# NQC Programmieranleitung

**Version 2.0 rev 2, von Dave Baum**

Titel des englischen Originals:
*NQC Programmer's Guide*

## Einleitung

NQC steht für *Not Quite C* ("nicht ganz C") und ist eine einfache Programmiersprache für das LEGO RCX. Der Präprozessor und die Kontrollstrukturen sind ähnlich wie bei der Programmiersprache C. NQC ist keine Allzweckprogrammiersprache - es gibt viele Einschränkungen, die von Beschränkungen der Standard--RCX-Firmware stammen.

Obwohl NQC speziell für das RCX geschaffen wurde, gibt es doch eine logische Trennung zwischen der NQC-Sprache und der RCX-API, die das RCX steuert. Diese Trennung ist in einigen Fällen nicht ganz scharf, zum Beispiel bei der Multi--tasking-Unterstützung. Im allgemeinen ist die Sprache durch den Übersetzer festgelegt und eine besondere NQC-Datei definiert die RCX-API in Form von NQC--Sprachanweisungen.

Diese Programmieranleitung beschreibt die NQC-Sprache und das RCX-API. Kurz, es stellt die Informationen bereit, die benötigt werden um ein NQC--Programm zu schreiben. Da es mehrere Werkzeuge und Programmierumgebungen für NQC gibt, beschreibt diese Anleitung nicht deren Verwendung. Schauen Sie wegen weiterer Informationen in die begleitende Dokumentation zu dem jeweiligen Werkzeug.

Die neuesten Informationen und die gültige Dokumentation für NQC finden Sie auf der Web-Seite

http://www.enteract.com/~dbaum/nqc

# Schreibweisen

## *Kommentare*

In NQC gibt es zwei Arten von Kommentaren. Die erste Form (traditioneller C--Kommentar) beginnt mit /* und endet mit */. Diese Kommentare können mehrere Zeilen umfassen, dürfen aber nicht geschachtelt werden:

```
/* dies ist ein Kommentar */

/* dies ist ein zweizeiliger
   Kommentar */

/* noch ein Kommentar ...
   /* Versuch einer Schachtelung ...
      Ende des inneren Kommentars ...*/
   dieser Text gehört nicht mehr zum Kommentar! */
```

Die zweite Kommentarform beginnt mit // und endet mit dem Zeilenende (gelegentlich auch C++ -Kommentar genannt).

```
// ein einzeiliger Kommentar
```

Kommentare werden vom Übersetzer nicht beachtet. Sie haben nur den Zweck, dem Programmierer die Kommentierung des Quellcodes zu erlauben.

## *Leerzeichen*

Leerzeichen (eigentliche Leerzeichen (engl. space), Tabulatoren und Zeilenvorschübe) werden dazu verwendet, die Programm- und Textbestandteile voneinander zu trennen und um das Programm lesbarer zu machen. Solange die Textbestandteile unterscheidbar bleiben, hat das Hinzufügen oder Entfernen von Leerzeichen keine Auswirkung auf die Bedeutung des Programms. Zum Beispiel haben die folgenden Programmzeilen dieselbe Bedeutung:

```
x=2;
x   =  2  ;
```

Einige der C++ -Operatoren bestehen aus mehreren Zeichen. Innerhalb dieser Operatoren dürfen keine Leerzeichen eingefügt werden. In dem nachfolgenden Beispiel wird in der ersten Zeile der Rechtsshift-Operator ('>>') verwendet. In der

zweiten Zeile bewirkt das eingefügte Leerzeichen, daß die Zeichen '>' als zwei unterschiedliche Textbestandteile betrachtet werden und deshalb zu einem Fehler beim Übersetzen führen.

```
x = 1 >> 4;  // setze x zu 1, um 4 Bits nach rechts geschoben
x = 1 > > 4; // Fehler
```

### *Numerische Konstanten*

Numerische Konstanten können dezimal oder hexadezimal geschrieben werden. Dezimalkonstanten bestehen aus einer oder mehreren Dezimalziffen. Hexadezimale Konstanten beginnen mit `0x` oder `0X` gefolgt von einer oder mehreren Hexadezimalziffen.

```
x = 10;   // setze x zu 10
x = 0x10; // setze x zu 16 (10 hexadezimal)
```

### *Namen und Schlüsselwörter*

Namen werden für die Benennung von Variablen, Tasks und Funktionen verwendet. Das erste Zeichen eines Namens muß ein großer oder ein kleiner Buchstabe oder ein Unterstrich ('_') sein. Weiteren Zeichen können Buchstaben, Ziffern oder Unterstriche sein.

Eine Reihe von Namen ist der NQC-Sprache vorbehalten. Diese reservierten Bezeichnungen werden Schlüsselwörter genannt und können nicht als Namen verwendet werden. Hier eine vollständige Liste dieser Schlüsselwörter:

| | | |
|---|---|---|
| **__sensor** | **false** | **stop** |
| **abs** | **if** | **sub** |
| **asm** | **inline** | **task** |
| **break** | **int** | **true** |
| **const** | **repeat** | **void** |
| **continue** | **return** | **while** |
| **do** | **sign** | |
| **else** | **start** | |

## Programmstruktur

Ein NQC-Programm besteht aus Programmblöcken und aus globalen Variablen. Es gibt drei unterschiedliche Blockarten: Tasks, inline-Funktionen und Unterprogramme. Jede Blockart hat besondere Eigenschaften und Beschränkungen, aber sie haben alle einen gemeinsamen Aufbau.

### *Tasks*

RCX unterstützt Multitasking, deshalb entspricht eine NQC-Task einer RCX-Task. Tasks werden unter Verwendung des Schlüsselwortes `task` und der folgenden Syntax definiert:

```
task name()
{
          // hier stehen die Programmanweisungen der Task (Rumpf)
}
```

Der Name der Task kann irgendein zulässiger Bezeichner sein. Ein Programm besteht mindestens aus einer Task, "main" genannt, die aufgerufen wird, wenn das Programm gestartet wird. Ein Programm kann bis zu 9 weitere Tasks enthalten.

Der Rumpf einer Task besteht aus einer Folge von Anweisungen. Task können mit den Anweisungen `start` und `stop` gestartet und angehalten werden (Beschreibung im Abschnitt *Anweisungen*). Es gibt auch einen RCX-API-Befehl, `StopAllTasks`, der alle laufenden Tasks anhält.

### *Inline-Funktionen*

Es ist oft hilfreich, eine Anzahl von Anweisungen in eine einzelne Funktion zusammenzupacken, die dann bei Bedarf aufgerufen werden kann. NQC kennt Funktionen mit Argumenten, aber keine Rückgabewerte. Funktionen werden nach folgender Syntax definiert:

```
void name( Argumentliste )
{
      // Rumpf der Funktion
}
```

Das Schlüsselwort `void` ist ein Überbleibsel aus der NQC-Abstammung - in der Sprache C muß für Funktionen der Datentyp des Rückgabewertes angegeben werden. Funktionen, die keinen Wert zurückgeben, besitzen den Rückgabetyp `void`. Da Rückgabewerte in NQC nicht zur Verfügung stehen, müssen alle Funktionen mit dem Schlüsselwort `void` vereinbart werden.

Die Argumentliste kann leer sein, oder sie kann eine oder mehrere Argumentdefinitionen enthalten. Ein Argument wird durch eine Typangabe, gefolgt von einem Namen, definiert. Mehrere Argumente werden durch Kommata getrennt. Alle Werte im RCX werden durch 16 Bit breite, vorzeichenbehaftete ganze Zahlen dargestellt. NQC unterstützt jedoch vier unterschiedliche Argumenttypen, die unterschiedlichen Übergabearten und Einschränkungen entsprechen:

| Typ | Bedeutung | Einschränkung |
|---|---|---|
| `int` | Wertübergabe | keine |
| `const int` | Wertübergabe | nur Konstanten können verwendet werden |
| `int&` | Adreßübergabe | nur Variablen können verwendet werden |
| `const int &` | Adreßübergabe | die Funktion kann den Wert des Arguments nicht ändern |

Argumente vom Typ `int` werden als Wert an die aufgerufene Funktion übergeben. Das heißt, daß der Übersetzer eine Hilfsvariable für den Wert des Arguments anlegen muß. Da die Funktion mit einer Kopie des tatsächlichen Arguments arbeitet, wird eine Veränderung des Wertes dieser Kopie für den Aufrufer nicht sichtbar. Im untenstehenden Beispiel ändert die Funktion `foo` den Wert des Arguments auf 2. Dies ist vollkommen zulässig, da aber die Funktion `foo` mit einer Kopie des tatsächlichen Arguments arbeitet, bleibt die Variable `y` in der task `main` unverändert.

```
void foo(int x)
{
      x = 2;
}

task main()
{
      int y = 1;         // y ist gleich 1
      foo(y);            // y ist immer noch 1!
}
```

Für den zweiten Argumenttyp, `const int`, findet ebenfalls Wertübergabe statt, aber mit der Einschränkung, daß nur konstante Werte (z.B. Zahlen) übergeben werden dürfen. Das ist ziemlich wichtig, da es eine Reihe von RCX-Funktionen gibt, die nur mit konstanten Argumenten arbeiten.

```
void foo(const int x)
{
      PlaySound(x);  // in Ordnung
      x = 1;         // Fehler - kann Argument nicht verändern
}

task main()
{

      foo(2);     // in Ordnung
      foo(4*5);   // in Ordnung - Ausdruck ist konstant
      foo(x);     // Fehler - x ist keine Konstante
}
```

Der dritte Argumenttyp, `int &`, übergibt anstatt des Wertes die Adresse des Arguments . Das erlaubt der aufgerufene Funktion den Wert des Arguments zu ändern, so daß anschließend die Änderung für den Aufrufer wirksam wird. Als `int &` -Argumente einer Funktion können jedoch nur Variablen verwendet werden:

```
void foo(int &x)
{
      x = 2;
}

task main()
{
      int y = 1;  // y ist gleich 1

      foo(y);     // y ist nun gleich 2
      foo(2);     // Fehler - nur Variablen erlaubt
}
```

Der letzte Argumenttyp, `const int &`, ist ziemlich ungewöhnlich. Er übergibt ebenfalls eine Adresse, aber mit der Einschränkung, daß die aufgerufene Funktion den Wert nicht verändern kann. Wegen dieser Einschränkung kann der Übersetzer alles (nicht nur Variablen) an Funktionen übergeben, die diesen Argumenttyp

verwenden. Im allgemeinen ist das die schnellste Art Argumente in NQC zu übergeben.

Es gibt einen wichtigen Unterschied zwischen `int`-Argumenten und `const int &` -Argumenten. Ein `int`-Argument wird als Wert übergeben, d.h. im Falle von dynamischen Daten (zum Beispiel eines Sensorwertes), daß der Wert einmal gelesen und dann übergeben wird. Bei einem `const int &` -Argument wird der Wert jedesmal gelesen, wenn er verwendet wird:

```
void foo(int x)
{
    if (x==x)  // das ist immer wahr
          PlaySound(SOUND_CLICK);
}
void bar(const int x)
{
    if (x==x)  // muß nicht wahr sein; ein Wert könnte sich ändern
          PlaySound(SOUND_CLICK);
}
task main()
{
    foo(SENSOR_1);    // gibt einen Ton aus
    bar(2);           // gibt einen Ton aus
    bar(SENSOR_1);    // gibt vielleicht keinen Ton aus
}
```

Funktionen müssen mit der richtigen Anzahl (und den richtigen Typen) der Argumente aufgerufen werden. Das folgende Beispiel zeigt verschiedene zulässige und falsche Aufrufe der Funktion `foo`:

```
void foo(int bar, const int baz)
{
    // hier wird irgendwas getan ...
}

task main()
{
    int x;      // Vereinbarung der Variablen x

    foo(1, 2);  // richtig
    foo(x, 2);  // richtig
    foo(2, x);  // Fehler - 2. Argument nicht konstant!
    foo(2);     // Fehler - falsche Argumentanzahl !
}
```

NQC-Funktionen werden immer als inline-Funktionen erweitert. Das bedeutet, daß für jeden Funktionsaufruf eine Kopie des Funktionscodes im Programm vorhanden ist. Inline-Funktionen können bei unüberlegter Verwendung zu einem großen Code-Umfang führen.

### *Unterprogramme*

Im Gegensatz zu inline-Funktionen wird ist der Code eines Unterprogramms nur einmal vorhanden und kann von mehreren Aufrufern verwendet werden. Dadurch sind Unterprogramme wesentlich speichereffizienter als inline-Funktionen. Wegen einiger Beschränkungen in RCX haben Unterprogramm allerdings deutliche Einschränkungen. Erstens können Unterprogramme keine Argumente verwenden. Zweitens kann ein Unterprogramm nicht ein anderes Unterprogramm aufrufen. Schließlich dürfen höchsten 8 Unterprogramme in einem Programm definiert sein. Desweiteren darf ein Unterprogramm keine lokalen Variablen besitzen, wenn es von mehreren Tasks aufgerufen wird. Wegen dieser einschneidenden Einschränkungen ist die Verwendung von Unterprogrammen wenig wünschenswert. Deshalb sollte ihr Gebrauch auf die Fälle beschränkt bleiben, in denen auf die erzielbare Speicherplatzersparnis nicht verzichtet werden kann. Der Aufbau eines Unterprogramms sieht wie folgt aus:

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

### *Variablen*

Alle Variablen im RCX besitzen denselben Datentyp, nämlich 16-Bit vorzeichenbehaftete ganze Zahl ( 16 bit signed integer). RCX erlaubt bis zu 32 solcher Variablen. Dieser Vorrat wird von NQC in mehreren unterschiedlichen Weisen genutzt. Variablen werden mit dem Schlüsselwort `int` vereinbart, dem eine durch Kommata getrennte Liste von Variablennamen folgt, die durch einen Strichpunkt ('`;`') abgeschlossen wird. Bei Bedarf kann jeder Variablen mit Hilfe eines Gleichheitszeichens ('=') nach dem Namen ein Anfangswert zugewiesen werden. Hier einige Beispiele:

```
int x;      // vereinbare x
int y,z;    // vereinbare y und z
int a=1,b;  // vereinbare a und b, setze a auf 1
```

Globale Variablen sind im gesamten Programm gültig und werden außerhalb von Blöcken vereinbart. Wenn sie einmal vereinbart sind, können sie in allen Tasks, Funktionen und Unterprogrammen verwendet werden. Ihre Gültigkeit beginnt mit der Vereinbarung und endet am Programmende.

Lokale Variablen können innerhalb von Tasks, Funktionen und in gewissen Fällen innerhalb von Unterprogrammen vereinbart werden. Diese Variablen haben nur in dem Block Gültigkeit in dem sie vereinbart sind. Genauer gesagt beginnt ihre Gültigkeit mit der Vereinbarung und endet am Blockende. Ein Block wird durch mehrere Anweisungen gebildet, die durch geschweifte Klammern ( `{` und `}` ) zusammengefaßt werden:

```
int x;  // x ist global

task main()
{
      int y;      // y ist lokal in der Task main
      x = y;      // in Ordnung
      {                 // Blockanfang
            int z;      // vereinbare lokales z
            y = z;      // in Ordnung
      }
      y = z; // Fehler - z ist nicht im Gültigkeitsbereich
}

task foo()
{
      x = 1; // in Ordnung
      y = 2; // Fehler - y ist nicht global
}
```

In vielen Fällen muß NQC eine Hilfsvariable für den internen Gebrauch belegen. In einigen Fällen ist eine Hilfsvariable zur Aufnahme eines Zwischenergebnisses bei einer Berechnung erforderlich. In anderen Fällen enthält sie einen Argumentwert, der an eine Funktion übergeben wird. Diese Hilfsvariablen verringern die Menge der zur Verfügung stehenden Variablen im RCX. NQC versucht deshalb so

sparsam wie möglich mit Hilfsvariablen umzugehen und verwendet diese, wenn möglich, auch mehrfach.

## Anweisungen

Der Rumpf eines Blockes (Task, Funktion, Unterprogramm) ist aus Anweisungen zusammengesetzt. Anweisungen werden mit einem Strichpunkt (';') abgeschlossen.

### *Variablenvereinbarung*

Eine Variablenvereinbarungen, wie sie im vorangehenden Abschnitt beschrieben wurde, ist eine mögliche Art von Anweisung. Sie vereinbart eine lokale Variable (mit optionalem Anfangswert) zum Gebrauch innerhalb eines Blockes. Die Syntax für die Variablenvereinbarung lautet:

```
int Variablen;
```

wobei *Variablen* eine durch Kommata getrennte Liste von Namen ist, denen zusätzlich Anfangswertzuweisungen beigefügt sein können:

```
name[=Ausdruck]
```

### *Zuweisungen*

Einmal vereinbart, können Variablen Werte von Ausdrücken zugewiesen werden:

```
Variable Zuweisungsoperator Ausdruck;
```

Es gibt neun unterschiedliche Zuweisungsoperatoren. Der einfachste Zuweisungsoperator, '=', weist einfach der Variablen den Wert des Ausdrucks zu. Die anderen Operatoren verändern den Wert der Variablen in unterschiedlicher Weise, wie in der nachfolgenden Tabelle gezeigt wird.

| Operator | Wirkung |
|---|---|
| = | weist den Wert des Ausdrucks zu |
| += | addiert den Wert des Ausdrucks zum Wert der Variablen |
| -= | subtrahiert den Wert des Ausdrucks vom Wert der Variablen |
| *= | multipliziert den Wert der Variablen mit dem Wert des Ausdrucks |
| /= | dividiert den Wert der Variablen durch den Wert des Ausdrucks |
| &= | bitweise UND-Verknüpfung des Variablenwertes mit dem Ausdruck |
| \|= | bitweise ODER-Verknüpfung des Variablenwertes mit dem Ausdruck |
| \|\|= | weist der Variablen den Absolutwert des Ausdrucks zu |
| +-= | weist der Variablen das Vorzeichen (-1,+1,0) des Ausdrucks zu |

Einige Beispiele:

```
x = 2;       // setzt x auf 2
y = 7;       // setzt y auf 7
x += y;      // x ist 9, y ist immer noch 7
```

## *Kontrollstrukturen*

Die einfachste Kontrollstruktur ist ein Block. Das ist einen Folge von Anweisungen, die in geschweiften Klammern ( `{` und `}` ) eingeschlossen sind:

```
{
      x = 1;
      y = 2;
}
```

Obwohl das nicht sehr bedeutend zu sein scheint, spielt es eine entscheidende Rolle in komplexeren Kontrollstrukturen. Viele Kontrollstrukturen verlangen eine einzelne Anweisung als Rumpf. Durch die Verwendung eines Blockes kann dieselbe Kontrollstruktur mehrere Anweisungen enthalten.

Die `if`-Anweisung wertet eine Bedingung aus. Wenn die Bedingung wahr ist, wird die nachstehende Anweisung (Folgerung) ausgeführt. Wenn die Bedingung falsch ist kann eine zweite, optionale Anweisung (Alternative) ausgeführt werden. Die beiden Schreibweisen werden nachstehend gezeigt.

```
if (Bedingung) Folgerung
if (Bedingung) Folgerung else Alternative
```

Beachten Sie, daß die Bedingung in Klammern eingeschlossen wird. Beispiele werden unten gezeigt. Beachten Sie weiterhin, wie im letzten Beispiel ein Block dazu verwendet wird, um die Ausführung von zwei Anweisungen als Folgerung zu ermöglichen.

```
if (x==1) y = 2;
if (x==1) y = 3; else y = 4;
if (x==1) { y = 1; z = 2; }
```

Eine `while`-Anweisung wird zum Aufbau einer bedingten Schleife verwendet. Die Bedingung wird ausgewertet, wenn diese wahr ist wird der Rumpf ausgeführt, anschließend wird wieder die Bedingung ausgewertet. Dieser Vorgang wird solange fortgesetzt bis die Bedingung falsch wird ( oder eine `break`-Anweisung ausgeführt wird). Die Syntax einer while-Schleife sieht wie folgt aus:

```
while (condition) body
```

Als Schleifenrumpf wird häufig ein Block verwendet:

```
while(x < 10)
{
      x = x+1;
      y = y*2;
}
```

Eine Variante der `while`-Schleife ist die `do-while`-Schleife. Die Syntax lautet:

```
do Rumpf while (Bedingung)
```

Der Unterschied zwischen einer `while`-Schleife und einer `do-while`-Schleife ist der, daß die `do-while`-Schleife ihren Rumpf mindestens einmal ausführt (die Bedingung wird immer nach der Ausführung des Rumpfes geprüft), während die `while`-Schleife ihren Rumpf überhaupt nicht auszuführen braucht (die Bedingung wird immer vor der Ausführung des Rumpfes geprüft).

Die `repeat`-Anweisung führt ihren Rumpf mehrmals aus:

```
repeat (Ausdruck) Rumpf
```

Der Ausdruck legt fest, wie oft der Rumpf ausgeführt wird. Zu beachten ist, daß der Ausdruck nur ein einziges Mal ausgewertet wird. Anschießend wird der Rumpf

entsprechend oft ausgeführt. Das ist unterschiedlich sowohl von der `while`--Schleife, als auch von der `do-while`-Schleife, die ihre Bedingung bei jedem Durchlauf auswerten.

In NQC ist zusätzlich das `until`-Makro definiert. Es stellt eine bequeme Alternative zur `while`-Schleife dar. Die tatsächlich verwendete Definition lautet:

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

Mit anderen Worten setzt `until` die Wiederholung solange fort, bis die Bedingung wahr wird. Es wird meistens in Verbindung mit einem leeren Rumpf verwendet:

```
until(SENSOR_1 == 1);   // warten, bis der Taster gedrückt wurde
```

### *Weitere Anweisungen*

Ein Funktionsaufruf ( oder Unterprogrammaufruf) ist einen Anweisung der Form:

```
Name(Argumente);
```

Die Argumentliste ist eine Liste von Ausdrücken, die durch Kommata getrennt sind. Die Anzahl und die Datentypen der übergebenen Argumente müssen die gleichen sein, die in der Definition der Funktion angegeben sind.

Tasks können mit den folgenden Anweisungen gestartet oder angehalten werden:

```
start task_name;
stop task_name;
```

Im Innern von Schleifen (z.B. der `while`-Schleife) kann die `break`-Anweisung zum Beenden der Schleife verwendet werden. Die `continue`-Anweisung kann dazu verwendet werden, zum Anfang des nächsten Schleifendurchlaufs zu springen.

```
break;
continue;
```

Mit der `return`-Anweisung ist es möglich, eine Funktion vor dem Erreichen ihres Endes zu verlassen.

```
return;
```

Außerdem ist jeder Ausdruck zulässig, wenn er durch einen Strichpunkt abgeschlossen ist. Eine derartige Verwendung kommt selten vor, da der Wert eines derartigen Ausdruckes nirgends verwendet wird. Die einzigen Ausnahmen sind Ausdrücke, die den Inkrement-Operato (++) oder den Dekrement-Operator (--) verwenden.

```
x++;
```

Die leere Anweisung (ein einzelner Strichpunkt) ist ebenfalls eine zulässige Anweisung.

## Ausdrücke und Bedingungen

In C wird zwischen Ausdrücken und Bedingungen nicht unterschieden. In NQC sind das jedoch zwei unterschiedliche syntaktische Einheiten. Die für Ausdrücke zulässigen Operationen können nicht für Bedingungen verwendet werden und umgekehrt.

Ausdrücke können an Variablen zugewiesen, als Funktionsargumente verwendet und als Wiederholungsangabe in `repeat`-Anweisung verwendet werden. Bedingungen werden in den meisten bedingten Kontrollanweisungen verwendet ( `if`, `while`, usw.)

### *Ausdrücke*

Werte sind die einfachste Art von Ausdrücken. Kompliziertere Ausdrücke werden aus Werten mit Hilfe unterschiedlicher Operatoren zusammengesetzt. Die NQC--Sprache kennt nur zwei Arten an Standardwerten: numerische Konstanten und Variablen. Das RCX-API kennt weitere Werte, die zu unterschiedlichen RCX--Einrichtungen wie Sensoren und Zählern passen.

In RCX sind numerische Konstanten durch 16 Bit breite vorzeichenbehaftete ganze Zahlen dargestellt (16 bit signed integers). NQC verwendet intern 32-Bit-

Arithmetik (mit Vorzeichen) zur Bewertung konstanter Ausdrücke und reduziert bei der Erzeugung von RCX-Code auf 16 Bit. Numerische Konstanten können entweder dezimal (z.B. `123`) oder hexadezimal (z.B. `0xABC`) geschrieben werden. Zur Zeit finden bei Konstanten kaum Bereichsüberprüfungen statt, so daß die Verwendung von Werten, die außerhalb der Bereichsgrenzen liegen, ungewöhnliche Auswirkungen haben können.

Werte können durch Operatoren verknüpft werden. Einige der Operatoren dürfen nur zur Bewertung konstanter Ausdrücke verwendet werden, d.h. ihre Operanden müssen entweder Konstanten sein oder sie müssen Ausdrücke sein, die nur aus Konstanten bestehen. Die Operatoren sind nachstehend mit fallender Priorität aufgeführt.

| Operator | Beschreibung | Bindung | Einschränkung | Beispiel |
|---|---|---|---|---|
| `abs()` | Absolutwert | nicht vorhanden | | `abs(x)` |
| `sign()` | Vorzeichen des Operanden | nicht vorhanden | | `sign(x)` |
| `++` | Inkrement | links | nur Variablen | `x++ or ++x` |
| `--` | Dekrement | links | nur Variablen | `x-- or --x` |
| `-` | unäres Minus (Vorzeichen) | rechts | | `-x` |
| `~` | Bitweise Negation (unär) | rechts | nur Konstante | `~123` |
| `*` | Multiplikation | links | | `x * y` |
| `/` | Division | links | | `x / y` |
| `%` | Modulo | links | nur Konstante | `123 % 4` |
| `+` | Addition | links | | `x + y` |
| `-` | Subtraktion | links | | `x - y` |
| `<<` | Linksshift-Operator | links | nur Konstante | `123 << 4` |
| `>>` | Rechtsshift-Operator | links | nur Konstante | `123 >> 4` |
| `&` | bitweise UND | links | | `x & y` |
| `^` | bitweise exklusives ODER | links | nur Konstante | `123 ^ 4` |
| `\|` | bitweise ODER | links | | `x \| y` |
| `&&` | logisches UND | links | nur Konstante | `123 && 4` |
| `\|\|` | logisches ODER | links | nur Konstante | `123 \|\| 4` |

Falls erforderlich, können Klammern verwendet werden, um die Auswertungsreihenfolge zu ändern:

```
x = 2 + 3 * 4;     // weist x den Wert 14 zu
y = (2 + 3) * 4;   // weist y den Wert 20 zu
```

### *Bedingungen*

Bedingungen werden im Allgemeinen durch den Vergleich zweier Ausdrücke gebildet. Es gibt auch zwei konstante Bedingungen, `true` und `false`, die entsprechend immer zu wahr oder falsch bewertet werden. Der Wahrheitswert einer Bedingung kann mit dem Negationsoperator umgekehrt werden. Zwei Bedingungen können mit UND- und ODER-Operator verknüpft werden. Die nachstehende Tabelle faßt die unterschiedlichen Bedingungen zusammen.

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

## Der Präprozessor

Der Präprozessor stellt folgende Anweisungen zur Verfügung:

#include, #define, #ifdef, #ifndef, #if, #elif, #else, #endif, #undef.

Seine Arbeitsweise entspricht ziemlich genau dem Standard-C-Präprozessor, so daß fast alles was von einem C-Präprozessor verarbeitet wird in NQC die gleiche Wirkung hat. Wichtige Abweichungen sind weiter unten aufgeführt.

### *#include*

Die `#include`-Anweisung arbeitet wie erwartet. Zu beachten ist, daß Dateinamen in doppelte Anführungszeichen einzuschließen sind. Es gibt keine Schreibweise für einen include-Pfad für Systemdateien. Deshalb ist die Verwendung von spitzen Klammern bei Dateinamen nicht erlaubt.

```
#include "foo.nqh"      // zulässig
#include <foo.nqh>      // Fehler !
```

### *#define*

Die `#define`-Anweisung wird für einfache Makroersetzungen verwendet. Die wiederholte Definition eines Makros führt zu einem Fehler ( nicht so in C, wo eine Warnung erzeugt wird). Makros werden normalerweise durch das Zeilenende abgeschlossen. Der Schrägstrich nach links (backslash '\' ) ermöglicht jedoch mehrzeilige Makros:

```
#define foo(x)  do { bar(x); \
                     baz(x); } while(false)
```

Die `#undef`-Anweisung macht die Makrodefinition ungültig.

### *Bedingte Übersetzung*

Die bedingte Übersetzung arbeitet wie beim C-Präprozessor. Folgende Präprozessoranweisungen können verwendet werden:

```
#if Bedingung
#ifdef Symbol
#ifndef Symbol
#else
#elif Bedingung
#endif
```

Bedingungen in der `#if`-Anweisung verwenden dieselben Operatoren und Prioritäten wie in C. Der Operator `definded()` wird ebenfalls unterstützt.

### *Programminitialisierung*

Der Übersetzer setzt einen Aufruf für eine besondere Initialisierungsfunktion, `_init`, an den Anfang eines Programmes. Diese Funktion ist Bestandteil des RCX-API und setzt alle drei Ausgange auf volle Leistung in Vorwärtsrichtung (aber noch ausgeschalten). Die Initialisierungsfunktion kann durch die Anweisung `#pragma noinit` unterdrückt werden.

```
#pragma noinit    // keine Programminitialisierung verwenden!
```

Die Initialisierungsfunktion `_init` kann durch die Anweisung `#pragma init` durch eine andere Funktion ersetzt werden.

```
#pragma init function   // eigene Initialisierung benutzen
```

## RCX-API

Das RCX-API legt eine Reihe von Konstanten, Funktionen, Werten und Makros fest, die Zugriff auf Einrichtungen des RCX, wie etwa Sensoren und Ausgänge, erlauben.

### *Sensoren*

Die Namen `SENSOR_1`, `SENSOR_2`, and `SENSOR_3` bezeichnen die Sensoreingänge des RCX. Bevor der Wert eines Sensors gelesen werden kann, muß der Sensor richtig konfiguriert werden. Ein Sensor besitzt zwei verschiedene Einstellungen: den Typ und den Modus. Der Typ legt fest, wie RCX den Sensor elektrisch anspricht, während der Modus festlegt, wie der Wert interpretiert wird. Für einige Sensorentypen ergibt nur ein Modus Sinn, aber andere (z.B. der Temperatursensor) können ebenso gut in verschiedenen Modi gelesen werden (z.B. Fahrenheit oder Celsius). Der Typ und der Modus können mit den Anweisungen

`SetSensorType(`*`Sensor`*`,`*`Typ`*`)` und `SetSensorMode(`*`Sensor`*`,`*`Modus`*`)` eingestellt werden.

```
SetSensorType(SENSOR_1, SENSOR_TYPE_LIGHT);
SetSensorMode(SENSOR_1, SENSOR_MODE_PERCENT);
```

Der Einfachheit halber können Modus und Typ auch gemeinsam mit dem Befehl `SetSensor(`*`Sensor`*`,`*`Konfiguration`*`)` eingestellt werden. Das ist die einfachste und gebräuchlichste Art einen Sensor einzustellen.

```
SetSensor(SENSOR_1, SENSOR_LIGHT);
SetSensor(SENSOR_2, SENSOR_TOUCH);
```

Zulässige Angaben für den Typ, den Modus und die Konfiguration eines Sensors sind nachfolgend angegeben.

| **Sensortyp** | **Bedeutung** |
|---|---|
| SENSOR_TYPE_TOUCH | Berührungssensor |
| SENSOR_TYPE_TEMPERATURE | Temperatursensor |
| SENSOR_TYPE_LIGHT | Lichtsensor |
| SENSOR_TYPE_ROTATION | Rotationssensor |

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

| **Sensorkonfiguration** | **Typ** | **Modus** |
|---|---|---|
| SENSOR_TOUCH | SENSOR_TYPE_TOUCH | SENSOR_MODE_BOOL |
| SENSOR_LIGHT | SENSOR_TYPE_LIGHT | SENSOR_MODE_PERCENT |
| SENSOR_ROTATION | SENSOR_TYPE_ROTATION | SENSOR_MODE_ROTATION |
| SENSOR_CELSIUS | SENSOR_TYPE_TEMPERATURE | SENSOR_MODE_CELSIUS |
| SENSOR_FAHRENHEIT | SENSOR_TYPE_TEMPERATURE | SENSOR_MODE_FAHRENHEIT |
| SENSOR_PULSE | SENSOR_TYPE_TOUCH | SENSOR_MODE_PULSE |
| SENSOR_EDGE | SENSOR_TYPE_TOUCH | SENSEO_MODE_EDGE |

Ein Sensorwert kann gelesen werden, indem man seinen Namen in einer Bedingung verwendet. So prüfen zum Beispiel die folgenden Zeilen, ob der Wert von Sensor 1 größer als 20 ist:

```
if (SENSOR_1 > 20)
      // tue etwas ...
```

Einige Sensortypen (wie etwa `SENSOR_TYPE_ROTATION`) erlauben es, den Wert ihres internen Zählers mit folgenden Befehl zurückzusetzen:

```
ClearSensor(Sensor);
```

## *Ausgänge*

Die Namen `OUT_A`, `OUT_B`, und `OUT_C` bezeichnen die drei Ausgänge des RCX. Alle Befehle zur Steuerung der Ausgänge können gleichzeitig auf mehrere Ausgänge angewendet werden. Um mehrere Ausgänge in einem Befehl anzugeben, werden die Namen der Ausgänge einfach mit einem Pluszeichen verbunden. So kann z.B. `"OUT_A+OUT_B"` dazu verwendet werden, die Ausgänge A und B anzugeben.

Jeder Ausgang hat drei unterschiedliche Merkmale: Modus, Richtung und Leistung. Der Modus kann mit dem Befehl `SetOutput`(*Ausgänge*,*Modus*) eingestellt werden. Die Modusangabe ist eine der folgenden Konstanten:

| **Ausgangsmodus** | **Bedeutung** |
|---|---|
| `OUT_OFF` | Ausgang abgeschaltet (Motor kann nicht drehen) |
| `OUT_ON` | Ausgang eingeschaltet (Motor dreht) |
| `OUT_FLOAT` | Motor im Freilauf |

Die beiden anderen Merkmale, Richtung und Leistung, können jederzeit gesetzt werden. Sie wirken jedoch nur bei eingeschaltetem Motor. Die Richtung wird mit dem Befehl `SetDirection`(*Ausgänge*,*Richtung*) eingestellt. Die Richtungsangabe ist eine der folgenden Konstanten:

| Richtung | Bedeutung |
|---|---|
| OUT_FWD | Vorwärtsrichtung |
| OUT_REV | Rückwärtsrichtung |
| OUT_TOGGLE | Richtungsumkehr |

Die Leistungsangabe reicht von 0 (niedrigste Stufe) bis 7 (höchste Stufe). Die Angaben OUT_LOW, OUT_HALF und OUT_FULL können zur Einstellung der Leistung verwendet werden. Die Leistung wird mit dem Befehl SetPower(*Ausgänge*,*Leistung*) eingestellt.

Bei Programmbeginn werden alle Motoren auf die höchste Leistungsstufe und auf die Vorwärtsrichtung eingestellt (bleiben aber abgeschalten).

Da Befehle zur Steuerung der Ausgänge häufig vorkommen, werden eine Reihe von Funktionen zur Verfügung gestellt, die diese Aufgabe erleichtern. Es ist jedoch anzumerken, daß diese Befehle keine Möglichkeiten bieten, die über die der Befehle SetOutput and SetDirection hinausgehen. Sie stellen lediglich eine abkürzende Schreibweise dar.

| Befehl | Aswirkung |
|---|---|
| On(Ausgänge) | Motor einschalten |
| Off(Ausgänge) | Motor ausschalten |
| Float(Ausgänge) | Freilauf |
| Fwd(Ausgänge) | Vorwärtsrichtung einstellen |
| Rev(Ausgänge) | Rückwärtsrichtung einstellen |
| Toggle(Ausgänge) | Bewegungsrichtung umkehren |
| OnFwd(Ausgänge) | Vorwärtsrichtung einstellen und Motor einschalten |
| OnRev(Ausgänge) | Rückwärtsrichtung einstellen und Motor einschalten |
| OnFor(Ausgänge, Zeit) | Ausgänge für die angegebene Zeitspanne einschalten (in 100-stel Sekunden) |

Hier einige Beispiele, die die Ausgangs-Befehle verwenden:

```
OnFwd(OUT_A);          // A in Vorwärtsrichtung einschalten
OnRev(OUT_B);          // B in Rückwärtsrichtung einschalten
```

```
Toggle(OUT_A + OUT_B);  // Bewegungsrichtung von A und B umkehren
Off(OUT_A + OUT_B);     // A und B abschalten
OnFor(OUT_C, 100);      // C für eine Sekunde einschalten
```

Alle Ausgangsfunktionen verlangen als Argumente Konstanten, mit folgenden Ausnahme:

| | |
|---|---|
| `OnPower` | zur Angabe der Leistungsstufe kann ein Ausdruck verwendet werden |
| `OnFor` | für die Zeitspanne kann ein Ausdruck verwendet werden |

## *Verschiedene Befehle*

`Wait(`*`Zeitspanne`*`)` - versetzt eine Task für eine bestimmte Zeit in einen Wartezustand (in 100-stel Sekunden). Die Zeitspanne kann als Ausdruck oder als Konstante angegeben werden:

```
Wait(100);        // 1 Sekunde warten
Wait(Random(100));// eine zufällige Zeit (bis zu 1 Sekunde) warten
```

`PlaySound(`*`Klang`*`)` - einen von 6 in RCX vorgegebenen Klängen spielen. Das Argument muß eine Konstante sein. Für den Befehl `PlaySound` sind folgende Konstanten vordefiniert:

| | | |
|---|---|---|
| `SOUND_CLICK` | `SOUND_DOUBLE_BEEP` | `SOUND_DOWN` |
| `SOUND_UP` | `SOUND_LOW_BEEP` | `SOUND_FAST_UP.` |

```
PlaySound(SOUND_CLICK);
```

`PlayTone(`*`Frequenz,Dauer`*`)` - Gibt einen Ton der angegebenen Frequenz und Dauer aus. Die *`Frequenz`* wird in Hertz angegeben, die *`Dauer`* in 100-stel Sekunden.

```
PlayTone(440, 50);// eine halbe Sekunde lang den Ton 'A' ausgeben
```

`SelectDisplay(`*`Modus`*`)` - legt die Informationen in der LCD-Anzeige fest. Es gibt 7 verschiedene Einstellmöglichkeiten. RCX verwendet die Einstellung `DISPLAY_WATCH`, wenn keine Angaben gemacht werden.

| **Modus** | **LCD Anzeige** |
|---|---|
| `DISPLAY_WATCH` | Systemuhr anzeigen |
| `DISPLAY_SENSOR_1` | Wert von Sensor 1 anzeigen |
| `DISPLAY_SENSOR_2` | Wert von Sensor 2 anzeigen |
| `DISPLAY_SENSOR_3` | Wert von Sensor 3 anzeigen |
| `DISPLAY_OUT_A` | Einstellungen von Ausgang A anzeigen |
| `DISPLAY_OUT_B` | Einstellungen von Ausgang B anzeigen |
| `DISPLAY_OUT_C` | Einstellungen von Ausgang C anzeigen |

```
SelectDisplay(DISPLAY_SENSOR_1);    // Wert von Sensor 1 anzeigen
```

`ClearMessage()` - Nachrichtenspeicher der RCX-zu-RCX-Kommunikation löschen. Das erlaubt das Auslesen der nächsten empfangenen Infrarot-Nachricht. Mit Hilfe der Funktion `Message()` kann der augenblickliche Inhalt des Nachrichtenspeichers ausgelesen werden.

```
ClearMessage();          // vorhandene Nachricht löschen
until(Message() > 0);    // auf die nächste Nachricht warten
```

`SendMessage(`*`Nachricht`*`)` - Sende eine Infrarot-Nachricht an ein anderes RCX. *`Nachricht`*' kann ein beliebiger Ausdruck sein, das RCX kann jedoch nur Nachrichten mit einem Wert zwischen 0 und 255 senden, d.h. nur die niederwertigsten 8 Bit werden verwendet.

```
SendMessage(  3); // sende Nachricht 3
SendMessage(259); // eine weitere Möglichkeit, Nachricht 3 zu senden
```

`SetWatch(`*`Stunden,Minuten`*`)` - Setze die Systemuhr auf die angegenen Stunden und Minuten. *`Stunden`* muß eine Konstante zwischen 0 und 23 sein. *`Minuten`* muß eine Konstante zwischen 0 und 59 sein.

```
SetWatch(3, 15);  // setze die Systemuhr auf 3:15
```

`ClearTimer(`*`Zähler`*`)` - Einen der vier internen Zähler des RCX zurücksetzen. *`Zähler`* muß eine Konstante zwischen 0 und 3 sein.

```
ClearTimer(0);    // den ersten Zähler zurücksetzen
```

`StopAllTasks()` - Halte alle laufenden Tasks an. Hiermit wird das gesamte Programm angehalten und dadurch wird jeder nachfolgende Befehl wirkungslos.

```
StopAllTasks();   // Programm anhalten
```

`SetTxPower(`*`Leistungsstufe`*`)` - Setze die Leistungsstufe des Infrarot-Senders des RCX. *`Leistungsstufe`* ist entweder `TX_POWER_LO` (gering) oder `TX_POWER_HI` (hoch).

```
SetTxPower(TX_POWER_LO);// setze IR auf geringe Leistung
```

## *Datenspeicher*

Das RCX enthält einen Datenspeicher, der zum Ablegen von Werten aus Sensorlesungen, von Variablen und von der Systemuhr dienen kann. Bevor Daten gespeichert werden können, muß der Datenspeicher zunächst mit dem Befehl `CreateDatalog(`*`Größe`*`)` angelegt werden. Das Argument *`Größe`* muß eine Konstante sein. Sie legt fest, wieviele Werte in diesem Datenspeicher aufgenommen werden können.

```
CreateDatalog(100);     // Datenspeicher für 100 Werte anlegen
```

Mit dem Befehl `AddToDatalog(`*`Wert`*`)` können nun Werte aufgenommen werden. Wenn der Datenspeicherinhalt in den Rechner geladen wird, zeigt er sowohl den Wert als auch die Quelle (Zähler, Variable, usw.) des jeweiligen Wertes an. Der Datenspeicher unterstützt die folgenden Datenquellen: Zähler, Sensorwerte, Variablen und Systemuhr. Andere Datenypen (z.B. Konstanten und Zufallszahlen) können ebenfalls gespeichert werden, aber NQC speichert in diesen Fällen den Wert in einer Variablen zwischen und legt ihn erst anschließend im Datenspeicher

ab. Der Wert wird zwar gewissenhaft im Datenspeicher festgehalten, die Quellenangabe kann aber ein wenig irreführend sein.

```
AddToDatalog(Timer(0)); // Wert von Zähler 0 speichern

AddToDatalog(x);  // Wert der Variablen 'x' speichern
AddToDatalog(7);  // 7 speichern - sieht wie ein Variablenwert aus
```

Das RCX kann selbst keine Werte aus dem Datenspeicher auslesen. Der Datenspeicher muß in den Rechner zurückgeladen werden. Die Besonderheiten dieser Rückspeicherung hängen von der benutzten NQC-Version ab. In der Kommandozeilenversion sorgen die beiden folgenden Befehle dafür, daß der Datenspeicher zurückgeladen und ausgegeben wird:

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

### *Werte*

Das RCX besitzt mehrere unterschiedliche Datenquelle: Sensoren, Variablen, Zähler, usw.. In NQC kann auf diese Quellen mit besonderen Befehlen zugegriffen werden.

Das RCX besitzt vier Zähler, die die Zeit mit einer Auflösung von 1/10 Sekunde (100 Millisekunden) messen. Um einen Zähler auszulesen wird die Anweisung `Timer(`*n*`)` verwendet, wobei *n* eine Konstante zwischen 0 und 3 ist, die die Nummer des zu lesenden Zählers angibt.

Sensoren enthalten mehere unterschiedliche Wert. So kann der Rohwert, der boolsche Wert und der Normalwert eines Sensors gelesen wewrden. Weiterhin ist es in einem Programm möglich, den Sensortyp und den Modus auszulesen. Alle diese Sensorbefehle erfordern ein Argument, welches den anzusprechenden Sensor bezeichnet. Diese Argument liegt zwischen 0 und 2. Es sei angemerkt, daß die Bezeichnungen `SENSOR_1`, `SENSOR_2` und `SENSOR_3` einfach nur Makros für Audrücke der Form `SensorValue(`*n*`)` sind:

```
#define SENSOR_1  SensorValue(0)
#define SENSOR_2  SensorValue(1)
#define SENSOR_3  SensorValue(2)
```

Das RCX-API stellt weiterhin Befehle zur Erzeugung von Zufallszahlen, zum Ablesen der Systemuhr und zum Lesen der letzten erhaltenen Infrarot-Nachricht zur Verfügung. Alle diese Befehle sind nachfolgend zusammengefaßt:

| **Befehl** | **Bedeutung** |
|---|---|
| `Timer(n)` | Wert des Zählers n |
| `Random(n)` | Zufallszahl zwischen 0 und n |
| `Watch()` | Systemzeit in Minuten |
| `Message()` | Wert der letzten erhaltenen Infrarot-Nachricht |
| `SensorValue(n)` | Wert des Sensors n |
| `SensorType(n)` | Typ des Sensors n |
| `SensorMode(n)` | Modus des Sensors n |
| `SensorValueRaw(n)` | Rohwert des Sensors n |
| `SensorValueBool(n)` | boolescher Wert des Sensors n |

# Anhang A - Kurzübersicht

## *Anweisungen*

| Anweisung | Beschreibung |
|---|---|
| `while (Bedingung) Rumpf` | Rumpf nicht ausführen, wenn die Bedingung falsch ist;<br>Rumpf solange ausführen, wie die Bedingung wahr ist |
| `do Rumpf while (Bedingung)` | Rumpf mindestens einmal ausführen und anschließend solange bis die Bedingung falsch ist |
| `until (Bedingung) Rumpf` | Rumpf solange ausführen bis die Bedingung wahr ist |
| `break` | while-, do- oder until-Schleife beenden |
| `continue` | den nächste Durchlauf einer while-, do- oder until-Schleife beginnen |
| `repeat (Ausdruck) Rumpf` | den Rumpf mit der angegebene Zahl an Dürchläufen ausführen |
| `if (Bedingung) Anwsg1`<br>`if (Bedingung)`<br>`  Anwsg1`<br>`else`<br>`  Anwsg2` | *Anwsg1* ausführen, wenn die Bedingung wahr ist, sonst *Anwsg2* ausführen (falls vorhanden) |
| `start task_name` | angegebene Task starten |
| `stop task_name` | angegebene Task anhalten |
| `Funktion(Argumente)` | Funktion mit den angegebenen Argumenten aufrufen |
| `var = Ausdruck` | Ausdruck bewerten und an die Variable var zuweisen |
| `var += Ausdruck` | Ausdruck bewerten und zur Variablen var addieren |
| `var -= Ausdruck` | Ausdruck bewerten und von der Variablen var subtrahieren |
| `var *= Ausdruck` | Ausdruck bewerten und den Wert der Variablen var damit multiplizieren |
| `var /= Ausdruck` | Ausdruck bewerten und den Wert der Variablen var dadurch dividieren |
| `var \|= Ausdruck` | Wert der Variablen var und Wert des Ausdrucks bitweise ODER-Verknüpfen, Ergebnis an var zuweisen |
| `var &= Ausdruck` | Wert der Variablen var und Wert des Ausdrucks bitweise UND-Verknüpfen, Ergebnis an var zuweisen |
| `return` | Funktion beenden und zum Aufrufer zurückkehren |
| `Ausdruck` | Ausdruck auswerten |

## *Bedingungen*

Bedingungen werden in Kontrollstrukturen verwendet um Entscheidungen zu treffen. In den meisten Fällen besteht die Bedingung aus einem Vergleich zweier Ausdrücke.

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

## *Ausdrücke*

Es gibt mehrere verschiedene Werte die in Ausdrücken verwendet werden können, unter anderem Konstanten, Variablen und Sensorwerte. Zu beachten ist, daß die Namen `SENSOR_1`, `SENSOR_2` und `SENSOR_3` Makros sind die zu `SensorValue(0)`, `SensorValue(1)` and `SensorValue(2)` expandiert werden.

| **Befehl** | **Bedeutung** |
|---|---|
| *`Zahl`* | ein konstanter Wert (z.B. 123) |
| *`Variable`* | eine benannte Variable (z.B. x) |
| `Timer(n)` | Wert des Zählers n (n zwischen 0 und 3) |

| | |
|---|---|
| `Random(n)` | Zufallszahl zwischen 0 und n |
| `Watch()` | Systemzeit in Minuten |
| `Message()` | Wert der letzten erhaltenen Infrarot-Nachricht |
| `SensorValue(n)` | augenblicklicher Wert des Sensors n |

Werte können durch Operatoren verknüpft werden. Einige der Operatoren dürfen nur in konstanter Ausdrücke verwendet werden, d.h. ihre Operanden müssen entweder Konstanten sein oder sie müssen Ausdrücke sein, die nur aus Konstanten bestehen. Die Operatoren sind nachstehend mit fallender Priorität aufgeführt.

| Operator | Beschreibung | Bindung | Einschränkung | Beispiel |
|---|---|---|---|---|
| `abs()`<br>`sign()` | Absolutwert<br>Vorzeichen des Operanden | nicht vorhanden<br>nicht vorhanden | | `abs(x)`<br>`sign(x)` |
| `++`<br>`--` | Inkrement<br>Dekrement | links<br>links | nur Variablen<br>nur Variablen | `x++ or ++x`<br>`x-- or --x` |
| `-`<br>`~` | unäres Minus (Vorzeichen)<br>Bitweise Negation (unär) | rechts<br>rechts | <br>nur Konstante | `-x`<br>`~123` |
| `*`<br>`/`<br>`%` | Multiplikation<br>Division<br>Modulo | links<br>links<br>links | <br><br>nur Konstante | `x * y`<br>`x / y`<br>`123 % 4` |
| `+`<br>`-` | Addition<br>Subtraktion | links<br>links | | `x + y`<br>`x - y` |
| `<<`<br>`>>` | Linksshift-Operator<br>Rechtsshift-Operator | links<br>links | nur Konstante<br>nur Konstante | `123 << 4`<br>`123 >> 4` |
| `&` | bitweise UND | links | | `x & y` |
| `^` | bitweise exklusives ODER | links | nur Konstante | `123 ^ 4` |
| `\|` | bitweise ODER | links | | `x \| y` |
| `&&` | logisches UND | links | nur Konstante | `123 && 4` |
| `\|\|` | logisches ODER | links | nur Konstante | `123 \|\| 4` |

## *RCX -Funktionen*

Die meisten dieser Funktionen verlangen, daß alle Argumente konstante Ausdrücke sind (Zahlen oder Verknüpfungen andrerer Konstanter Ausdrücke). Ausnahmen bilden die Funktionen, die Sensoren als Argumente haben, und jene, die beliebige Argumente haben können. Im Fall der Sensoren sollten die Argumente Sensornamen sein: `SENSOR_1`, `SENSOR_2` oder `SENSOR_3`. In manchen Fällen gibt es vordefinierte Namen (z.B. `SENSOR_TOUCH`) für entsprechende Konstanten.

| **Funktion** | **Beschreibung** | **Beispiel** |
|---|---|---|
| `SetSensor(`*`Sensor, Konfig`*`)` | Sensor konfigurieren | `SetSensor(SENSOR_1, SENSOR_TOUCH)` |
| `SetSensorMode(`*`Sensor, Modus`*`)` | Sensormodus einstellen | `SetSensor(SENSOR_2, SENSOR_MODE_PERCENT)` |
| `SetSensorType(`*`Sensor, Typ`*`)` | Sensortyp einstellen | `SetSensor(SENSOR_2, SENSOR_TYPE_LIGHT)` |
| `ClearSensor(`*`Sensor`*`)` | Sensor'wert löschen | `ClearSensor(SENSOR_3)` |
| `On(`*`Ausgänge`*`)` | *`Ausgänge`* einschalten | `On(OUT_A + OUT_B)` |
| `Off(`*`Ausgänge`*`)` | *`Ausgänge`* ausschalten | `Off(OUT_C)` |
| `Float(`*`Ausgänge`*`)` | Freilauf einschalten | `Float(OUT_B)` |
| `Fwd(`*`Ausgänge`*`)` | *`Ausgänge`* auf Vorwärtsrichtung setzen | `Fwd(OUT_A)` |
| `Rev(`*`Ausgänge`*`)` | *`Ausgänge`* auf Rückwärtsrichtung setzen | `Rev(OUT_B)` |
| `Toggle(`*`Ausgänge`*`)` | Bewegungsrichtung der *`Ausgänge`* umschalten | `Toggle(OUT_C)` |
| `OnFwd(`*`Ausgänge`*`)` | *`Ausgänge`* in Vorwärtsrichtung einschalten | `OnFwd(OUT_A)` |
| `OnRev(`*`Ausgänge`*`)` | *`Ausgänge`* in Rückwärtsrichtung einschalten | `OnRev(OUT_B)` |
| `OnFor(`*`Ausgänge, Zeitspanne`*`)` | *`Ausgänge`* für eine bestimmte Zeitspanne einschalten (Angabe in Schritten zu 1/100 Sekunden *`Zeitspanne`* kann als Ausdruck angegeben werden | `OnFor(OUT_A, x)` |

| | | |
|---|---|---|
| `SetOutput(`*`Ausgänge, Modus`*`)` | Ausgangsmodus setzen | `SetOutput(OUT_A, OUT_ON)` |
| `SetDirection(`*`Ausgänge, Rcht`*`)` | Ausgangsrichtung setzen | `SetDirection(OUT_A, OUT_FWD)` |
| `SetPower(`*`Ausgänge, Leistung`*`)` | Ausgangsleistung (0-7) setzen.<br>*`Leistung`* kann als Ausdruck angegeben werden | `SetPower(OUT_A, x)` |
| `Wait(`*`Zeitspanne`*`)` | eine bestimmte Zeitspanne warten<br>Angabe in 1/100 Sekunden<br>*`Zeitspanne`* kann als Ausdruck angegeben werden | `Wait(x)` |
| `PlaySound(`*`Klang`*`)` | angegeben Klang (0-5) ausgeben | `PlaySound(SOUND_CLICK)` |
| `PlayTone(`*`Frequenz`*`, Dauer)` | Ton in der angegebene Frequenz und Dauer ( 1/10 Sekunden) ausgeben | `PlayTone(440, 5)` |
| `ClearTimer(`*`Zähler`*`)` | Zähler (0-3) auf den Wert 0 setzen | `ClearTimer(0)` |
| `StopAllTasks()` | alle laufenden Tasks anhalten | `StopAllTasks()` |
| `SelectDisplay(`*`Modus`*`)` | einen von 7 Anzeigemodi auswählen:<br>0: Systemuhr<br>1-3: Sensorwert,<br>4-6: Ausgangseinstellung<br>*`Modus`* kann ein Ausdruck sein | `SelectDisplay(1)` |
| `SendMessage(`*`Nachricht`*`)` | eine Infrarot-Nachricht (1-255) senden<br>*`Nachricht`* kann ein Ausdruck sein | `SendMessage(x)` |
| `ClearMessage()` | Infrarot-Nachrichten--speicher löschen | `ClearMessage()` |

| | | |
|---|---|---|
| `CreateDatalog(`*`Größe`*`)` | neuen Datenspeicher der angegebenen Größe anlegen | `CreateDatalog(100)` |
| `AddToDatalog(`*`Wert`*`)` | *`Wert`* im Datenspeicher ablegen<br>*`Wert`* kann ein Ausdruck sein | `AddToDatalog(Timer(0))` |
| `SetWatch(`*`Stunden, Minuten`*`)` | Systemuhr einstellen | `SetWatch(1,30)` |
| `SetTxPower(`*`hi_lo`*`)` | Sendeleistung des Infrarotsenders hoch oder niedrig einstellen | `SetTxPower(TX_POWER_LO)` |

## *RCX -Konstanten*

Für viele der Übergabewerte an RCX-Funktionen haben gibt es benannte Konstaten, die dazu beitragen können ein Programm lesbarer zu machen. Wenn möglich, sind benannte Konstanten der Angabe von Zahlenwerten vorzuziehen.

| | |
|---|---|
| **Konfigurationswerte für SetSensor()** | `SENSOR_TOUCH, SENSOR_LIGHT, SENSOR_ROTATION, SENSOR_CELSIUS, SENSOR_FAHRENHEIT, SENSOR_PULSE, SENSOR_EDGE` |
| **Modi für SetSensorMode()** | `SENSOR_MODE_RAW, SENSOR_MODE_BOOL, SENSOR_MODE_EDGE, SENSOR_MODE_PULSE, SENSOR_MODE_PERCENT, SENSOR_MODE_CELSIUS, SENSOR_MODE_FAHRENHEIT, SENSOR_MODE_ROTATION` |
| **Typen für SetSensorType()** | `SENSOR_TYPE_TOUCH, SENSOR_TYPE_TEMPERATURE, SENSOR_TYPE_LIGHT, SENSOR_TYPE_ROTATION` |
| **Ausgänge für On(), Off() usw.** | `OUT_A, OUT_B, OUT_C` |
| **Modi für SetOutput()** | `OUT_ON, OUT_OFF, OUT_FLOAT` |
| **Richtungen für SetDirection()** | `OUT_FWD, OUT_REV, OUT_TOGGLE` |
| **Leistungsstufen für SetPower()** | `OUT_LOW, OUT_HALF, OUT_FULL` |
| **Klänge für PlaySound()** | `SOUND_CLICK, SOUND_DOUBLE_BEEP, SOUND_DOWN, SOUND_UP, SOUND_LOW_BEEP, SOUND_FAST_UP` |
| **Modi für SelectDisplay()** | `DISPLAY_WATCH, DISPLAY_SENSOR_1, DISPLAY_SENSOR_2, DISPLAY_SENSOR_3, DISPLAY_OUT_A, DISPLAY_OUT_B, DISPLAY_OUT_C` |
| **Tx-Leistungsstufen für SetTxPower()** | `TX_POWER_LO, TX_POWER_HI` |

### *Schlüsselwörter*

Schlüsselwörter sind die Bezeichnungen, die der NQC-Übersetzter für die Sprache reserviert. Es ist nicht erlaubt, irgendeinen dieser Namen zur Bezeichnung von Funktionen, Tasks oder Variablen zu verwenden.

| | | |
|---|---|---|
| **`__sensor`** | **`false`** | **`stop`** |
| **`abs`** | **`if`** | **`sub`** |
| **`asm`** | **`inline`** | **`task`** |
| **`break`** | **`int`** | **`true`** |
| **`const`** | **`repeat`** | **`void`** |
| **`continue`** | **`return`** | **`while`** |
| **`do`** | **`sign`** | |
| **`else`** | **`start`** | |

## Anhang B - Datei *rcx.nqh*

Die Datei rcx.nqc enthält interne Informationen für den NQC-Übersetzer, die die RCX-API definieren. Der volllständige Ausdruck der Datei rcx.nqc ist zum Nachlesen unten wiedergegeben. Beachten Sie bitte, daß wegen des Zeilenunbruchs in diesen Handbuch die Datei rcx.nqh (im NQC-Paket enthalten) auch direkt eingesehen werden sollte.

```
/*
 * rcx.nqh - version 2.0
 * Copyright (C) 1998,1999 Dave Baum
 *
 * CyberMaster definitions by Laurentino Martins
 *
 * This file is part of nqc, the Not Quite C compiler for the RCX
 *
 * The contents of this file are subject to the Mozilla Public License
 * Version 1.0 (the "License"); you may not use this file except in
 * compliance with the License. You may obtain a copy of the License at
 * http://www.mozilla.org/MPL/
 *
 * Software distributed under the License is distributed on an "AS IS"
 * basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the
 * License for the specific language governing rights and limitations
 * under the License.
 *
 * The Initial Developer of this code is David Baum.
 * Portions created by David Baum are Copyright (C) 1998 David Baum.
 * All Rights Reserved.
 */

/*
 *  This file defines various system constants and macros to be used
 *  with the RCX.  Most of the real functionality of the RCX is
 *  defined here rather than within nqcc itself.
 *
 */


// these are the standard system definitions for 2.0

/**************************
 * sensors
 **************************/

// constants for selecting sensors
```

```
#define SENSOR_1    SensorValue(0)
#define SENSOR_2    SensorValue(1)
#define SENSOR_3    SensorValue(2)

#ifdef __CM
// alternative names for input sensors
#define SENSOR_L SENSOR_1 // Left sensor
#define SENSOR_M SENSOR_2 // Middle sensor
#define SENSOR_R SENSOR_3 // Right sensor
#endif

// modes for SetSensorMode()
#define SENSOR_MODE_RAW                 0x00
#define SENSOR_MODE_BOOL         0x20
#define SENSOR_MODE_EDGE         0x40
#define SENSOR_MODE_PULSE        0x60
#define SENSOR_MODE_PERCENT             0x80
#ifdef __RCX
#define SENSOR_MODE_CELSIUS             0xa0
#define SENSOR_MODE_FAHRENHEIT   0xc0
#define SENSOR_MODE_ROTATION     0xe0
#endif


void SetSensorMode(__sensor sensor, const int mode)  { asm { 0x42, &sensor :
0x03000200, mode }; }

#ifdef __RCX
// types for SetSensorType()
#define SENSOR_TYPE_TOUCH        1
#define SENSOR_TYPE_TEMPERATURE  2
#define SENSOR_TYPE_LIGHT        3
#define SENSOR_TYPE_ROTATION            4

// type/mode combinations for SetSensor()
#define _SENSOR_CFG(type,mode)    (((type)<<8) + (mode))
#define SENSOR_TOUCH             _SENSOR_CFG(SENSOR_TYPE_TOUCH,
SENSOR_MODE_BOOL)
#define SENSOR_LIGHT             _SENSOR_CFG(SENSOR_TYPE_LIGHT,
SENSOR_MODE_PERCENT)
#define SENSOR_ROTATION    _SENSOR_CFG(SENSOR_TYPE_ROTATION,
SENSOR_MODE_ROTATION)
#define SENSOR_CELSIUS     _SENSOR_CFG(SENSOR_TYPE_TEMPERATURE,
SENSOR_MODE_CELSIUS)
#define SENSOR_FAHRENHEIT  _SENSOR_CFG(SENSOR_TYPE_TEMPERATURE,
SENSOR_MODE_FAHRENHEIT)
#define      SENSOR_PULSE        _SENSOR_CFG(SENSOR_TYPE_TOUCH,
SENSOR_MODE_PULSE)
#define SENSOR_EDGE              _SENSOR_CFG(SENSOR_TYPE_TOUCH,
SENSOR_MODE_EDGE)

// set a sensor's type
```

```
void SetSensorType(__sensor sensor, const int type)    { asm { 0x32, &sensor :
0x03000200, (type) }; }


// set a sensor's type and mode using a config - e.g. SetSensor(SENSOR_1,
SENSOR_LIGHT);
void SetSensor(__sensor sensor, const int tm)  { SetSensorType(sensor, tm>>8);
SetSensorMode(sensor, tm); }


#endif




/**************************
 * ouput
 **************************/


// constants for selecting outputs
#define OUT_A (1 << 0)
#define OUT_B (1 << 1)
#define OUT_C (1 << 2)

// output modes
#define OUT_FLOAT   0
#define OUT_OFF            0x40
#define OUT_ON             0x80

// output directions
#define OUT_REV            0
#define OUT_TOGGLE  0x40
#define OUT_FWD            0x80

// output power levels
#define OUT_LOW            0
#define OUT_HALF    3
#define OUT_FULL    7

// output functions
void SetOutput(const int o, const int m)       { asm { 0x21, (o) + (m) }; }
void SetDirection(const int o, const int d)    { asm { 0xe1, (o) + (d) }; }
void SetPower(const int o, const int &p)       { asm { 0x13, (o), &p :
0x1000015}; }


void On(const int o)       { SetOutput(o, OUT_ON); }
void Off(const int o)      { SetOutput(o, OUT_OFF); }
void Float(const int o) { SetOutput(o, OUT_FLOAT); }
void Toggle(const int o) { SetDirection(o, OUT_TOGGLE); }
void Fwd(const int o)      { SetDirection(o, OUT_FWD); }
void Rev(const int o)      { SetDirection(o, OUT_REV); }
void OnFwd(const int o)    { Fwd(o); On(o); }
void OnRev(const int o)    { Rev(o); On(o); }
```

```
void OnFor(const int o, const int &t)   { On(o); Wait(t); Off(o); }

// CyberMaster specific stuff
#ifdef __CM
// alternate names for motors
#define OUT_L OUT_A // Left motor
#define OUT_R OUT_B // Right motor
#define OUT_X OUT_C // External motor

#define Drive(m0, m1)                     asm { 0x41, DIRSPEED(m0) |
DIRSPEED(m1)<<4 }
#define OnWait(m, n, t)                   asm { 0xc2, (m)<<4 | DIRSPEED(n), t }
#define OnWaitDifferent(m, n0, n1, n2, t) asm { 0x53, (m)<<4 | DIRSPEED(n0),
DIRSPEED(n1)<<4 | DIRSPEED(n2), t }

// Aux. function: Transforms a number between -7 and 7 to a 4 bit sequence:
// Bits: 1..3 - Speed: 0 to 7
// Bit : 4    - Direction: 1 if v>=0, 0 if v<0
#define DIRSPEED(v)        ((v)&8^8|((v)*(((v)>>3)*2^1))&7)


#endif



/*************************
 * data sources
 *************************/

#define Timer(n)                 @(0x10000 + (n))
#define Random(n)                @(0x40000 + (n))
#define SensorValue(n)           @(0x90000 + (n))
#define SensorType(n)            @(0xa0000 + (n))
#define SensorMode(n)            @(0xb0000 + (n))

#ifdef __RCX
// RCX specific data sources
#define Program()                @(0x8)
#define SensorValueRaw(n)  @(0xc0000 + (n))
#define SensorValueBool(n) @(0xd0000 + (n))
#define Watch()                          @(0xe0000)
#define Message()                @(0xf0000)
#endif

#ifdef __CM
// CM specific data sources
#define TachoCount(n)          @(0x50000 + (n)-1) // Use OUT_x as parameter
#define TachoSpeed(n)          @(0x60000 + (n)-1) // Use OUT_x as parameter
#define ExternalMotorRunning() @(0x70002)         // Refered in the SDK as
MotorCurrent(2). Non zero if external motor running.
```

```
#define AGC()                   @(0x100000)        // Automatic Gain Control
#endif


/*************************
 * miscellaneous
 *************************/

// wait for a condition to become true
#define until(c)           while(!(c))

// playing sounds and notes
void PlaySound(const int x)                                  {      asm { 0x51, x
}; }
void PlayTone(const int f, const int d)        {      asm { 0x23, (f), (f)>>8,
(d) }; }

// sounds - for PlaySound()
#define SOUND_CLICK              0
#define SOUND_DOUBLE_BEEP  1
#define SOUND_DOWN               2
#define SOUND_UP                 3
#define SOUND_LOW_BEEP           4
#define SOUND_FAST_UP            5

// sleep for v ticks (10ms per tick)
void Wait(const int &v)    { asm { 0x43, &v : 0x0015}; }


void ClearTimer(const int n) {    asm { 0xa1, n }; }
#define ClearSensor(sensor)       asm    { 0xd1, &sensor : 0x03000200 }

void StopAllTasks() { asm { 0x50 }; }

#ifdef __RCX
// set the display mode
void SelectDisplay(const int &v) { asm { 0x33, &v : 0x0005}; }

// display modes - for SelectDisplay
#define DISPLAY_WATCH            0
#define DISPLAY_SENSOR_1   1
#define DISPLAY_SENSOR_2   2
#define DISPLAY_SENSOR_3   3
#define DISPLAY_OUT_A            4
#define DISPLAY_OUT_B            5
#define DISPLAY_OUT_C            6

// IR message support
void SendMessage(const int &v)    { asm { 0xb2, &v : 0x1000005 }; }
void ClearMessage()                     { asm { 0x90 }; }
```

```
// Data logging
void CreateDatalog(const int size) { asm { 0x52, (size), (size)>>8 }; }
void AddToDatalog(const int &v)   { asm { 0x62, &v : 0x1004203}; }
void UploadDatalog(const int s, const int n) { asm { 0xa4, (s), (s)>>8, (n),
(n)>>8 }; }

// set the system clock
void SetWatch(const int h, const int m)        { asm { 0x22, h, m }; }

// support for controlling the IRMode
#define TX_POWER_LO 0
#define TX_POWER_HI 1
void SetTxPower(const int p)      { asm { 0x31, p }; }
#endif

#ifdef __CM
#define ClearTachoCounter(m) asm { 0x11, (m) }
#endif

// initialization function
void _init()
{
       SetPower(OUT_A + OUT_B + OUT_C, OUT_FULL);
       Fwd(OUT_A + OUT_B + OUT_C);
}
```