemerkung

Wenn Sie dieses Tutorial nochmals mit Ihren eigenen Ideen durchgearbeitet haben können Sie sich durchaus als Expertin und Experte in Sachen NXC bezeichnen. Wenn nicht, wird es nun Zeit selbst das NXT-System zu erforschen.

Dieses Tutorial betrachtet nicht alle Aspekte der Programmierumgebung BricxCC. Lesen Sie den NXC Guide Kapitel für Kapitel durch. Darüber hinaus befindet sich NXC immer noch in der Entwicklung, zukünftige Versionen können möglicherweise inkompatible zusätzliche Funktionalitäten mit sich bringen. Viele Konzepte der Programmierung wurden innerhalb dieses Tutorials nicht behandelt. Vor allem wurde nicht auf lernbasiertes Verhalten von Robotern und anderen Konzepte der Künstlichen Intelligenz eingegangen.

Es ist ebenso möglich, den NXT Roboter direkt vom PC aus zu steuern. Dieses setzt aber voraus, dass Sie ein Programm in den Programmiersprachen C++, Visual Basic, Java oder Delphi schreiben. Ebenso möglich ist es, derartige Programme in Kombination mit NXC auf dem NXT-Stein laufen zu lassen. Diese Kombinationen sind sehr mächtig. Sollten Sie an derartigen Programmkonzepten Interesse haben, ist es am besten, Sie starten mit dem Download des Fantom SDK und Open Source Dokumenten vom NXTreme Bereich der Lego MindStorms Webseiten:

http://mindstorms.lego.com/Overview/NXTreme.aspx

Das Internet stellt zudem eine perfekte Quelle zusätzlicher Informationen dar. Weitere wichtige Einstiegsseiten für das NXT-System ist LUGNET das inoffizielle LEGO – Users Group Netzwerk.

http://www.lugnet.com/robotics/nxt