NQC-Programmieranleitung

Dave Baum

Version 2.5a4

Titel des englischen Originals:

NQC Programmer's Guide

Inhaltsverzeichnis

Übersetzung: Dr.-Ing. Fritz Mehner Email: mehner@fh-swf.de
Stand: 18. September 2003 Web: lug.fh-swf.de/lego/

Dieses Dokument wurde mit LaTeX 2ε unter LINUX erzeugt.

1

Einleitung

NQC steht für *Not Quite C* („ nicht ganz C “) und ist eine einfache Programmiersprache für mehrere LEGO-MINDSTORMS-Produkte. Einige NQC-Eigenschaften sind von dem MINDSTORMS-Produkt abhängig, welches Sie benutzen. Dieses Produkt wird als Zielgerät für NQC bezeichnet. Zur Zeit unterstützt NQC fünf Zielgeräte: RCX, RCX2 (ein RCX mit der Firmware-Version 2.0), CyberMaster, Scout und Spybotics.

Alle Zielgeräte besitzen eine Bytecode-Interpreter (von LEGO) der dazu verwendet wird die Programme auszuführen. Der NQC-Compiler übersetzt ein Quellprogramm in LEGO-Bytecode, der dann auf dem Zielgerät ausgeführt wird. Obwohl der Präprozessor und die Kontrollstrukturen denen der Programmiersprache C sehr ähnlich sind, ist NQC keine Allzweckprogrammiersprache - es gibt viele Einschränkungen, die von Beschränkungen des LEGO-Bytecode-Interpreters herrühren.

Logisch zerfällt NQC in zwei verschiedene Teile. Die NQC-Sprache legt die Syntax fest, die beim Schreiben von Programmen verwendet werden muß. Die NQC-API (Application Programming Interface (A.d.Ü.)) beschreibt Systemaufrufe, Konstanten und Makros die in Programmen verwendet werden können. Diese API ist in einer besonderen Datei enthalten die in den Compiler eingebaut ist. Diese Datei wird normalerweise vor der Übersetzung berücksichtigt (der Präprozessor (A.d.Ü.)).

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 NQC, wie z.B. das *NQC User Manual*.

Die neuesten Informationen und die gültige Dokumentation für NQC finden Sie auf der Web-Seite http://www.baumfamily.org/nqc

2

Die NQC-Sprache

Dieses Kapitel beschreibt die Sprache NQC. Dies umfaßt die lexikalischen Regeln des Übersetzers, die Programmstruktur, die Anweisungen, die Ausdrücke und die Arbeitsweise des Präprozessors.

2.1 Lexikalische Regeln

Die lexikalischen Regeln beschreiben, wie NQC den Programmtext in einzelne Bestandteile zerlegt. Dies schließt die Art und Weise ein, in der Kommentare geschrieben werden, sowie die Behandlung von Leerzeichen und die Festlegung der gültigen Zeichen für Namen.

2.1.1 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.

2.1.2 Leerzeichen

Leerzeichen (eigentliche Leerzeichen (engl. *space*), Tabulatoren und Zeilenvorschübe) werden dazu verwendet, die Programm- und Textbestandteile voneinander zu trennen 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
```

2.1.3 Numerische Konstanten

Numerische Konstanten können dezimal oder hexadezimal geschrieben werden. Dezimalkonstanten bestehen aus einer oder mehreren Dezimalziffern. 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)
```

2.1.4 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:

`__event_src`	`__res`	`__taskid`	`abs`
`__nolist`	`__sensor`	`__type`	`acquire`
`asm`	`do`	`int`	`sub`
`break`	`else`	`monitor`	`switch`
`case`	`false`	`repeat`	`task`
`catch`	`for`	`return`	`true`
`const`	`goto`	`sign`	`void`
`continue`	`if`	`start`	`while`
`default`	`inline`	`stop`	

2.2 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.

2.2.1 Tasks

Da RCX Multitasking unterstützt, 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 aus mindestens einer Task, `"main"` genannt, die aufgerufen wird, wenn das Programm gestartet wird. Die höchstens zulässige Anzahl von Tasks hängt vom Zielgerät ab - RCX unterstützt 10 Tasks, CyberMaster 4 und der Scout 6.

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

2.2.2 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 werden als 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 ihres 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 in der aufrufenden Umgebung 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 deren 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; der 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 in das Programm eingesetzt wurde. Inline-Funktionen können bei unüberlegter Verwendung zu einem großen Code-Umfang führen.

2.2.3 Unterprogramme

Im Gegensatz zu inline-Funktionen ist der Code eines Unterprogramms nur einmal vorhanden und kann von mehreren Aufrufern verwendet werden. Dadurch nutzen Unterprogramme den Speicher wesentlich besser aus als inline-Funktionen. Wegen einiger Beschränkungen im LEGO-Bytecode-Interpreter haben Unterprogramm allerdings deutliche Einschränkungen. Erstens können Unterprogramme keine Argumente verwenden. Zweitens kann ein Unterprogramm nicht ein anderes Unterprogramm aufrufen. Schließlich ist die Zahl der Unterprogramme beim RCX auf 8, beim CyberMaster auf 4, beim Scout auf 3 und bein Spybotics auf 32 beschränkt. Beim RCX 1.0 und beim CyberMaster darf ein Unterprogramm weiterhin keine lokalen Variablen besitzen oder Berechnungen ausführen die temporären Speicherplatz benötigen, wenn es von mehreren Tasks aufgerufen wird . Wegen dieser einschneidenden Einschränkungen ist die Verwendung von Unterprogrammen weniger wünschenswert als die von Funktionen. 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 der Unterprogramms
}
```

2.2.4 Variablen

Alle Variablen im RCX besitzen denselben Datentyp, nämlich den einer 16-Bit vorzeichenbehafteten ganzen Zahl (16 bit signed integer). 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 zu 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 mehr 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.

Der RCX (und anderen Geräte) stellt eine Anzahl von Speicherplätzen für Variablen in NQC-Programmen zur Verfügung. Es gibt zwei Arten von Speicherplätzen - globale und lokale. Beim Übersetzen weist NQC jeder Variablen einen eigenen Speicherplatz zu. In der Regel können Programmierer die Einzelheiten dieser Speicherplatzzuweisung außer Acht lassen, wenn die beiden folgenden Regeln beachtet werden:

- Wenn eine Variable überall zugreifbar sein soll, dann muß sie global vereinbart werden
- Wenn eine Variable nicht global sein muß, dann sollte sie so lokal wie möglich vereinbart werden. Das gibt dem Übersetzer die größte Freiheit bei der Speicherplatzzuweisung.

Die Anzahl der verfügbaren globalen und lokalen Speicherplätzen hängt vom Gerät ab:

Gerät	**global**	**lokal**
RCX	32	0
CyberMaster	32	0
Scout	10	8
RCX2	32	16
Spybotics	32	4

2.2.5 Felder

RCX2 und Spybotics unterstützen Felder (die Firmware der anderen Geräte bietet nicht genügend Unterstützung für Felder). Felder werden wie gewöhnliche Variablen vereinbart, wobei die Feldlänge in eckigen Klammern anzugeben ist. Die Feldlänge muß eine Konstante sein:

```
int my_array[3];              // Feld mit 3 Elementen vereinbaren
```

Die Elemente eines Feldes werden über ihre Position im Feld angesprochen (den sog. Index). Das erste Element hat den Index 0, das zweite den Index 1 usw.. Hier ein Beispiel:

```
my_array[0] = 123;            // dem ersten Element den Wert 123 zuweisen
my_array[1] = my_array[2];    // 3. Element in das 2. Element kopieren
```

Zur Zeit bestehen einige Einschränkungen bei der Verwendung von Feldern. Diese Beschränkungen werden wahrscheinlich bei zukünftigen NQC-Versionen wegfallen:

- Ein Feld kann nicht als Funktionsargument verwendet werden. Ein einzelnes Element kann jedoch an eine Funktion übergeben werden.
- Weder auf ein gesamtes Feld noch auf ein einzelnes Element kann der Inkrementoperator (++) oder der Dekrementoperator (--) angewendet werden.

- Für Feldelemente sind nur einfache Zuweisungen (`=`) erlaubt. Die kombinierten Zuweisungen (z.B. `+=`) sind nicht erlaubt).
- Anfangswertzuweisungen sind für Felder nicht erlaubt. Die Werte müssen ausdrücklich im Programm zugewiesen werden.

2.3 Anweisungen

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

2.3.1 Variablenvereinbarung

Eine Variablenvereinbarungen, wie sie im vorangehenden Abschnitt beschrieben wurde, ist eine mögliche Art von Anweisung. Sie vereinbart eine lokale Variable (mit optionalem Anfangswertzuweisung) 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*]

Felder können ebenfalls vereinbart werden (nur RCX2):

`int` *array* `[` *Länge* `];`

2.3.2 Zuweisungen

An Variablen die bereits vereinbart sind können die Werte von Ausdrücken zugewiesen werden:

Variable Zuweisungsoperator Ausdruck`;`

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

<table>
<tr><th>Operator</th><th>Wirkung</th></tr>
<tr><td>=</td><td>weist den Wert des Ausdrucks zu</td></tr>
<tr><td>+=</td><td>addiert den Wert des Ausdrucks zum Wert der Variablen</td></tr>
<tr><td>-=</td><td>subtrahiert den Wert des Ausdrucks vom Wert der Variablen</td></tr>
<tr><td>*=</td><td>multipliziert den Wert der Variablen mit dem Wert des Ausdrucks</td></tr>
<tr><td>/=</td><td>dividiert den Wert der Variablen durch den Wert des Ausdrucks</td></tr>
<tr><td>%=</td><td>dividiert den Wert der Variablen durch den Wert des Ausdrucks
und weist den Divisionsrest zu</td></tr>
<tr><td>&=</td><td>bitweise UND-Verknüpfung des Variablenwertes mit dem Ausdruck</td></tr>
<tr><td>|=</td><td>bitweise ODER-Verknüpfung des Variablenwertes mit dem Ausdruck</td></tr>
<tr><td>^=</td><td>bitweise exklusives ODER-Verknüpfung des Variablenwertes mit dem Ausdruck</td></tr>
<tr><td>||=</td><td>weist der Variablen den Absolutwert des Ausdrucks zu</td></tr>
<tr><td>+-=</td><td>weist der Variablen das Vorzeichen (-1,+1,0) des Ausdrucks zu</td></tr>
<tr><td>«=</td><td>Wert um einen feste Stellenzahl Linksschieben</td></tr>
<tr><td>»=</td><td>Wert um einen feste Stellenzahl Rechtsschieben</td></tr>
</table>

Einige Beispiele:

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

2.3.3 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, und 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 (` *Bedingung* `)` *Rumpf*

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).

Eine weiter Schleife ist die `for`-Schleife:

`for (` *Anweisung-1* `;` *Bedingung* `;` *Anweisung-2* `)` *Rumpf*

Eine `for`-Schleife führt immer zunächst die *Anweisung-1* aus. Dann wird wiederholend die *Bedingung* ausgewertet und falls diese wahr ist, der Rumpf ausgeführt. Danach folgt jeweils die Ausführung von *Anweisung-2*. Die `for`-Schleife entspricht der folgenden Anweisungsfolge:

```
Anweisung-1;
while ( Bedingung )
{
Rumpf
Anweisung-2;
}
```

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.

Eine `switch`-Anweisung kann dazu verwendet werden, abhängig von einem Auswahlausdruck einen von mehreren Anweisungsblöcken auszuführen. Vor jedem Block stehen eine oder mehrere `case`-Marken. Jede Fallbezeichnung muß konstant sein und darf innerhalb der `switch`-Anweisung nur einmal vorkommen. In der `switch`-Anweisung wird zunächst der Auswahlausdruck ausgewertet und dann wird der zutreffende Fall angesprungen. Es werden dann jeweils alle Anweisungen des jeweiligen Falles ausgeführt bis entweder eine `break`-Anweisung oder das Ende der `switch`-Anweisung erreicht wird. Höchstens eine `default`-Marke darf ebenfalls verwendet werden - dieser Fall trifft für alle Auswahlwerte zu, für die keine Fallmarke vorhanden ist. Die `switch`-Anweisung hat den folgenden Aufbau:

`switch (` *Auswahlausdruck* `)` *Rumpf*

Die `case`- und `default`-Marken sind keine selbständigen Anweisungen - sie sind Marken die vor Anweisungen stehen. Mehrere Marken können vor derselben Anweisung stehen. Diese Marken haben die folgende Syntax:

`case` *konstanter_Ausdruck* `:`
`default :`

Eine typische `switch`-Anweisung könnte folgendermaßen aussehen:

```
switch ( x )
{
  case 1:
         // tue etwas, wenn x den Wert 1 hat
         break;
  case 2:
  case 3:
         // tue etwas, wenn x den Wert 2 oder 3 hat
         break;
  default:
         // tue etwas, wenn x nicht den Wert 1,2 oder 3 hat
         break;
}
```

Die `goto`-Anweisung erzwingt eine Sprung zu einer bestimmten Stelle im Programm. Anweisungen in einem Programm können mit einem Bezeichner gefolgt von einem Doppelpunkt markiert werden. Eine `goto`-Anweisung enthält dann eine Marke zu der das Programm springen soll. Das folgende Beispiel zeigt, wie eine Endlosschleife, die eine Variable inkrementiert, mittels `goto` geschrieben werden kann:

```
my_loop:
    x++;
    goto my_loop;
```

Die `goto`-Anweisung sollte selten und mit Bedacht verwendet werden. In nahezu allen Fällen ergeben Kontrollstrukturen, wie etwa `if`, `while` und `switch`, ein besser lesbares und besser wartbares Programm als die Verwendung von `goto`. Es muß darauf geachtet werden, niemals ein `goto` zu verwenden um in eine oder aus einer `monitor`- oder `acquire`-Anweisung zu springen. Eine `monitor`-

und eine `acquire`-Anweisung enthalten besonderen Programmcode, der normalerweise beim Ein- und Austritt ausgeführt wird. Ein `goto` würde diesen Code überspringen und vermutlich zu unerwünschtem Verhalten führen.

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
```

2.3.4 Zugriffskontrolle und Ereignisse

Scout, RCX2 und Spybotics unterstützen Zugriffskontrolle und Ereignisüberwachung. Zugriffskontrolle erlaubt einer Task ein oder mehrere Betriebsmittel (engl.: *resources*) in Besitz zu nehmen. In NQC wird die Zugriffskontrolle über die `acquire`-Anweisung durchgeführt. Diese hat zwei Formen:

`acquire (` *Betriebsmittel* `)` *Rumpf*

`acquire (` *Betriebsmittel* `)` *Rumpf* `catch` *Botschafteninterpreter*

Betriebsmittel ist eine Konstante die das angeforderte Betriebsmittel bezeichnet, *Rumpf* und *Botschafteninterpreter* (engl.: *handler*) sind Anweisungen. In der NQC-API sind Konstanten für einzelne Betriebsmittel festgelegt die addiert werden können um mehrere Betriebsmittel gleichzeitig zu belegen. Die Wirkung der `acquire`-Anweisung ist wie folgt: Der Zugriff auf ein Betriebsmittel wird angefordert. Wenn eine andere Task mit höherer Priorität das Betriebsmittel besitzt wird die Anforderung scheitern und die Programmausführung springt in den Botschafteninterpreter (falls einer vorhanden ist). Im anderen Fall ist die Anforderung erfolgreich und der Rumpf der Anweisung wird ausgeführt. Wenn bei der Ausführung des Rumpfes eine andere Task mit gleicher oder höherer Priorität eines der zugewiesen Betriebsmittel anfordert verliert die laufende Task den Zugriff darauf. Wenn der Zugriff verloren geht springt die Programmausführung in den Botschafteninterpreter (falls einer vorhanden ist). Die Betriebsmittel werden an das Betriebssystem zurückgeben sobald der Rumpf vollständig ausgeführt ist (Tasks mit geringerer Priorität können diese nun anfordern). Die Programmausführung wird mit der nächsten Anweisung nach der `acquire`-Anweisung fortgesetzt. Wenn kein Botschafteninterpreter angegeben ist wird in beiden Fällen (fehlgeschlagene Anforderung oder Verlust des Zugriffs) die Programmausführung mit der nächsten Anweisung nach der `acquire`-Anweisung fortgesetzt. Der nachfolgende Programmausschnitt fordert z.B. ein Betriebsmittel für 10 Sekunden an und spielt eine Klang ab, wenn er nicht erfolgreich zu Ende laufen kann:

```
acquire ( ACQUIRE_OUT_A)
{
  Wait(1000);
}
catch
{
  PlaySound(SOUND_UP);
}
```

Die Ereignisüberwachung wird mit der `monitor`-Anweisung durchgeführt, die eine Syntax ähnlich der `acquire`-Anweisung besitzt:

`monitor (` *Ereignisse* `)` *Rumpf*

`monitor (` *Ereignisse* `)` *Rumpf Botschafteninterpreter_ Liste*

wobei die *Botschafteninterpreter_ Liste* einen oder mehrere Botschafteninterpreter in der folgenden Form angibt:

`catch (` *Ereignisse* `)` *Botschafteninterpreter*

Beim letzten Botschafteninterpreter der Liste kann die Ereignisangabe fehlen:

`catch` *Botschafteninterpreter*

Ereignisse ist dabei eine Konstante, die die zu überwachenden Ereignisse festlegt. Beim Scout sind die Ereignisse fest vorgegeben, sodaß Konstanten wie etwa `EVENT_1_PRESSED` zur Bezeichnung der Ereignisse verwendet werden können. Beim RCX2 wird die Bedeutung jedes Ereignisses durch den Programmierer festgelegt. Es gibt 16 Ereignisse (Nummer 0 bis 15). Um ein Ereignis in einer `monitor`-Anweisung zu verwenden muß die Ereignisnummer mit Hilfe des Makros `EVENT_MASK()` in eine Ereignismaske umgewandelt werden. Mehrere Masken können mit einem bitweisen Oder verknüpft werden.

Die `monitor`-Anweisung führt ihren Rumpf aus während auf die Ereignisse gewartet wird. Wenn irgendeines der Ereignisse eintritt wird der erste Botschafteninterpreter für dieses Ereignis angesprungen (ein Botschafteninterpreter ohne eine Ereignisangabe behandelt jedes Ereignis). Wenn kein Botschafteninterpreter existiert wird die erste Anweisung nach der `monitor`-Anweisung ausgeführt. Im nachfolgenden Beispiel wird 10 Sekunden auf das Eintreffen der Ereignisse 2, 3 und 4 im RCX2 gewartet:

```
monitor( EVENT_MASK(2) | EVENT_MASK(3) | EVENT_MASK(4) )
{
  Wait(1000);
}
catch( EVENT_MASK(4) )
{
  PlaySound(SOUND_DOWN);   // Ereignis 4 eingetreten
}
catch
{
  PlaySound(SOUND_UP);    // Ereignis 2 oder 3 eingetreten
}
```

Zu beachten ist, daß die `acquire`- und `monitor`-Anweisungen nur bei Geräten unterstützt werden, die Ereignisüberwachung und Zugriffskontrolle zur Verfügung stellen – das sind insbesondere der Scout und der RCX2.

2.3.5 Weitere Anweisungen

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

Name `(` *Argumentliste* `);`

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-Operator (`++`) oder den Dekrement-Operator (`--`) verwenden.

```
x++;
```

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

2.4 Ausdrücke

Frühere Versionen von NQC haben zwischen Ausdrücken und Bedingungen unterschieden. Ab Version 2.3 wurde diese Unterscheidung fallen gelassen: alles ist ein Ausdruck und es gibt nun Vergleichsoperatoren für Ausdrücke. Das ist vergleichbar mit der Verfahrensweise in C/C++.

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 gehören.

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.

Zwei besondere Werte sind vordefiniert: `true` und `false`. Der interne Wert von `false` ist Null, während für den internen Wert von `true` nur garantiert ist, daß er ungleich Null ist. Dasselbe gilt für Vergleichsoperatoren (z.B. `<`): wenn der Vergleich unwahr ist, dann ist der Wert 0, andernfalls ist er ungleich 0.

Operator	Beschreibung	Bindung	Einschränkung	Beispiel
`abs()` `sign()`	Absolutwert Vorzeichen des Operanden	nicht anwendbar nicht anwendbar		`abs(x)` `sign(x)`
`++ --`	Inkrement, Dekrement	links	nur Variablen	`x++` oder `++x`

`-`
`~`
`!`
	unäres Minus
	bitweise Negation (unär)
	logische Negation
		rechts
		rechts
		rechts
			nur Konstanten	`-x`
				`~123`
				`!x`

`* / %`	Multiplikation, Division, Modulo-Operator	links		`x*y`
`+ -`	Addition, Subtraktion	links		`x+y`
`<< >>`	Linksschieben, Rechtsschieben	links	konstante Verschiebung	`x<<4`
`< >` `<= >=`	Vergleichsoperatoren	links		`x<y`
`== !=`	Gleichheit, Ungleichheit	links		`x == 1`
`&`	bitweises Und	links		
`^`	bitweises exklusives Oder	links	nur Konstanten	`123^4`
`|`	bitweises Oder	links		`x|y`
`&&`	logisches Und	links		`x&&y`
`||`	logisches Oder	links		`x||y`
`?:`	bedingte Auswertung	nicht anwendbar		`x==1?y:x`

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 in der voranstehenden Tabelle mit fallender Priorität aufgeführt.

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
```

2.4.1 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
Ausdruck	wahr, wenn *Ausdruck* ungleich 0 ist
Ausdruck1 `==` *Ausdruck2*	wahr, wenn *Ausdruck1* gleich *Ausdruck2* ist
Ausdruck1 `!=` *Ausdruck2*	wahr, wenn *Ausdruck1* ungleich *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*	wahr, wenn *Bedingung* falsch ist (logische Negation)
Bedingung1 `&&` *Bedingung2*	wahr, wenn beide Bedingungen wahr sind (logisches Und)
Bedingung1 `||` *Bedingung2*	wahr, wenn mindestens eine Bedingung wahr ist (logisches Oder)

2.5 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.

2.5.1 #include

Die `#include`-Anweisung arbeitet wie erwartet. Zu beachten ist, daß der Dateiname in doppelte Anführungszeichen einzuschließen ist. 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 !
```

2.5.2 #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 Rückstrich (backslash '\') ermöglicht jedoch mehrzeilige Makros:

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

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

2.5.3 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 `defined()` wird ebenfalls unterstützt.

2.5.4 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 Ausgänge auf volle Leistung in Vorwärtsrichtung (aber noch ausgeschaltet). Die Initialisierungsfunktion kann durch die Anweisung `#pragma noinit` unterdrückt werden.

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

Die Ersatzinitialisierungsfunktion kann mit Hilfe der Anweisung `#pragma init` durch eine andere Funktion ersetzt werden.

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

2.5.5 Speicherplatzreservierung

Der NQC-Übersetzer weist Variablen automatisch Speicherplatz zu. Gelegentlich kann es erforderlich sein, dem Übersetzer die Verwendung bestimmter Speicherplätze zu verbieten. Das kann mit der Anweisung `#pragma reserve` geschehen:

```
#pragma reserve Anfangsadresse
#pragma reserve Anfangsadresse Endadresse
```

Diese Anweisung zwingt den Übersetzer, einen oder mehrere Speicherplätze bei der Platzzuweisung an Variablen nicht zu verwenden. *Anfangsadresse* und *Endadresse* müssen Zahlenwerte sein, die gültige Speicherplätze bezeichnen. Wenn nur eine Anfangsadresse angegeben ist, dann wird nur diese Adresse nicht verwendet. Wenn sowohl Anfangsadresse als auch Endadresse angegeben sind, dann wird der gesamte Bereich (einschließlich der Endadresse) nicht verwendet. Die häufigste Anwendung diese Anweisung ist die Sperrung der Speicherplätze 0, 1 und/oder 2, wenn im RCX2 Zähler verwendet werden. Das liegt darin begründet, daß die RCX-Zähler sich mit den Speicherplätzen 0, 1 und 2 überlappen. Wenn alle drei Zähler verwendet werden sollen sieht das so aus:

```
#pragma reserve 0 2
```

3

NQC-API

Das NQC-API legt eine Reihe von Konstanten, Funktionen, Werten und Makros fest, die Zugriff auf Einrichtungen des Zielgerätes, wie etwa Sensoren und Ausgänge, erlauben. Einige Möglichkeiten sind nur bei bestimmten Geräten vorhanden. Wo das zutrifft, werden in der Abschnittsüberschrift die entsprechenden Geräte genannt. Die Fähigkeiten des RCX2 sind eine Obermenge der Fähigkeiten des RCX, d.h. wenn der RCX aufgeführt ist, dann stehen diese Fähigkeiten sowohl bei der Original-Firmware als auch bei der 2.0-Firmware zur Verfügung. Ist der RCX2 aufgeführt, dann stehen die Fähigkeiten nur mit der 2.0-Firmware zur Verfügung. CyberMaster, Scout und Spybotics werden im folgenden durch CM, Scout und Spy angezeigt.

Die API besteht aus Funktionen, Variablen und Konstanten. Eine Funktion kann in einer Anweisung aufgerufen werden. Üblicherweise führt sie einige Aktionen aus oder legt gewisse Parameter fest. Werte stehen für gewisse Parameter oder Mengenangaben und können ebenfalls in Ausdrücken verwendet werden. Konstanten sind symbolische Namen für Werte, die bei einem Gerät eine besondere Bedeutung haben. Öfters wird eine Menge von Konstanten zusammen mit einer Funktion verwendet. So übernimmt z.B. die Funktion `PlaySound` ein einzelnes Argument welches den Klang festlegt, der gespielt werden soll. Für jeden Klang sind Konstanten, wie etwa `SOUND_UP`, festgelegt.

3.1 Sensoren

Es gibt drei Sensoren, die intern mit 0, 1 und 2 numeriert sind. Das ist möglicherweise verwirrend, weil sie extern am RCX als Sensoren 1, 2 und 3 bezeichnet sind. Um die Verwirrung zu mildern wurden Namen `SENSOR_1`, `SENSOR_2` und `SENSOR_3` festgelegt. Diese Namen können in allen Funktionsaufrufen verwendet werden, die Sensoren als Argumente verlangen. Weiterhin können die Namen immer dort verwendet werden wo ein Programm einen Sensorwert lesen möchte:

```
x = SENSOR_1;     // lies Sensor und speichere Wert in x
```

3.1.1 Typen und Betriebsarten RCX, CyberMaster

Die Sensorschnittstellen des RCX können mit mehreren unterschiedlichen Sensoren verbunden werden (andere Geräte haben keine konfigurierbaren Sensoren). Es ist dem Programm überlassen dem RCX mitzuteilen, welche Sensoren mit welcher Schnittstelle verbunden sind. Ein Sensortyp kann mit dem Aufruf `SetSensorType` angegeben werden. Es gibt vier Sensortypen, die jeweils einem LEGO-Sensor entsprechen. Ein fünfter Typ (`SENSOR_TYPE_NONE`) kann dazu verwendet werden, einen Rohwert aus einem beliebigen passiven Sensor auszulesen. Im Allgemeinen sollte ein Programm den Typ passend zum tatsächlich verwendeten Sensor setzen. Wenn eine Schnittstelle für einen falschen Typ konfiguriert ist kann RCX möglicherweise den Sensor nicht richtig auslesen.

Sensortyp	Bedeutung
`SENSOR_TYPE_NONE`	beliebiger passiver Sensor
`SENSOR_TYPE_TOUCH`	Berührungssensor
`SENSOR_TYPE_TEMPERATURE`	Temperatursensor
`SENSOR_TYPE_LIGHT`	Lichtsensor
`SENSOR_TYPE_ROTATION`	Rotationssensor

Beim RCX, CyberMaster und Spybotics kann ein Sensor auf unterschiedliche Betriebsarten (Modi) eingestellt werden. Der Modus legt fest, wie der Rohwert eines Sensors verarbeitet wird. Einige Modi machen nur bei bestimmten Sensoren Sinn, so ist z.B. der Modus `SENSOR_MODE_ROTATION` nur bei einem Rotationssensor sinnvoll. Der Modus kann mit einem Aufruf der Funktion `SetSensorMode` eingestellt werden. Die möglichen Modi sind weiter unten aufgelistet. Da der CyberMaster weder Rotationssensoren noch Temperatursensoren besitzt sind die letzten drei Einträge auf den RCX beschränkt. Der Spybotics ist noch stärker eingeschränkt und erlaubt nur den raw-, boolean- und percentage-Modus.

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`	Werte von 0 bis 100
`SENSOR_MODE_FAHRENHEIT`	Grad Fahrenheit - nur RCX
`SENSOR_MODE_CELSIUS`	Grad Celsius - nur RCX
`SENSOR_MODE_ROTATION`	Drehlage (16 Schritte pro Umdrehung) - nur RCX

Beim RCX ist es üblich, den Typ und den Modus gleichzeitig zu setzen. Die Funktion `SetSensor` erleichtert das, indem sie mit einem Aufruf das Setzen einer Typ-Modus-Kombination ermöglicht.

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`	`SENSOR_MODE_EDGE`

Der RCX stellt für alle Sensortypen, nicht nur für Berührungssensoren, auch boolesche Ausgabewerte bereit. Der boolesche Wert wird normalerweise mit Hilfe eines voreingestellten Schwellwertes vom Rohwert des Sensors abgeleitet. Ein „niederer“ Wert (kleiner als 460) ergibt den booleschen Wert 1. Ein hoher Wert (größer als 562) ergibt den booleschen Wert 0. Diese Umwandlung kann beeinflußt werden: beim Aufruf von `SetSensorMode` kann zum Sensormodus ein Anstiegswert (engl. *slope value*) zwischen 0 und 31 hinzugefügt werden. Wenn sich der Sensorwert innerhalb einer gewissen Zeit (3 ms) stärker als dieser Wert ändert, dann wechselt der booleschen Wert des Sensors. Dadurch kann der boolesche Wert einem raschen Wechsel des Sensorrohwertes folgen. Ein schneller Anstieg ergibt den booleschen Wert 0, ein schneller Abfall ergibt den booleschen Wert 1.

Selbst wenn ein Sensor auf einen anderen Modus eingestellt ist kann die Umwandlung in einen booleschen Wert durchgeführt werden.

SetSensor (*Sensor*, *Konfiguration*) **Funktion - RCX**

Setze Typ und Modus des angegebenen Sensors auf die angegebene Konfiguration. Diese muß eine spezielle Konstante sein, die sowohl den Typ als auch den Modus enthält.

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

SetSensorType (*Sensor*, *Typ*) **Funktion - RCX**

Setze den Sensortyp, der eine der vordefinierten Sensortypkonstanten sein muß.

```
SetSensorType(SENSOR_1, SENSOR_TYPE_TOUCH);
```

SetSensorMode (*Sensor*, *Modus*) **Funktion - RCX, CM, Spy**

Setze den Sensormodus. Eine der vordefinierten Sensormoduskonstanten sollte verwendet werden. Falls erwünscht, kann ein Anstiegswert für eine boolesche Umwandlung hinzugefügt werden (nur RCX).

```
SetSensorMode(SENSOR_1, SENSOR_MODE_RAW);         // Rohmodus
SetSensorMode(SENSOR_1, SENSOR_MODE_RAW + 10);    // Anstiegswert 10
```

ClearSensor (*Sensor*, *Typ*) **Funktion - Alle**

Sensorwert löschen – beeinflußt nur diejenigen Sensoren, die so eingestellt sind, daß sie einen Wert durch Weiterzählung messen (z.B. Rotationssensoren und Pulsgeber).

```
ClearSensor(SENSOR_1);
```

3.1.2 Sensorinformationen

Es gibt eine Reihe von Werten, die von einem Sensor abgerufen werden können. Dabei muß der Sensor durch seine Sensornummer (`0`, `1` oder `2`), nicht durch seinen Namen (z.B. `SENSOR_1`), angegeben werden.

SensorValue (*n*) **Wert - Alle**

Gibt den bearbeiteten Wert von Sensor *n* zurück, wobei *n* gleich `0`, `1` oder `2` ist. Das ist derselbe Wert der durch die Sensornamen (z.B. `SENSOR_1`) zurückgeben wird.

```
x = SensorValue( 0 );     // lies Sensor 1
```

SensorType (*n*) **Wert - RCX, CM, Scout**

Gibt den augenblicklich konfigurierten Typ von Sensor *n* zurück, wobei *n* gleich `0`, `1` oder `2` ist. Nur der RCX hat konfigurierbare Sensortypen, alle anderen Geräte geben den voreingestellten Sensortyp zurück.

```
x = SensorType( 0 );
```

SensorMode (*n*) **Wert - RCX, CM, Spy**

Gibt den augenblicklich konfigurierten Modus von Sensor *n* zurück, wobei *n* gleich `0`, `1` oder `2` ist.

```
x = SensorMode( 0 );
```

SensorValueBool (*n*) **Wert - RCX**

Gibt den booleschen Wert von Sensor *n* zurück, wobei *n* gleich `0`, `1` oder `2` ist. Die Ermittlung der booleschen Werte wird entweder auf der Grundlage voreingestellter Schwellwerte durchgeführt oder mit Hilfe eines Anstiegswertes, der bei einem Aufruf von `SetSensorMode` übergeben wurde.

```
x = SensorValueBool( 0 );
```

SensorValueRaw (*n*) **Wert - RCX, Scout, Spy**

Gibt den Rohwert von Sensor *n* zurück, wobei *n* gleich `0`, `1` oder `2` ist. Rohwerte liegen im Bereich von 0 bis 1023 (RCX) oder 0 bis 255 (Scout).

```
x = SensorValueRaw( 0 );
```

3.1.3 Scout-Lichtsensor **Scout**

Beim Scout bezeichnet `SENSOR_3` den eingebauten Lichtsensor. Das Auslesen des Lichtsensors (mit `SENSOR_3`) ergibt eine von drei Stufen: `0` (dunkel), `1` (normal) oder `2` (hell). Der Rohwert des Sensors kann mit `SensorValueRaw(SENSOR_3)` gelesen werden, dabei ist aber zu beachten daß helleres Licht niedrigere Rohwerte ergibt. Die Umwandlung des Rohwertes (zwischen 0 und 1023) in eine der drei Stufen hängt von drei Parametern ab: *Untergrenze*, *Obergrenze* und *Hysterese*. Die Untergrenze ist der kleinste (hellste) Rohwert der noch als *normal* betrachtet wird. Werte unterhalb der Untergrenze werden der Stufe *hell* zugeordnet. Die Obergrenze ist der größte (dunkelste) Rohwert der noch als *normal* betrachtet wird. Werte oberhalb der Obergrenze werden der Stufe *dunkel* zugeordnet.

Hysterese kann dazu verwendet werden einen Wechsel einer Stufe durch Schwankungen des Rohwertes um eine Stufengrenze zu unterdrücken. Das wird erreicht, indem man das Verlassen der Stufen *dunkel* und *hell* etwas schwieriger macht als das Erreichen der Stufen. Genauer gesagt liegt die Grenze für den Wechsel von *normal* nach *hell* ein wenig tiefer als die Grenze für den Wechsel von *hell* nach *normal*. Der Unterschied zwischen diesen beiden Grenzen ist der Wert der *Hysterese*. Die spiegelbildlichen Verhältnisse gelten für den Wechsel zwischen *normal* und *dunkel*.

SetSensorLowerLimit (*Wert*) **Funktion - Scout**

Untergrenze des Lichtsensors setzen. Der Wert darf ein beliebiger Ausdruck sein.

```
SetSensorLowerLimit( 100 );
```

SetSensorUpperLimit (*Wert*) **Funktion - Scout**

Obergrenze des Lichtsensors setzen. Der Wert darf ein beliebiger Ausdruck sein.

```
SetSensorUpperLimit( 900 );
```

SetSensorHysteresis (*Wert*) **Funktion - Scout**

Hysterese des Lichtsensors setzen. Der Wert darf ein beliebiger Ausdruck sein.

```
SetSensorHysteresis( 20 );
```

CalibrateSensor ( ) **Funktion - Scout**

Liest den augenblicklichen Wert des Lichtsensors und setzt dann die Ober- und Untergrenze auf Werte 12,5 Prozent ober- und unterhalb des augenblicklichen Wertes. Der Hysteresewert wird auf 3,12 Prozent des augenblicklichen Wertes gesetzt.

```
CalibrateSensor( );
```

3.1.4 Spybotics-Sensoren Spy

Der Spybotics besitzt statt extern anschließbarer Sensoren eingebaute Sensoren. Der Berührungssensor an der Vorderseite des Spybotics ist `SENSOR_1`. Er ist normalerweise auf den Prozentmodus eingestellt, hat also im nicht gedrückten Zustand den Wert 0 und im gedrückten Zustand den Wert 100.

`SENSOR_2` ist der Lichtsensor (die Schnittstelle auf der Geräterückseite die zur Datenübertragung mit dem Rechner verwendet wird). Er ist normalerweise auf den Prozentmodus eingestellt, wobei höhere Werte helleres Licht bedeuten.

3.2 Ausgänge

3.2.1 Einfache Funktionsaufrufe

Alle Funktionen für die Behandlung von Ausgängen haben als erstes Argument eine oder mehrere Ausgänge. Die Namen `OUT_A`, `OUT_B` und `OUT_C` bezeichnen die drei Ausgänge. 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 zusammen zu verwenden. Die Angabe der Ausgänge muß bereits zur Übersetzungszeit einen konstanten Wert ergeben (kann also keine Variable sein).

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, 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`	setze Richtung auf vorwärts
`OUT_REV`	setze Richtung auf rückwärts
`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 Leistungsstufe verwendet werden. Die Leistungsstufe wird mit dem Befehl `SetPower(` ***`Ausgänge`***`,` ***`Leistung`*** `)` eingestellt.

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

SetOutput (*Ausgänge*, *Modus*) **Funktion - Alle**

Setzt den angegeben Modus für die Ausgänge. *Ausgänge* sind eine oder mehrere der Bezeichnungen `OUT_A`, `OUT_B` und `OUT_C`. *Modus* ist `OUT_ON`, `OUT_OFF` oder `OUT_FLOAT`.

```
SetOutput( OUT_A + OUT_B, OUT_ON );     // A und B einschalten
```

SetDirection (*Ausgänge*, *Richtung*) **Funktion - Alle**

Setzt die angegeben Richtung für die Ausgänge. *Ausgänge* sind eine oder mehrere der Bezeichnungen `OUT_A`, `OUT_B` und `OUT_C`. *Richtung* ist `OUT_FWD`, `OUT_REV` oder `OUT_TOGGLE`.

```
SetDirection( OUT_A, OUT_REV );     // Richtung von A umkehren
```

SetPower (*Ausgänge*, *Leistung*) **Funktion - Alle**

Setzt die Leistungsstufe für die angegeben Ausgänge. *Leistung* kann ein Ausdruck sein, sollte aber einen Wert zwischen 0 und 7 ergeben. Die Konstanten `OUT_LOW`, `OUT_HALF` und `OUT_FULL` können verwendet werden.

```
SetPower( OUT_A, OUT_FULL );     // A aus volle Leistung setzen
SetPower( OUT_B, x );
```

OutputStatus (*n*) **Wert - Alle**

Gibt den augenblicklichen Ausgangszustand von Motor *n* zurück. *n* muß 0, 1 oder 2 sein – nicht `OUT_A`, `OUT_B` oder `OUT_C`.

```
x = OutputStatus( 0 );           // Status von OUT_A
```

3.2.2 Vereinfachende Funktionsaufrufe

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.

On (*Ausgänge*) **Funktion - Alle**

Schaltet die angegebenen Ausgänge ein. *Ausgänge* sind eine oder mehrere der Bezeichnungen `OUT_A`, `OUT_B` und `OUT_C`, ggf. addiert.

```
On( OUT_A + OUT_C );           // Ausgänge A und C einschalten
```

Off (*Ausgänge*) **Funktion - Alle**

Schaltet die angegebenen Ausgänge aus. *Ausgänge* sind eine oder mehrere der Bezeichnungen `OUT_A`, `OUT_B` und `OUT_C`, ggf. addiert.

```
Off( OUT_A );                  // Ausgang A ausschalten
```

Float (*Ausgänge*) **Funktion - Alle**

Schaltet die angegebenen Ausgänge frei. *Ausgänge* sind eine oder mehrere der Bezeichnungen `OUT_A`, `OUT_B` und `OUT_C`, ggf. addiert.

```
Float( OUT_A );                // Ausgang A im Freilauf
```

Fwd (*Ausgänge*) **Funktion - Alle**

Schaltet die angegebenen Ausgänge in Vorwärtsrichtung. *Ausgänge* sind eine oder mehrere der Bezeichnungen `OUT_A`, `OUT_B` und `OUT_C`, ggf. addiert.

```
Fwd( OUT_A );
```

Rev (*Ausgänge*) **Funktion - Alle**

Schaltet die angegebenen Ausgänge in Rückwärtsrichtung. *Ausgänge* sind eine oder mehrere der Bezeichnungen `OUT_A`, `OUT_B` und `OUT_C`, ggf. addiert.

```
Rev( OUT_A );
```

Toggle (*Ausgänge*) **Funktion - Alle**

Kehrt die Richtung der angegebenen Ausgänge um. *Ausgänge* sind eine oder mehrere der Bezeichnungen `OUT_A`, `OUT_B` und `OUT_C`, ggf. addiert.

```
Toggle( OUT_A );
```

OnFwd (*Ausgänge*) **Funktion - Alle**

Schaltet die angegebenen Ausgänge in Vorwärtsrichtung und schaltet sie ein. *Ausgänge* sind eine oder mehrere der Bezeichnungen `OUT_A`, `OUT_B` und `OUT_C`, ggf. addiert.

```
OnFwd( OUT_A );
```

OnRev (***Ausgänge***) **Funktion - Alle**

Schaltet die angegebenen Ausgänge in Rückwärtsrichtung und schaltet sie ein. *Ausgänge* sind eine oder mehrere der Bezeichnungen `OUT_A`, `OUT_B` und `OUT_C`, ggf. addiert.

```
OnRev( OUT_A );
```

OnFor (***Ausgänge***, ***Zeitdauer***) **Funktion - Alle**

Schaltet die angegebenen Ausgänge für eine Zeitdauer ein und dann wieder aus. *Ausgänge* sind eine oder mehrere der Bezeichnungen `OUT_A`, `OUT_B` und `OUT_C`, ggf. addiert. *Zeitdauer* wird in 10ms-Schritten angegeben (1 Sekunde = 100 Schritte) und kann ein beliebiger Ausdruck sein.

```
OnFor( OUT_A, x );
```

3.2.3 Globale Kontrolle RCX2, Scout

SetGlobalOutput (***Ausgänge***, ***Modus***) **Funktion - RCX2, Scout, Spy**

Abhängig von einem Modusparameter werden Ausgänge freigegeben oder gesperren. Wenn der Modus `OUT_OFF` ist werden die Ausgänge gesperrt. Im gesperrten Zustand bleiben alle nachfolgenden Aufrufe von `SetOutput()` wirkungslos (einschließlich der vereinfachenden Aufrufe wie `On()`). Die Modusangabe `OUT_FLOAT` schaltet die Ausgänge vor der Sperrung in den Freilauf. Ausgänge können durch einen Aufruf von `SetGlobalOutput()` mit dem Modus `OUT_ON` wieder freigegeben werden. Es ist zu beachten, daß das Freigeben eines Ausganges diesen nicht sofort einschaltet – es ermöglicht lediglich den zukünftigen Aufrufen von `SetOutput()` wirksam zu werden.

```
SetGlobalOutput( OUT_A, OUT_OFF );    // Ausgang A sperren
SetGlobalOutput( OUT_A, OUT_ON );     // Ausgang A freigegeben
```

SetGlobalDirection (***Ausgänge***, ***Richtung***) **Funktion - RCX2, Scout, Spy**

Kehrt die Richtung von Ausgängen um oder stellt sie wieder her. Der Parameter *Richtung* sollte eine der Konstanten `OUT_FWD`, `OUT_REV` oder `OUT_TOGGLE` sein. Normalerweise ist die globale Richtung `OUT_FWD`. Wenn die globale Richtung `OUT_REV` ist, dann stellen die Aufrufe der Ausgabefunktionen die jeweils gegenteilige Richtung gegenüber dem Normalzustand ein. Ein Aufruf von `SetGlobalDirection()` mit `OUT_TOGGLE` schaltet zwischen normalem und gegenteiligem Vehalten um.

```
SetGlobalDirection( OUT_A, OUT_REV );    // Gegenrichtung
SetGlobalDirection( OUT_A, OUT_FWD );    // normale Richtung
```

SetMaxPower (*Ausgänge*, *Leistungsstufe*) **Funktion - RCX2, Scout, Spy**

Setzt die höchstzulässige Ausgangsleistung für die Ausgänge. *Leistungsstufe* kann eine Variable sein, sollte aber zwischen `OUT_LOW` und `OUT_FULL` liegen.

```
SetMaxPower( OUT_A, OUT_HALF );
```

GlobalOutputStatus (*n*) **Wert - RCX2, Scout, Spy**

Gibt den augenblicklichen Ausgangszustand von Motor *n* zurück. *n* muß 0, 1 oder 2 sein – nicht `OUT_A`, `OUT_B` oder `OUT_C`.

```
x = GlobalOutputStatus( 0 );             // globaler Status von OUT_A
```

3.2.4 Spybotics-Ausgänge

Der Spybotics hat zwei eingebaute Motoren. Der rechte Motor wird mit `OUT_A` bezeichnet, der linke Motor mit `OUT_B`. Über `OUT_C` werden VLL-Befehle über die LED an der Rückseite ausgegeben (wird zur Datenübertragung zum Computer verwendet). Das ermöglicht die Verwendung eines VLL-Gerätes, wie z.B. der Micro-Scout, als dritten Motor am Spybotics. Die selbe LED kann mittels der Funktionen `SendVLL()` und `SetLight()` angesteuert werden.

3.3 Klänge

PlaySound (*n*) **Funktion - Alle**

Spielt einen von 6 in RCX festgelegten Klängen. Das Argument *n* muß eine Konstante sein (außer beim Spybotics, bei dem eine Variable verwendet werden kann). 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*) **Funktion - Alle**

Gibt einen Ton der angegebenen Frequenz und Dauer aus. *Frequenz* wird in Hertz angegeben und kann beim RCX2, Scout und Spybotics eine Variable sein, muß beim RCX und beim CyberMaster aber eine Konstante sein. *Dauer* wird in in 100-stel Sekunden angegeben und muß eine Konstante sein.

```
PlayTone( 440, 50 );        // ein halbe Sekunde den Ton 'A' spielen
```

MuteSound ( ) **Funktion - RCX2, Scout, Spy**

Die Klangausgabe unterdrücken.

```
MuteSound( );
```

UnMuteSound ( ) **Funktion - RCX2, Scout, Spy**

Die Klangausgabe wieder zulassen.

```
UnMuteSound( );
```

ClearSound ( ) **Funktion - RCX2, Spy**

Alle noch gespeicherten Klänge aus dem Klangpuffer entfernen.

```
ClearSound( );
```

SelectSounds (*Gruppe*) **Funktion - Scout**

Legt fest, welche Gruppe von Systemklängen verwendet werden soll. *Gruppe* muß eine Konstante sein.

```
SelectSounds( 0 );
```

3.4 LCD-Display

Der RCX besitzt 7 unterschiedliche Display-Modi. 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

Der RCX2 hat einen achten Displaymodus – `DISPLAY_USER`. In diesem Benutzer-Display-Modus wird fortlaufend ein Quellwert ausgelesen und die Anzeige aufgefrischt. Zusätzlich kann an beliebiger Stelle innerhalb einer Zahl ein Dezimalpunkt angezeigt werden. Das Display erlaubt scheinbar die Anzeige von Brüchen, obwohl intern alle Werte als ganze Zahlen gespeichert sind. So zeigt z.B. der folgende Aufruf im Display den Wert 1234 mit zwei Ziffern nach dem Dezimalpunkt. Als Ergebnis erscheint `12.34` auf der Anzeige.

```
SetUserDisplay( 1234, 2 );
```

Das folgende kurze Programm zeigt das Auffrischen des Displays:

```
task main ()
{
  ClearTimer(0);
  SetUserDisplay( Timer(0), 0 );
  until(false);
}
```

Weil im Benutzer-Display-Modus die LCD-Anzeige fortlaufend aufgefrischt wird bestehen gewisse Einschränkungen für den Quellwert. Wenn eine Variable verwendet wird, dann muß diese einen globalen Speicherplatz besitzen. Der beste Weg um dies zu erreichen ist, die Variable als global zu vereinbaren. Es kann außerdem seltsame Seiteneffekte geben. Wenn zum Beispiel eine Variable dargestellt und später als Ergebnisvariable für eine Berechnung verwendet wird, dann ist es möglich, daß in der Anzeige einige Zwischenergebnisse erscheinen:

```
int x;
task main ()
{
  SetUserDisplay( x, 0 );
  while(true)
  {
    // Anzeige zeigt kurz den Wert 1
    x = 1 + Timer(0);
  }
}
```

SelectDisplay (*Modus*) **Funktion - RCX**

Darstellungsmodus auswählen.

```
SelectDisplay( DISPLAY_SENSOR_1 );     // zeige Sensor 1
```

SetUserDisplay (*Wert*, *Genauigkeit*) **Funktion - RCX2**

Steuert das LCD-Display so, daß fortlaufend der angegebene *Wert* angezeigt wird. *Genauigkeit* gibt die Anzahl der Stellen rechts vom Dezimalpunkt an. Die *Genauigkeit* 0 erzeugt keinen Dezimalpunkt.

```
SetUserDisplay( Timer(0), 0 );     // zeige Zeitzähler 0
```

3.5 Kommunikation

3.5.1 Nachrichten **RCX, Scout**

Der RCX und der Scout können über die Infrarotschnittstelle einfache Nachrichten senden und empfangen. Eine Nachricht besteht aus einem Wert zwischen 0 und 255, die Verwendung der Nachricht 0 wird aber nicht empfohlen. Die jeweils zuletzt empfangene Nachricht wird gespeichert und kann durch `Message()` gelesen werden. Wenn noch keine Nachricht empfangen wurde wird eine 0 zurückgegeben. Es ist zu beachten, daß wegen der Arbeitsweise der Infrarotübertragung kein Empfang möglich ist solange eine Nachricht gesendet wird.

ClearMessage ( ) **Funktion - RCX, Scout**

Löscht den Nachrichtenpuffer. Das erleichtert die Erkennung der nächsten zu empfangenden Nachricht weil das Programm nun warten kann bis der Rückgabewert von `Message()` ungleich 0 wird:

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

SendMessage (*Nachricht*) **Funktion - RCX, Scout**

Eine Infrarot-Nachricht senden. *Nachricht* kann ein beliebiger Ausdruck sein. Der RCX kann jedoch nur Nachrichten mit einem Wert zwischen 0 und 255 senden, d.h. nur die niederwertigsten 8 Bit des Arguments werden verwendet.

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

SetTxPower (*Leistung*) **Funktion - RCX, Scout**

Setze die Sendeleistung für die Infrarotübertragung. *Leistung* sollte eine der Konstanten `TX_POWER_LO` oder `TX_POWER_HI` sein.

3.5.2 Serielle Kommunikation RCX2

Der RCX2 kann seriell Daten über die Infrarotschnittstelle übertragen. Vor jeder Übertragung müssen die Übertragungs- und Paketeinstellungen angegeben werden. Dann können für jede Übertragung die Daten in den Übertragungspuffer geschrieben und mit der Funktion `SendSerial()` übermittelt werden.

Die Übertragungseinstellungen werden mit `SetSerialComm` eingestellt und legen fest, wie die Bits über Infrarot übertragen werden. Die folgenden Werte sind möglich:

Option	Wirkung
`SERIAL_COMM_DEFAULT`	Ersatzwert
`SERIAL_COMM_4800`	4800 Baud
`SERIAL_COMM_DUTY25`	25% Tastverhältnis
`SERIAL_COMM_76KHZ`	76 kHz Trägerfrequenz

Die Ersatzwerte sind: Senden mit 2400 Baud bei einem Tastverhältnis (engl. *duty cycle*) von 50% auf einem 38kHz-Träger. Zur Angabe mehrerer Optionen (z.B. 4800 Baud bei 25% Tastverhältnis) können die einzelnen Optionen mit einem bitweisen Oder verknüpft werden (`SERIAL_COMM_4800 | SERIAL_COMM_DUTY25`).

Die Paketeinstellungen werden mit `SetSerialPacket` eingestellt und legen fest, wie die Bytes zu Paketen zusammengestellt werden. Die folgenden Werte sind möglich:

Option	Wirkung
`SERIAL_PACKET_DEFAULT`	kein Paketformat – nur Datenbytes
`SERIAL_PACKET_PREAMPLE`	Paketvorspann senden
`SERIAL_PACKET_NEGATED`	jedes Byte ist durch sein Komplement dargestellt (negiert)
`SERIAL_PACKET_CHECKSUM`	Prüfsumme für jedes Paket übertragen
`SERIAL_PACKET_RCX`	Standard-RCX-Format (Vorspann, Daten negiert, Prüfsumme)

Negierte Pakete haben immer eine Prüfsumme. Die Option `SERIAL_PACKET_CHECKSUM` besitzt also nur eine Bedeutung wenn `SERIAL_PACKET_NEGATED` nicht angegeben wurde. Weiterhin ist die Verwendung von Vorspann, negierter Darstellung und Prüfsumme in der Option `SERIAL_PACKET_RCX` enthalten.

Der Übertragungspuffer kann bis zu 16 Datenbytes aufnehmen. Diese Bytes können mit `SetSerialData` übergeben und mit `SendSerial` übertragen werden. So sendet z.B. der folgende Programmcode zwei Bytes (`0x12` und `0x34`) über die serielle Verbindung:

```
SetSerialComm( SERIAL_COMM_DEFAULT );
SetSerialPacket( SERIAL_PACKET_DEFAULT );
SetSerialData( 0, 0x12 );
SetSerialData( 1, 0x34 );
SendSerial( 0, 2 );
```

SetSerialComm (*Einstellungen*) **Funktion - RCX2**

Übergebe die Übertragungseinstellungen die festlegen, wie Bits über die Infrarotschnittstelle übertragen werden.

```
SetSerialComm( SERIAL_COMM_DEFAULT );
```

SetSerialPacket (*Einstellungen*) **Funktion - RCX2**

Übergebe die Paketeinstellungen die festlegen, wie Bytes zu einem Paket zusammengestellt werden.

```
SetSerialPacket( SERIAL_PACKET_DEFAULT );
```

SetSerialData (*n*, *Wert*) **Funktion - RCX2**

Ein Datenbyte in den Übertragungspuffer schreiben. *n* ist der Index des zu schreibenden Bytes (0 - 15), *Wert* kann ein beliebiger Ausdruck sein.

```
SetSerialData( 3, x );     // Byte 3 erhält den Wert von x
```

SerialData (*n*) **Wert - RCX2**

Gibt den Wert eines Byte im Übertragungspuffer zurück (NICHT im Empfangspuffer). *n* muß eine Konstante zwischen 0 und 15 sein.

```
x = SerialData( 7 );       // Byte 7 lesen
```

SendSerial (*Anfang*, *Anzahl*) **Funktion - RCX2**

Verwende den Inhalt des Übertragungspuffers um ein Paket zusammenzustellen und sende es über die Infrarotschnittstelle (entsprechend den Paket- und Übertragungseinstellungen). *Anfang* und *Anzahl* sind beides Konstanten die das erste Byte und die Anzahl der zu sendenden Bytes im Übertragungspuffer angeben.

```
SendSerial( 0, 2 );        // sende die ersten beiden Bytes im Puffer
```

3.5.3 VLL Scout

SendVLL (*Wert*) **Funktion - Scout, Spy**

Sendet ein Visible-Light-Link-Kommando (VLL), welches verwendet werden kann um mit dem MicroScout oder Code Pilot zu kommunizieren. Die einzelnen VLL-Kommandos sind im Scout-SDK beschrieben.

```
SendVLL( 4 );           // sende VLL-Kommando 4
```

3.6 Timer

Alle Geräte stellen mehrere Timer mit 100 ms Auflösung (10 Zählimpulse pro Sekunde) zur Verfügung. Der Scout besitzt 3 Timer, während der RCX, CyberMaster und Spybotics 4 haben. Die Timer laufen nach 32767 Zählimpulsen (etwa 55 Minuten) auf den Wert 0 über. Der Wert eines Timers kann mit **`Timer(n)`** gelesen werden, wobei *n* eine Konstante ist die den zu benutzenden Timer festlegt (0 - 2 beim Scout, 0 - 3 bei den anderen). Beim RCX2 und Spybotics besteht die Möglichkeit, mit Hilfe von **`FastTimer(n)`** dieselben Timer mit höherer Auflösung zu lesen. Dies liefert die Zählerwert mit einer Auflösung von 10 ms (100 Zählimpulse pro Sekunde).

ClearTimer (*n*) **Funktion - Alle**

Den angegebenen Timer auf 0 zurücksetzen.

```
ClearTimer( 0 );
```

Timer (*n*) **Wert - Alle**

Den augenblicklichen Wert des angegebenen Timers zurückgegeben (Auflösung 100 ms).

```
x = Timer( 0 );
```

SetTimer (*n*, *Wert*) **Funktion - RCX2, Spy**

Den angegebenen Timer auf den angegebenen *Wert* (beliebiger Ausdruck) setzen.

```
SetTimer( 0, x );
```

FastTimer (*n*) **Wert - RCX2, Spy**

Den augenblicklichen Wert des angegebenen Timers zurückgegeben (Auflösung 10 ms).

```
x = FastTimer( 0 );
```

3.7 Zähler RCX2, Scout, Spy

Zähler gleichen sehr einfachen Variablen die erhöht, vermindert und gelöscht werden können. Der Scout besitzt zwei Zähler (0 und 1), während der RCX2 und der Spybotics drei besitzen (0, 1 und 2). Beim RCX2 überlappen diese Zähler mit den globalen Speicherplätzen 0 - 2. Wenn diese verwendet werden sollen, sollte NQC durch ein `#pragma reserve` daran gehindert werden, diese Speicherplätze für reguläre Variablen zu verwenden. Um beispielsweise Zähler 1 zu verwenden schreibt man:

```
#pragma reserve 1
```

ClearCounter (*n*) **Funktion - RCX2, Scout, Spy**

Setze Zähler *n* auf 0 zurück. *n* muß beim Scout 0 oder 1 sein, beim RCX2 und Spybotics 0 - 2.

```
ClearCounter( 1 );
```

IncCounter (*n*) **Funktion - RCX2, Scout, Spy**

Erhöhe Zähler *n* um 1. *n* muß beim Scout 0 oder 1 sein, beim RCX2 und Spybotics 0 - 2.

```
IncCounter( 1 );
```

DecCounter (*n*) **Funktion - RCX2, Scout, Spy**

Vermindere Zähler *n* um 1. *n* muß beim Scout 0 oder 1 sein, beim RCX2 und Spybotics 0 - 2.

```
DecCounter( 1 );
```

Counter (*n*) **Wert - RCX2, Scout**

Gib den augenblicklichen Wert des Zählers *n* zurück. *n* muß beim Scout 0 oder 1 sein, beim RCX2 und Spybotics 0 - 2.

```
x = Counter( 1 );
```

3.8 Zugriffskontrolle **RCX2, Scout, Spy**

Die Zugriffskontrolle wird haupsächlich durch die `acquire`-Anweisung ausgeübt. Die Funktion `SetPriority` kann zur Festlegung der Priorität einer Task verwendet werden. Die folgenden Konstanten können zur Angabe von Ressourcen in einer `acquire`-Anweisung verwendet werden. Benutzerdefinierten Ressourcen stehen nur beim RCX2 zur Verfügung.

Konstante	**Betriebsmittel**

`ACQUIRE_OUT_A`
`ACQUIRE_OUT_B`
`ACQUIRE_OUT_C`
	Ausgänge

`ACQUIRE_SOUND`	Klang
`ACQUIRE_LED`	LEDs (nur Spybotics)

`ACQUIRE_USER_1`
`ACQUIRE_USER_2`
`ACQUIRE_USER_3`
`ACQUIRE_USER_4`
	benutzerdefiniert – nur RCX2

SetPriority (*p*) **Funktion - RCX2, Scout, Spy**

Setzt die Taskpriorität auf *p* (muß eine Konstante sein). RCX2 unterstützt die Prioritäten 0 - 255, während der Scout die Prioritäten 0 - 7 unterstützt. Zu beachten ist, daß die niedrigeren Werte die höheren Prioritäten bezeichnen.

```
SetPriority( 1 );
```

3.9 Ereignisse **RCX2, Scout**

Obwohl der RCX2, Scout und Spybotics die gleichen Ereignissteuerungen haben, besitzen RCX2 und Spybotics 16 vollständig konfigurierbar Ereignisse, während der Scout 15 vordefinierte Ereignisse hat. Die einzigen Funktionen die beiden gemeinsam sind, sind die Befehle zum Abfragen und Erzwingen von Ereignissen.

ActiveEvents (*Task*) **Wert - RCX2, Scout, Spy**

Gibt die Menge der Ereignisse zurück, die für eine bestimmte *Task* ausgelöst wurden.

```
x = ActiveEvents( 0 );
```

CurrentEvents ( ) **Wert - RCX2, Spy**

Gibt die Menge der Ereignisse zurück, die für die aktive Task ausgelöst wurden.

```
x = CurrentEvents( );
```

Events (*Ereignis*) **Wert - RCX2, Scout, Spy**

Löst per Funktionsaufruf ein bestimmtes *Ereignis* aus. Das kann nützlich sein um die Ereignishandhabung eines Programmes zu testen oder um Ereignisse für andere Zwecke zu simulieren. Zu beachten ist, daß sich die Eigenschaften der einzelnen Ereignisse für den RCX2 und den Scout leicht unterscheiden. Der RCX2 verwendet das Makro `EVENT_MASK` zur Berechnung der Ereignismaske während der Scout festgelegte Masken hat.

```
Events( EVENT_MASK(3) );          // RCX2-Ereignis auslösen
Events( EVENT_1_PRESSED );        // Scout-Ereignis auslösen
```

3.9.1 Konfigurierbare Ereignisse RCX2, Spy

Anmerkung. Spybotics-Ereignisse scheinen den RCXEreignissen ziemlich ähnlich zu sein, obwohl die NQC-API für den Spybotics nur sehr wenig getestet ist. Die folgenden Angaben wurden aus der Sicht des RCX2 geschrieben und noch nicht für den Spybotics angepaßt.

RCX2 und Spybotics besitzen ein äußerst anpassungsfähiges Ereignissystem. Es gibt 16 Ereignisse und jedes davon kann einer von mehreren Ereignisquellen (Ursache, die Ereignisse auslösen) und einem Ereignistyp (Bedingung für die Ereignisauslösung) zugeordnet werden. Abhängig vom Ereignistyp können weitere Parameter angegeben werden. Bei allen Funktionsaufrufen für die Konfigurierung wird ein Ereignis durch seine Ereignisnummer (Konstante zwischen 0 und 15) angegeben.

Zulässige Ereignisquellen sind Sensoren, Timer, Zähler und Nachrichtenpuffer. Ein Ereignis wird durch eine Aufruf von `SetEvent(` ***`Ereignis`***`,` ***`Quelle`***`,` ***`Typ`*** `)` konfiguriert. *Ereignis* ist eine konstante Ereignisnummer zwischen 0 und 15, *Quelle* ist die Ereignisquelle und *Typ* ist einer der unten gezeigten Ereignistypen (einige Kombinationen von Ereignisquellen und -typen sind nicht zulässig).

Ereignisart	**Bedingung**	**Ereignisquelle**
`EVENT_TYPE_PRESSED`	Wert wechselt auf *ein*	nur Sensoren
`EVENT_TYPE_RELEASED`	Wert wechselt auf *aus*	nur Sensoren
`EVENT_TYPE_PULSE`	Wert wechselt von *aus* über *ein* zu *aus*	nur Sensoren
`EVENT_TYPE_EDGE`	Wert wechselt von *aus* nach *ein* oder umgekehrt	nur Sensoren
`EVENT_TYPE_FASTCHANGE`	Wert ändert sich schnell	nur Sensoren
`EVENT_TYPE_LOW`	Wert wechselt auf *niedrig*	beliebig
`EVENT_TYPE_NORMAL`	Wert wechselt auf *normal*	beliebig
`EVENT_TYPE_HIGH`	Wert wechselt auf *hoch*	beliebig
`EVENT_TYPE_CLICK`	Wert wechselt von *niedrig* über *hoch* nach *niedrig*	beliebig
`EVENT_TYPE_DOUBLECLICK`	zwei Klicks innerhalb einer gewissen Zeit	beliebig
`EVENT_TYPE_MESSAGE`	neue Nachricht angekommen	nur `Message()`

Die vier ersten Ereignistypen verwenden den booleschen Wert eines Sensors und sind deshalb am nützlichsten bei Berührungssensoren. Beispiel: Ereignis Nummer 2 soll ausgelöst werden, wenn der Berührungssensor an Eingang 1 gedrückt wird:

```
SetEvent( 2, SENSOR_1, EVENT_TYPE_PRESSED );
```

Um `EVENT_TYPE_PULSE` oder `EVENT_TYPE_EDGE` verwenden zu können muß der Sensor im Modus `SENSOR_MODE_PULSE` oder `SENSOR_MODE_EDGE` konfiguriert sein.

`EVENT_TYPE_FASTCHANGE` sollte für Sensoren verwendet werden, die mit einem Anstiegswertparameter konfiguriert wurden.

Die nächsten drei Typen (`EVENT_TYPE_LOW`, `EVENT_TYPE_NORMAL` und `EVENT_TYPE_HIGH`) ordnen den Wert einer Ereignisquelle einem von drei Bereichen (*niedrig*, *normal* oder *hoch*) zu und lösen ein Ereignis aus, wenn ein Wert von einem Bereich in einen anderen wechselt. Die Bereiche werden durch die Größen *Untergrenze* und *Obergrenze* des Ereignisses festgelegt. Wenn der Wert kleiner als die Untergrenze ist wird der Zustand *niedrig* angenommen. Wenn der Wert größer als die Obergrenze ist wird der Zustand *hoch* angenommen. Der Zustand ist *normal*, wenn der Wert zwischen diesen Grenzen liegt.

Das folgende Beispiel konfiguriert das Ereignis 3 so, daß es ausgelöst wird wenn der Wert des Sensors an Eingang 2 in den Zustand *hoch* wechselt. Die Obergrenze wird auf 80 und die Untergrenze wird auf 50 gesetzt. Diese Konfiguration ist typisch für einen Lichtsensor der ein Ereignis auslöst sobald er ein helles Licht entdeckt.

```
SetEvent( 3, SENSOR_2, EVENT_TYPE_HIGH );
SetLowerLimit( 3, 50 );
SetUpperLimit( 3, 80 );
```

Wenn der Sensorwert schankt kann ein Hysteresewert verwendet werden um stabilere Übergänge zu erzeugen. Ein Hysterese bewirkt, daß der Wechsel von *niedrig* nach *normal* etwas höher liegt als der Wechsel von *normal* nach *niedrig*. Es ist gewissermaßen einfacher in den Zustand *niedrig* zu gelangen als aus dem Zustand heraus. Die spiegelbildlichen Verhältnisse gelten für den Wechsel zwischen *normal* und *hoch*.

Ein Wechsel von *niedrig* über *hoch* zurück nach *niedrig* löst ein Ereignis vom Typ `EVENT_TYPE_CLICK` aus, vorausgesetzt daß die ganze Abfolge schneller abläuft als der Parameter *Klickzeit* für diese Ereignis angibt. Wenn zwei Klicks mit kürzerem Abstand als *Klickzeit* aufeinanderfolgen, dann wird ein Ereignis vom Typ `EVENT_TYPE_DOUBLCLICK` ausgelöst. Das System zählt auch die gesamte Anzahl von Klicks für ein Ereignis.

Der letzte Ereignistyp, `EVENT_TYPE_MESSAGE`, kann nur zusammen mit der Ereignisquelle `Message()` verwendet werden. Das Ereignis wird immer dann ausgelöst, wenn eine neue Nachricht eingetroffen ist (auch wenn der Wert derselbe ist wie bei der vorherigen Nachricht).

Die `monitor`-Anweisung und einige API-Funktionen (z.B. `ActiveEvents()` oder `Events()`) müssen mit mehreren Ereignissen umgehen können. Das wird durch die Umwandlung von Ereignisnummern in Ereignismasken erreicht, die dann bitweise mit Oder verknüpft werden. Das Makro `EVENT_MASK(`*`Ereignis`*`)` wandelt eine Ereignisnummer in eine Maske. Um z.B. die Ereignisse 2 und 3 zu überwachen, könnte die folgende Anweisung verwendet werden:

```
monitor( EVENT_MASK(2) | EVENT_MASK(3) )
```

SetEvent (*Ereignis*, *Quelle*, *Typ*) **Funktion - RCX2, Spy**

Konfiguriere ein *Ereignis* (eine Zahl zwischen 0 und 15) für eine bestimmte *Quelle* und einen *Typ*. *Ereignis* und *Typ* müssen beide konstant sein.

```
SetEvent( 2, Timer(0), EVENT_TYPE_HIGH );
```

ClearEvent (*Ereignis*) **Funktion - RCX2, Spy**

Konfiguration für ein *Ereignis* löschen. Das verhindert die Auslösung dieses Ereignisses bis zu einer erneuten Konfigurierung.

```
ClearEvent( 2 );                // Ereignis 2 löschen
```

ClearAllEvents ( ) **Funktion - RCX2, Spy**

Konfiguration für all Ereignisse löschen.

```
ClearAllEvents( );
```

EventState (*Ereignis*) **Wert - RCX2, Spy**

Gibt den Status von einem bestimmten *Ereignis* zurück. Statuswerte können sein: `0` : *niedrig*, `1` : *normal*, `2` : *hoch*, `3` : nicht definiert, `4` : *Beginn der Kalibrierung*, `5` : *Kalibrierung läuft*.

```
x = EventState( );
```

CalibrateEvent (*Ereignis*, *Untergrenze*, *Obergrenze*, *Hysterese*) **Wert - RCX2, Spy**

Kalibriere das *Ereignis* durch Auslesen des augenblicklichen Sensorwertes und Anwenden der *Untergrenze*, der *Obergrenze* und der *Hysterese* zur Festlegung der tatsächlichen Untergrenze, Obergrenze und Hysterese. Die verwendeten Formeln für die Kalibrierung hängen vom Sensortyp ab und sind im LEGO SDK erläutert. Die Kalibrierung dauert einen Augenblick (meist etwa 50 ms) – mit `EventState()` kann abgefragt werden, ob die Kalibrierung beendet ist.

```
CalibrateEvent( 2, 50, 50, 20 );
until ( EventState(2) != 5 );       // auf das Ende der Kalibrierung warten
```

SetUpperLimit (*Ereignis*, *Obergrenze*) **Funktion - RCX2, Spy**

Setze die *Obergrenze* für ein *Ereignis*, wobei *Ereignis* eine konstante Zahl und *Obergrenze* ein beliebiger Ausdruck ist.

```
SetUpperLimit( 2, x );         // setze Obergrenze für Ereignis 2 auf x
```

UpperLimit (***Ereignis***) **Wert - RCX2, Spy**

Gib die augenblickliche *Obergrenze* für das *Ereignis* zurück.

```
x = UpperLimit( 2 );          // Obergrenze für Ereignis 2 zurückgegeben
```

SetLowerLimit (***Ereignis***, ***Untergrenze***) **Funktion - RCX2, Spy**

Setze die *Untergrenze* für ein *Ereignis*, wobei *Ereignis* eine konstante Zahl und *Untergrenze* ein beliebiger Ausdruck ist.

```
SetLowerLimit( 2, x );        // setze Untergrenze für Ereignis 2 auf x
```

LowerLimit (***Ereignis***) **Wert - RCX2, Spy**

Gib die augenblickliche *Untergrenze* für das *Ereignis* zurück.

```
x = LowerLimit( 2 );          // Untergrenze für Ereignis 2 zurückgegeben
```

SetHysteresis (***Ereignis***, ***Hysteresewert***) **Funktion - RCX2, Spy**

Setze die *Hysteresewert* für ein *Ereignis*, wobei *Ereignis* eine konstante Zahl und *Hysteresewert* ein beliebiger Ausdruck ist.

```
SetHysteresis( 2, x );
```

Hysteresis (***Ereignis***) **Wert - RCX2, Spy**

Gib die augenblickliche *Hysteresewert* für das *Ereignis* zurück.

```
x = Hysteresis( 2 );
```

SetClickTime (***Ereignis***, ***Klickzeit***) **Funktion - RCX2, Spy**

Setze die *Klickzeit* für ein *Ereignis*, wobei *Ereignis* eine konstante Zahl und *Klickzeit* ein beliebiger Ausdruck ist. Die Zeit wird in Schritte von 10 ms angegeben, sodaß eine Sekunde durch den Wert 100 dargestellt wird.

```
SetClickTime( 2, x );
```

ClickTime (*Ereignis*) **Wert - RCX2, Spy**

Gib die augenblickliche *Klickzeit* für das *Ereignis* zurück.

```
x = ClickTime( 2 );
```

SetClickCounter (*Ereignis*, *Klickzählerwert*) **Funktion - RCX2, Spy**

Setze die *Klickzählerwert* für ein *Ereignis*, wobei *Ereignis* eine konstante Zahl und *Klickzählerwert* ein beliebiger Ausdruck ist.

```
SetClickCounter( 2, x );
```

ClickCounter (*Ereignis*) **Wert - RCX2, Spy**

Gib die augenblickliche *Klickzählerwert* für das *Ereignis* zurück.

```
x = ClickCounter( 2 );
```

3.9.2 Scout-Ereignisse Scout

Der Scout stellt 15 Ereignisse zur Verfügung deren Bedeutung festgelegt ist (siehe nachfolgende Tabelle).

Ereignisname	**Bedingung**
`EVENT_1_PRESSED`	Sensor 1 gedrückt
`EVENT_1_RELEASED`	Sensor 1 losgelassen
`EVENT_2_PRESSED`	Sensor 2 gedrückt
`EVENT_2_RELEASED`	Sensor 2 losgelassen
`EVENT_LIGHT_HIGH`	Lichtsensor *hoch*
`EVENT_LIGHT_NORMAL`	Lichtsensor *normal*
`EVENT_LIGHT_LOW`	Lichtsensor *niedrig*
`EVENT_LIGHT_CLICK`	von *niedrig* über *hoch* zu *niedrig*
`EVENT_LIGHT_DOUBLECLICK`	zwei Klicks
`EVENT_COUNTER_0`	Überlauf Zähler 0
`EVENT_COUNTER_1`	Überlauf Zähler 1
`EVENT_TIMER_0`	Überlauf Timer 0
`EVENT_TIMER_1`	Überlauf Timer 1
`EVENT_TIMER_2`	Überlauf Timer 2
`EVENT_MESSAGE`	neue Nachricht eingetroffen

Die ersten vier Ereignisse werden von Berührungssensoren ausgelöst die an den beiden Sensoreingängen angeschlossen sind. `EVENT_LIGHT_HIGH`, `EVENT_LIGHT_NORMAL` und `EVENT_LIGHT_LOW` werden von einem Lichtsensor ausgelöst der von einem Bereich in eine anderen wechselt.
Die Bereiche werden mit Hilfe von `SetSensorUpperLimit`, `SetSensorLowerLimit` und `SetSensorHysteresis` festgelegt die bereits beschrieben wurden.

`EVENT_LIGHT_CLICK` und `EVENT_LIGHT_DOUBLECLICK` werden ebenfalls vom Lichtsensor ausgelöst. Ein Klick ist ein Zustandswechsel von *niedrig* über *hoch* zurück nach *niedrig* innerhalb eine gewissen Zeit, der *Klickzeit*.

Jeder Zähler hat eine Obergrenze. Wenn ein Zähler die Obergrenze überschreitet werden die Ereignisse `EVENT_COUNTER_0` oder `EVENT_COUNTER_1` ausgelöst. Timer haben auch eine Obergrenze und sie lösen entsprechend die Ereignisse `EVENT_TIMER_0`, `EVENT_TIMER_1` und `EVENT_TIMER_2` aus.

`EVENT_MESSAGE` wird beim Eintreffen jeder neuen Nachricht ausgelöst.

SetSensorClickTime (***Klickzeit***) **Funktion - Scout**

Setze die *Klickzeit* die verwendet wird um Ereignisse des Lichtsensors auszulösen. Werte können in Schritten von 10 ms angegeben werden und dürfen beliebige Ausdrücke sein.

```
SetSensorClickTime( x );
```

SetCounterLimit (***n***, ***Obergrenze***) **Funktion - Scout**

Setze die *Obergrenze* für den Zähler *n*. *n* muß 0 oder 1 sein, *Obergrenze* kann ein beliebiger Ausdruck sein.

```
SetCounterLimit( 0, 100 );     // setze Obergrenze für Zähler 0 auf 100
```

SetTimerLimit (***n***, ***Obergrenze***) **Funktion - Scout**

Setze die *Obergrenze* für den Timer *n*. *n* muß 0 oder 1 sein, *Obergrenze* kann ein beliebiger Ausdruck sein.

```
SetTimerLimit( 1, 100 );       // setze Obergrenze für Timer 1 auf 100
```

3.10 Data Logging RCX

Der RCX enthält einen Datenspeicher, der zum Speichern von Werten der Sensoren, Timer, Variablen und der Systemuhr verwendet werden 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 hochgeladen 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 des Zählers 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 einen Host-Rechner hochgeladen werden. Die Besonderheiten dieser Speicherung hängen von der benutzten NQC-Umgebung ab. In der Kommandozeilenversion sorgen z.B. die beiden folgenden Befehle dafür, daß der Datenspeicher hochgeladen und ausgegeben wird:

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

CreateDatalog (*Größe*) **Funktion - RCX**

Einen Datenspeicher der angegebenen *Größe* (Konstante) anlegen. Die Größe 0 löscht den vorhandenen Datenspeicher ohne einen neuen anzulegen.

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

AddToDatalog (*Wert*) **Funktion - RCX**

Wert im Datenspeicher ablegen. *Wert* kann ein Ausdruck sein. Wenn der Datenspeicher voll ist hat der Aufruf keine Wirkung.

```
AddToDatalog(x);
```

UploadDatalog (*Anfang*, *Anzahl*) **Funktion - RCX**

Upload von *Anzahl* Werten beginnend beim Wert mit der Nummer *Anfang*. Das ist von verhältnismäßig geringem Wert, da üblicherweise der Host-Rechner die Übertragung einleitet.

```
UploadDatalog( 0, 100 );     // alle 100 Datenpunkte hochladen
```

3.11 Allgemeine Funktionen

Wait (*Zeitdauer*) **Funktion - Alle**

Task für eine bestimmte *Zeitdauer* (in 1/100-Sekunden) aussetzen. Die *Zeitdauer* kann eine Konstante oder ein Ausdruck sein.

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

StopAllTasks ( ) **Funktion - Alle**

Alle laufenden Tasks anhalten. Dieser Aufruf hält das gesamte Programm an, so daß der darauffolgende Programmcode wirkungslos bleibt.

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

Random (*n*) **Funktion - Alle**

Gibt eine Zufallszahl zwischen 0 und n zurück. n kann ein beliebiger Ausdruck sein.

```
x = Random( 100 );
```

SetRandomSeed (*n*) **Funktion - RCX2, Spy**

Übergibt dem Zufallszahlengenerator den Initialisierungswert n. n kann ein beliebiger Ausdruck sein.

```
SetRandomSeed( x );          // Initialisierungswert x übergeben
```

SetSleepTime (*Minuten*) **Funktion - Alle**

Setzt die Schlafzeit auf die angegebene Anzahl von *Minuten* (muß eine Konstante sein). Der Wert 0 läßt keinen Schlaf zu..

```
SetSleepTime( 5 );          // nach 5 Minuten schlafen
SetSleepTime( 0 );          // keinen Schlaf zulassen
```

SleepNow ( ) **Funktion - Alle**

Zwingt das Gerät in den Schlafmodus. Wirkt nur bei einer Schlafzeit ungleich Null.

```
SleepNow( );                  // sofort schlafen
```

3.12 Besondere Funktionen des RCX

Program ( ) **Wert - RCX**

Nummer des gerade gewählten Programmes zurückgeben.

```
x = Program( );
```

SelectProgram (*n*) **Wert - RCX2**

Wählt das angegebene Programm aus und startet es. Es ist zu beachten, daß die Programme mit 0 - 4 numeriert sind (die LCD-Anzeige stellt 1 - 5 dar).

```
x = SelectProgram( 3 );
```

BatteryLevel ( ) **Wert - RCX2**

Gibt die Batteriespannung in Millivolt zurück.

```
x = BatteryLevel( );
```

FirmwareVersion ( ) **Wert - RCX2**

Gibt die Firmware-Version als ganze Zahl zurück. Version 3.2.6 wird z.B. zu 326.

```
x = FirmwareVersion( );
```

Watch ( ) **Wert - RCX**

Gibt den Wert der Systemuhr in Minuten zurück.

```
x = Watch( );
```

SetWatch (*Stunden*, *Minuten*) **Funktion - RCX**

Stellt die Systemuhr auf die angegebene Zeit in *Stunden* und *Minuten* ein. *Stunden* muß eine Konstante zwischen 0 und 23 sein (jeweils einschließlich). *Minuten* muß eine Konstante zwischen 0 und 59 sein (jeweils einschließlich).

```
SetWatch( 3, 15 );          // Uhr auf 3:15 einstellen
```

3.13 Besondere Funktionen des Scout

SetScoutRules (*Bewegung*, *Berührung*, *Licht*, *fx*) **Funktion - Scout**

Setze die unterschiedlichen Regeln für den Scout im stand-alone-Modus.

ScoutRules (*n*) **Wert - Scout**

Gibt die augenblicklichen Einstellungen für Regel *n* zurück. *n* sollte eine Konstante zwischen 0 und 4 sein.

```
x = ScoutRules( 1 );          // Einstellungen für Regel 1 lesen
```

SetScoutMode (*Modus*) **Funktion - Scout**

Setzt den Scout in den stand-alone-Modus (0) oder in den power-Modus (1). Als Programmaufruf macht wirklich nur ein Wechsel in den stand-alone-Modus Sinn, da das Gerät bereits im power-Modus sein muß um ein NQC-Programm ausführen zu können.

SetEventFeedback (*Ereignisse*) **Funktion - Scout**

Legt fest welche *Ereignisse* von Klängen begleitet werden.

```
SetEventFeedback( EVENT_1_PRESSED );
```

EventFeedback ( ) **Wert - Scout**

Gibt die Menge der Ereignisse zurück die von Klängen begleitet werden.

```
x = EventFeedback( );
```

SetLight (*Modus*) **Funktion - Scout**

Steuert die LED des Scout. *Modus* muß `LIGHT_ON` oder `LIGHT_OFF` sein.

```
SetLight( LIGHT_ON );              // LED einschalten
```

3.14 Besondere Funktionen des CyberMaster

Der CyberMaster verwendet andere Namen für die Sensoren: `SENSOR_L`, `SENSOR_M` und `SENSOR_R`. Er verwendet außerdem andere Namen für die Ausgänge: `OUT_L`, `OUT_R`, `OUT_X`. Zusätzlich haben die zwei internen Motoren Tachometer die die relative Lage und die Geschwindigkeiten messen, wenn die Motoren laufen. Eine Umdrehung der Welle entspricht etwa 50 Schritten. Die Tachometer können z.B. zum Bau eines Roboters verwendet werden der ohne externen Sensor erkennen kann daß er auf einen Gegenstand aufgefahren ist. Die Tachometer geben als größten Wert 32767 aus und unterscheiden nicht zwischen Drehrichtungen. Sie zählen auch dann weiter, wenn die Welle mit der Hand gedreht wird, selbst dann wenn kein Programm läuft.

Drive (*Motor0*, *Motor1*) **Funktion - CyberMaster**

Schaltet die Motoren mit der angegebenen Leistungsstufe ein. Wenn der Wert negativ ist läuft der Motor rückwärts. Das entspricht dem folgenden Programmcode:

```
SetPower( OUT_L, abs(power0) );
SetPower( OUT_R, abs(power1) );
if(power0 < 0)
  { SetDirection(OUT_L, OUT_REV) }
else
  { SetDirection(OUT_L, OUT_FWD) }
if(power1 < 0)
  { SetDirection(OUT_R, OUT_REV) }
else
  { SetDirection(OUT_R, OUT_FWD) }
SetOutput(OUT_L + OUT_R, OUT_ON);
```

OnWait (***Motoren***, ***n***, ***Zeitdauer***) **Funktion - CyberMaster**

Schaltet die angegebenen *Motoren* ein und wartet die angegebene *Zeitdauer*. Die Motoren haben alle dieselbe Leistungsstufe. Die Zeitdauer wird in 10-tel Sekunden angegeben, der Höchstwert ist 255 (oder 25,5 Sekunden). Das entspricht dem folgenden Programmcode:

```
SetPower( Motoren, abs(power) );
if(power < 0)
  { SetDirection(Motoren, OUT_REV) }
else
  { SetDirection(Motoren, OUT_FWD) }
SetOutput(Motoren, OUT_ON);
Wait( Zeitdauer * 10 );
```

OnWaitDifferent (***Motoren***, ***n0***, ***n1***, ***n2***, ***Zeitdauer***) **Funktion - CyberMaster**

Wie `OnWait()`, es kann jedoch für jeden Motor eine andere Leistungsstufe angegeben werden.

ClearTachoCounter (***Motoren***) **Funktion - CyberMaster**

Setzt die Tachometer für die angegebenen Motoren zurück.

TachoCount (***n***) **Wert - CyberMaster**

Gibt den augenblicklichen Wert des Tachometers für den angegebenen Motor zurück.

TachoSpeed (***n***) **Wert - CyberMaster**

Gibt die augenblicklichen Geschwindigkeit des angegebenen Motors zurück. Bei einem unbelasten Motor ist die Geschwindigkeit ziemlich konstant; der Höchstwert ist 90 (der Wert vermindert sich wenn die Batterieladung nachläßt!). Der Geschwindigkeitswert fällt wenn die Motorbelastung ansteigt. Der Wert 0 zeigt an daß der Motor blockiert ist.

ExternalMotorRunning ( ) **Wert - CyberMaster**

Der Aufruf liefert tatsächlich einMaß für den Strom den der Motor zieht. Die zurückgegebenen Werte neigen dazu leicht zu schwanken, haben aber im Mittel für einen unbelasteten Motor die folgenden Werte:

0	Motor im Freilauf
1	Motor aus
<=7	Motor läuft und hat etwa diese Leistungsstufe. Hier schwanken die Werte am meisten (das liegt wahrscheinlich an der PWM-Steuerung der Motoren). In jedem Fall sollte man wissen, auf welche Leistungsstufe man den Motor ursprünglich gesetzt hat.

Die Werte wachsen mit der Motorbelastung. Ein Wert zwischen 260 und 300 zeigt an, daß der Motor blockiert ist.

AGC ( ) **Wert - CyberMaster**

Gibt den augenblicklichen Wert der automatischen Verstärkung des RF-Empfängers zurück. Das kann als eine sehr grobes (und damit ungenaues) Maß für den Abstand zwischen dem CyberMaster und dem RF-Sender verwendet werden.

4

Technische Einzelheiten

Dieses Kapitel erläutert einige der tiefergehenden Eigenschaften von NQC. Im allgemeinen sollten diese Mechanismen nur als letzter Ausweg dienen, da sie in zukünftigen Versionen geändert werden können. Die meisten Programmierer werden die unten dargestellten Möglichkeiten niemals benötigen – sie werden hauptsächlich zur Erzeugung der NQC-API-Datei verwendet.

4.1 Die `asm`-Anweisung

Die `asm`-Anweisung wird zu Erzeugung fast aller NQC-API-Aufrufe verwendet. Die Syntax lautet:

```
asm { Arg1, Arg2 ... ArgN }
```

Ein *Arg* ist eine der folgenden Möglichkeiten

konstanter Ausdruck
Ausdruck
Ausdruck : Begrenzer

Die Anweisung gibt einfach die Argumentwerte als Bytecode aus. Konstanten sind die einfachsten Argumente – sie ergeben ein einzelnes Byte Rohdaten (die 8 niederwertigsten Bits des Konstantenwertes). Die API-Datei definiert z.B. die folgende inline-Funktion:

```
void ClearMessage() { asm { 0x90 }; }
```

Jedesmal wenn `ClearMessage()` von einem Programm aufgerufen wird, wird der Wert `0x90` als Bytecode ausgegeben.

Viele API-Funktionen haben Argumente und diese Argumente müssen in die entsprechenden effektiven Adressen für den Bytecode-Interpreter umgewandelt werden. Im allgemeinsten Fall besteht eine effektive Adresse aus einem Quellencode gefolgt von einem Wert bestehend aus zwei Bytes (das niederwertigste Byte zuerst). Die Quellencodes werden in der SDK-Dokumentation erläutert, die von LEGO bezogen werden kann. Es ist jedoch öfters wünschenswert den Wert in einer anderen Weise zu codieren – z.B. um nur ein einzelnes Byte hinter dem Quellencode zu verwenden, um den Quellencode ganz wegzulassen oder um nur bestimmte Quellencodes zuzulassen. Um die Bildung der effektiven Adresse zu steuern kann ein *Begrenzer* verwendet werden. Ein Begrenzer ist ein konstanter Wert von 32 Bits. Die 24 niederwertigsten Bits bilden eine Bitmaske die festlegt welche Quellencodes gültig sind (Bit 0 wird gesetzt um die Quelle 0 zuzulassen, usw.). Die höchstwertigen 8 Bits enthalten Formatierungs-Flags für die effektive Adresse. Wenn kein Begrenzer angegeben ist, ist das das gleiche wie die Verwendung des Begrenzers 0 (keine Beschränkung bei den Quellen; dem Quellencode folgen zwei Werte-Bytes). In der API-Datei sind die folgenden Konstanten vereinbart die zur Bildung von Begrenzern verwendet werden können:

```
#define __ASM_SMALL_VALUE  0x01000000
#define __ASM_NO_TYPE      0x02000000
#define __ASM_NO_LOCAL     0x04000000

#if __RCX==2
  // no restriction
  #define __ASM_SRC_BASIC  0
  #define __ASM_SRC_EXT    0
#else
  #define __ASM_SRC_BASIC  0x000005
  #define __ASM_SRC_EXT    0x000015
#endif
```

Das Flag `__ASM_SMALL_VALUE` legt fest, daß ein 1-Byte-Wert anstelle eines 2-Byte-Wertes verwendet werden soll. Das Flag `__ASM_NO_TYPE` legt fest, daß lokale Variablen keine zulässigen Quellen für den Ausdruck sind. Man beachte, daß die RCX2-Firmware weniger einschränkend ist als andere Interpreter, so daß die Definitionen von `__ASM_SRC_BASIC` und `__ASM_SRC_EXT` dort gelockert sind. Die API-Datei für NQC enthält zahlreiche Beispiele für die Verwendung von Beschränkungen bei `asm`-Anweisungen. Bei Verwendung der Kommandozeilenversion von NQC kann die API-Datei mit dem folgenden Befehl ausgegeben werden:

```
nqc -api
```

4.2 Datenquellen

Der Bytecode-Interpreter verwendet unterschiedliche Datenquellen zur Darstellung der verschiedenen Datenarten (Konstanten, Variablen, Zufallszahlen, Sensorwerte, usw.). Die einzelnen Quellen hängen bis zu einem gewissen Grad von dem verwendeten Gerät ab und sind in der SDK-Dokumentation beschrieben, die von LEGO bezogen werden kann.

NQC stellt einen eigenen Operator zur Darstellung von Datenquellen bereit:

`@` *Konstante*

Der Wert dieses Ausdruckes ist die Datenquelle die durch die *Konstante* beschrieben wird. Die 16 niederwertigsten Bits der Konstanten stellen den Datenwert dar, die nächsten 8 Bits geben die Datenquelle an. So ist z.B. der Quellencode für eine Zufallszahl 4, so daß der Ausdruck für eine Zufallszahl zwischen 0 und 9 wie folgt aussieht:

```
@0x40009
```

In der API-Datei sind eine Reihe von Makros definiert die die Verwendung des `@`-Operators klar machen, z.B. für Zufallszahlen:

```
#define    Random(n)    @(0x40000 + (n))
```

Da die Datenquelle 0 dem Speicherplatz der globalen Variablen entspricht, kann dieser globale Speicherbereich mit Hilfe von Nummern adressiert werden: `@0` adressiert den Speicherplatz 0. Wenn man aus irgendeinem Grund kontrollieren muß wo Variablen gespeichert werden, dann sollte man `#pragma reserve` verwenden um NQC anzuweisen, diese Speicherplätze nicht zu verwenden. Man kann dann selbst mit dem `@`-Operator darauf zugreifen. So resrviert z.B. der folgende Codeschnippsel den Speicherplatz 0 und erzeugt dafür ein Makro `x`.

```
#pragma    reserve    0
#define    x          (@0)
```

Wegen der Art und Weise in der Sensoren implementiert sind, ist es bei der Verwendung von Makros wie etwa `SensorValueRaw()` erforderlich, die Sensordatenquelle in einen Sensorindex umzuwandeln. Dazu kann der `__sensor`-Ausdruck verwendet werden:

```
#define  SensorValueRaw(n)  @(0xc0000 + (__sensor(n)))
```

Index