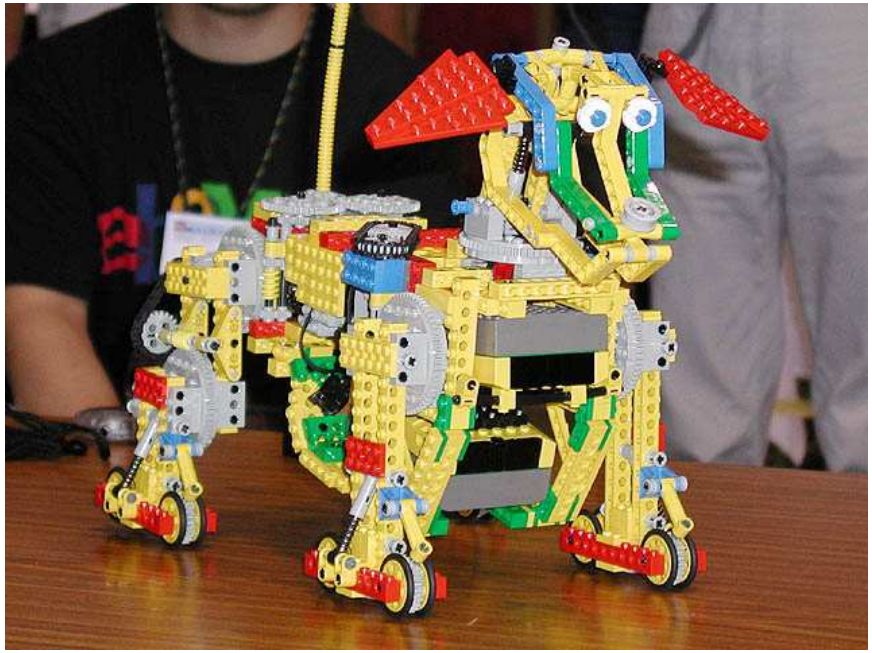
technische universität
dortmund

fakultät für informatik
informatik 12

Programmierung Eingebetteter Systeme

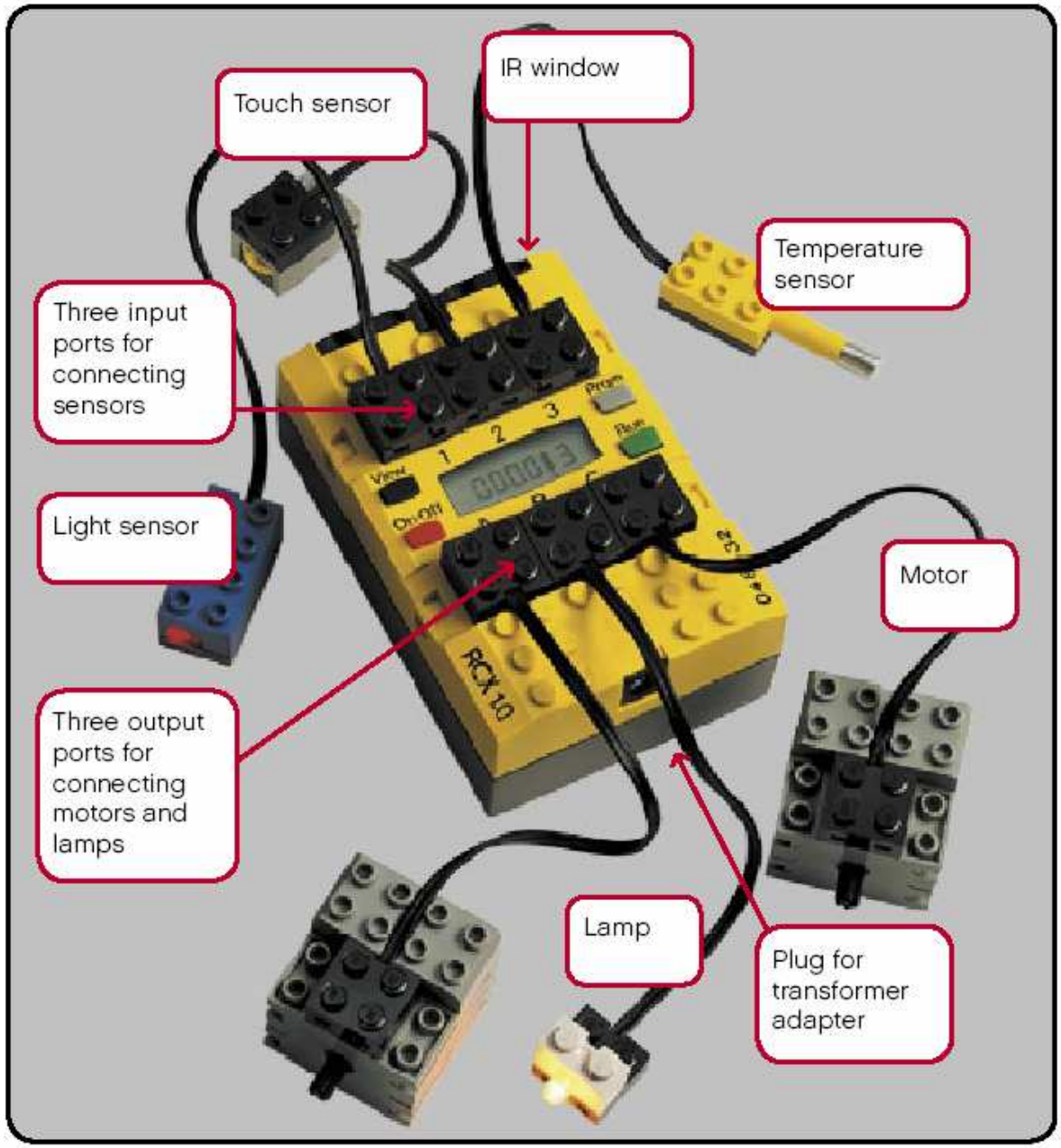
Microcontroller and
Lego® Mindstorms
im HAPRA



Peter Marwedel, Karl-Heinz Temme
TU Dortmund, Informatik 12

www.watch.impress.co.jp/.../20000821/minds.htm

RCX, die Lego® Kontrolleinheit



Wesentliche Eigenschaften des Roverbots

- Elektromotoren des Roverbots

71427	Dreh-moment	Rotations-Geschwin-digkeit	Strom-stärke	Mechani-sche Leistung	Elektri-sche Leistung	Effi-zienz
4.5 V	2.25 N.cm	57 rpm	0.12 A	0.13 W	0.54 W	24 %
7 V	2.25 N.cm	160 rpm	0.12 A	0.38 W	0.85 W	45 %
9 V	2.25 N.cm	250 rpm	0.12 A	0.58 W	1.1 W	54 %
12 V	2.25 N cm	375 rpm	0.12 A	0.88W	1.5 W	61 %

Wesentliche Eigenschaften des Roverbots

- Sensoren

Konfiguration: Sensortyp und Sensormodus

Der Sensortyp bestimmt, wie das RCX mit dem Sensor interagiert. Der Sensormodus sagt dem RCX, wie die Werte eines Sensors zu interpretieren sind. Zu jedem Sensor gehören drei separate Werte:

1. Rohwert,
2. logischer Wert oder auch boolescher Wert und
3. aufbereiteter Wert.

Wesentliche Eigenschaften des Roverbots

- Sensoren

Rohwert: eine diskrete Zahl zwischen 0 und 1023.

Alle drei Millisekunden liest das RCX diesen Rohwert vom Sensor aus und konvertiert ihn dann in einen logischen oder aufbereiteten Wert.

Logischer Wert: 0 oder 1.

Rohwert > 562: 0

Rohwert < 460: 1

460 ≤ Rohwert ≤ 562: unverändert

Aufbereiteter Wert: abhängig vom Sensortyp

Wesentliche Eigenschaften des Roverbots

- Sensoren

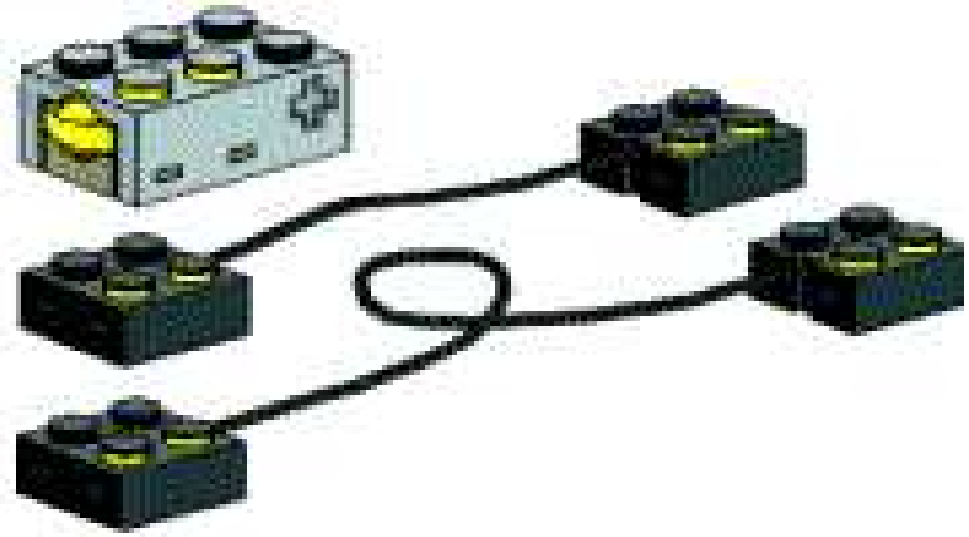
Passive und aktive Sensoren.

Ein passiver Sensor besteht aus einem Widerstand. Alle 3ms wird eine Widerstandsmessung (10k) mit einer Spannung von 5V ausgeführt. 0 V = 0, 5 V = 1023.

Bei aktiven Sensoren wird zusätzlich 8V Spannung periodisch über einen Transistor zugeschaltet. Der Sensor wird 3ms mit Spannung versorgt, gefolgt von einer 0,1ms langen Unterbrechung. Diese Zeit dient dazu, um den aktuellen Sensorwert auslesen zu können.

Berührungssensor

Es können mehrere Berührungssensoren parallel angeschlossen werden, Auswertung: ODER.

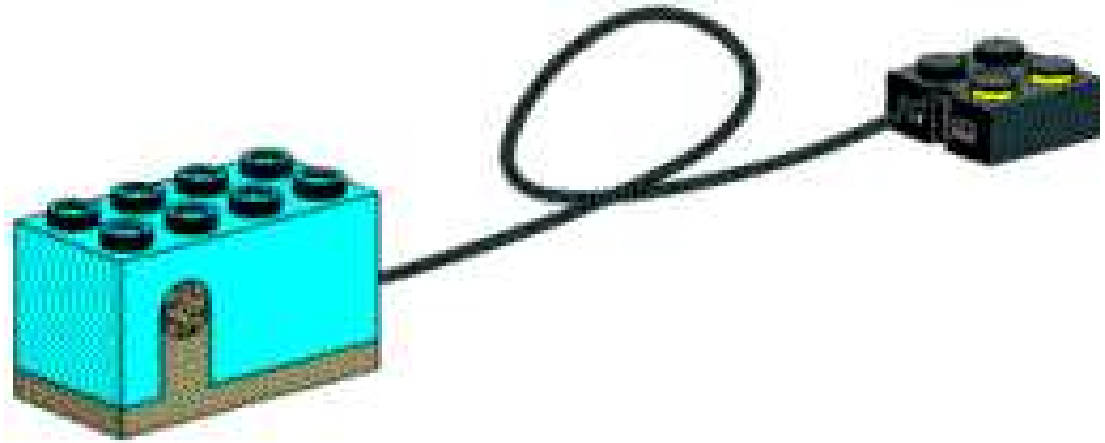


Lichtsensor

Der Lichtsensor besteht aus einer roten LED und einem Fototransistor. Die eigene Lichtquelle bzw. die LED ermöglicht ihm das von einer nahe gelegenen Oberfläche reflektierte Licht zu messen.

Rotationssensor

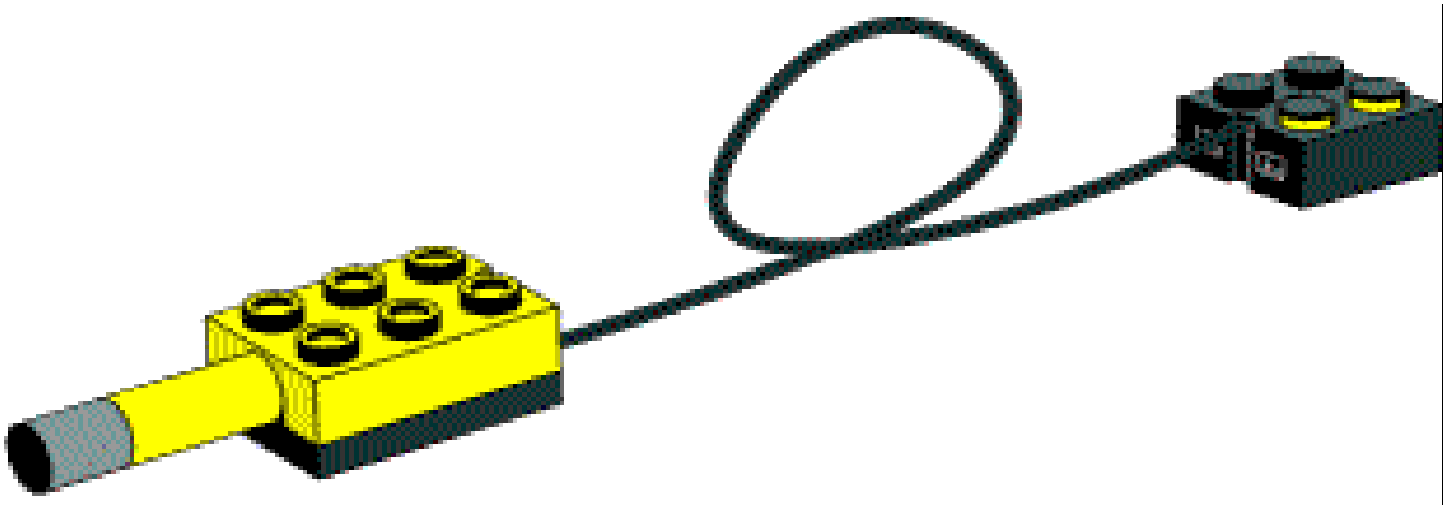
Der Rotationssensor stellt einen aktiven Sensor dar. Er misst die relative Drehung. Die Messung findet in Schritten von 22,5 Grad statt. Eine volle Drehung entspricht einem Sensorwert von 16.



Temperatursensor

Die Temperatur kann in Grad Celsius oder Grad Fahrenheit gemessen werden. Der Betriebsbereich des Sensors beträgt -20 ℃ bis +70 ℃ bzw. -3 F bis +157 F.

Reaktionzeit: bei direktem Kontakt im Sekunden-, sonst im Minutenbereich (pro Grad Differenz).



Sensormodi

- **Rohmodus**
  Im Rohmodus ist der verarbeitete Wert eines Sensors immer gleich seinem Rohwert: eine Zahl zwischen 0 und 1023.
- **Logischer Modus**
  In diesem Modus wird der verarbeitete Wert des Sensors in einen logischen (0 oder 1) konvertiert. Dies ist der Standardmodus für einen Berührungssensor.
- **Flankenzählmodus**
  Im Flankenzählmodus zählt das RCX, wie oft sich der logische Wert ändert. Der Zähler beginnt mit der Konstanten 0 und wird jedes Mal um 1 inkrementiert, wenn sich der logische Wert von 0 auf 1 oder von 1 auf 0 ändert.

Sensormodi

- **Impulszählungsmodus**

Der Impulszählungsmodus ist dem Flankenzählmodus ähnlich. Allerdings wird in diesem Modus der Zähler nur inkrementiert, wenn sich ein logischer Wert von 1 auf 0 ändert. Wie schon bei dem Flankenzählungsmodus wird auch hier der logische Wert entprellt, um Störsignale zu beseitigen.

Sensormodi

- **Prozentmodus**
  In diesem Modus wird der Rohwert in einen Wert zwischen 0 und 100 konvertiert. Höhere Rohwerte entsprechen einem niedrigeren Prozentsatz, d.h. der Rohwert 0 entspricht dem Prozentwert 100 und der Rohwert 1023 entspricht dem Prozentwert 0. Dies ist der Standardmodus für einen Lichtsensor.
- **Rotationsmodus**
  Der Ergebniswert in diesem Modus ist eine kumulative Drehung in Schritten von 22,5 Grad. Das entspricht der Konstanten 16, die für eine volle Drehung steht.

Sensormodi

- **Celsius- und Fahrenheitmodi**

Um den Rohwert in einen Temperaturwert zu konvertieren, benutzt der Celsiusmodus eine spezielle Funktion. Diese Funktion kompensiert die speziellen Eigenschaften des Temperatursensors. Analog benutzt der Fahrenheitmodus dieselbe Funktion wie der Celsiusmodus, um einen Rohwert in eine Temperatur zu konvertieren.

Kommunikation

Die RCX-Einheit kann mit einem Computer oder einer anderen RCX-Einheit kommunizieren.

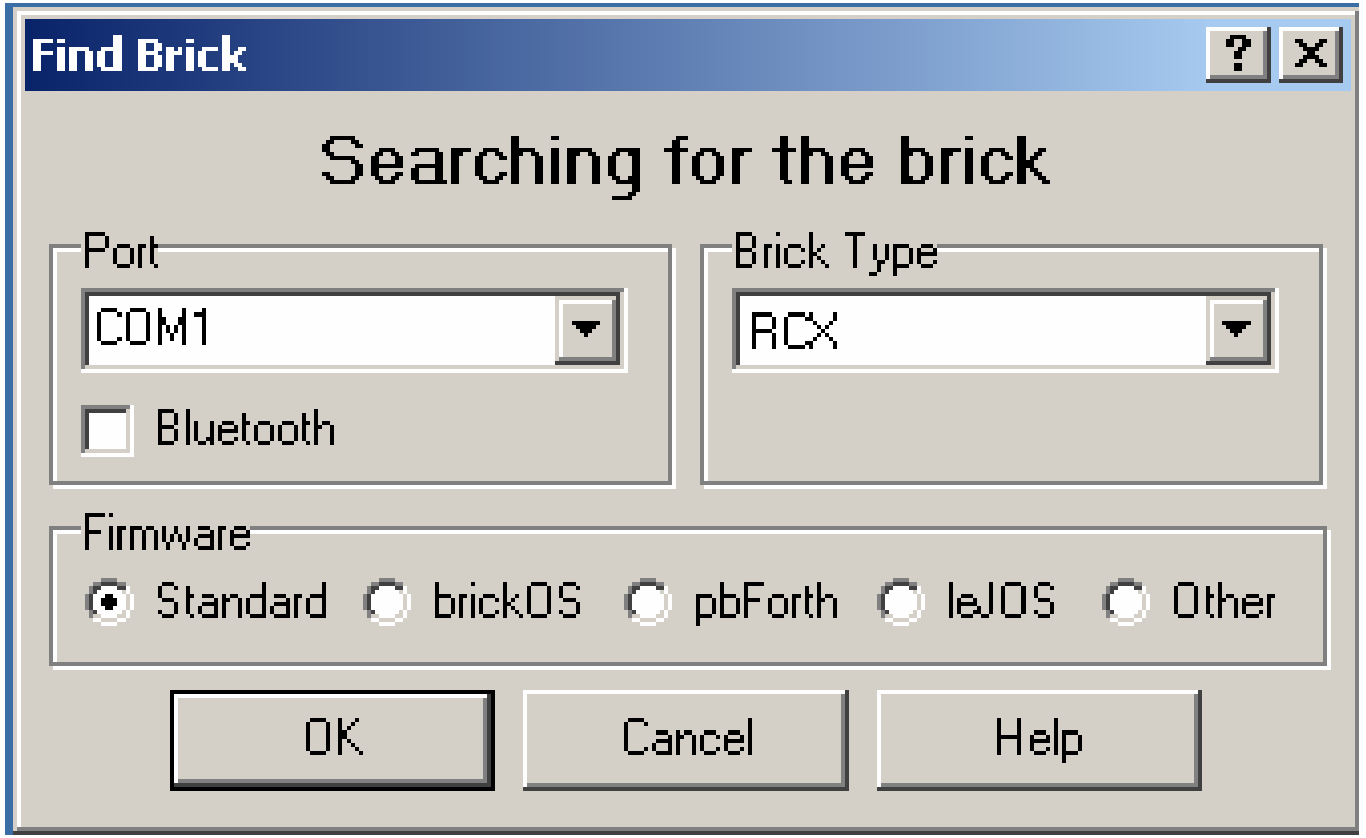
Infrarotschnittstelle bzw. IR-Tower, Senden und Empfangen.

Reichweite: 1-2 Meter

Stärke: „schwach“.

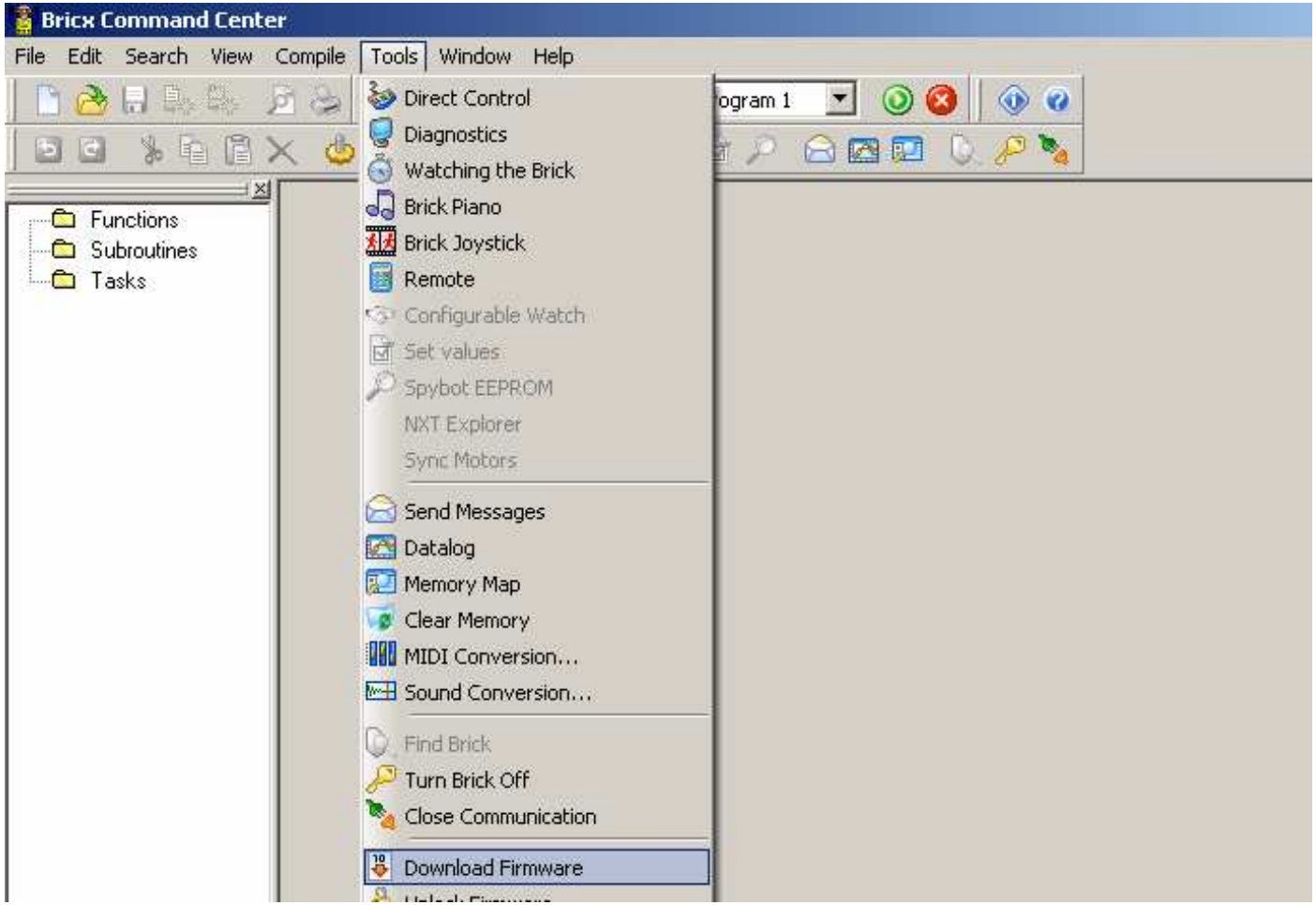
Bricx Command Center

- **Anmeldung / Identifikation**



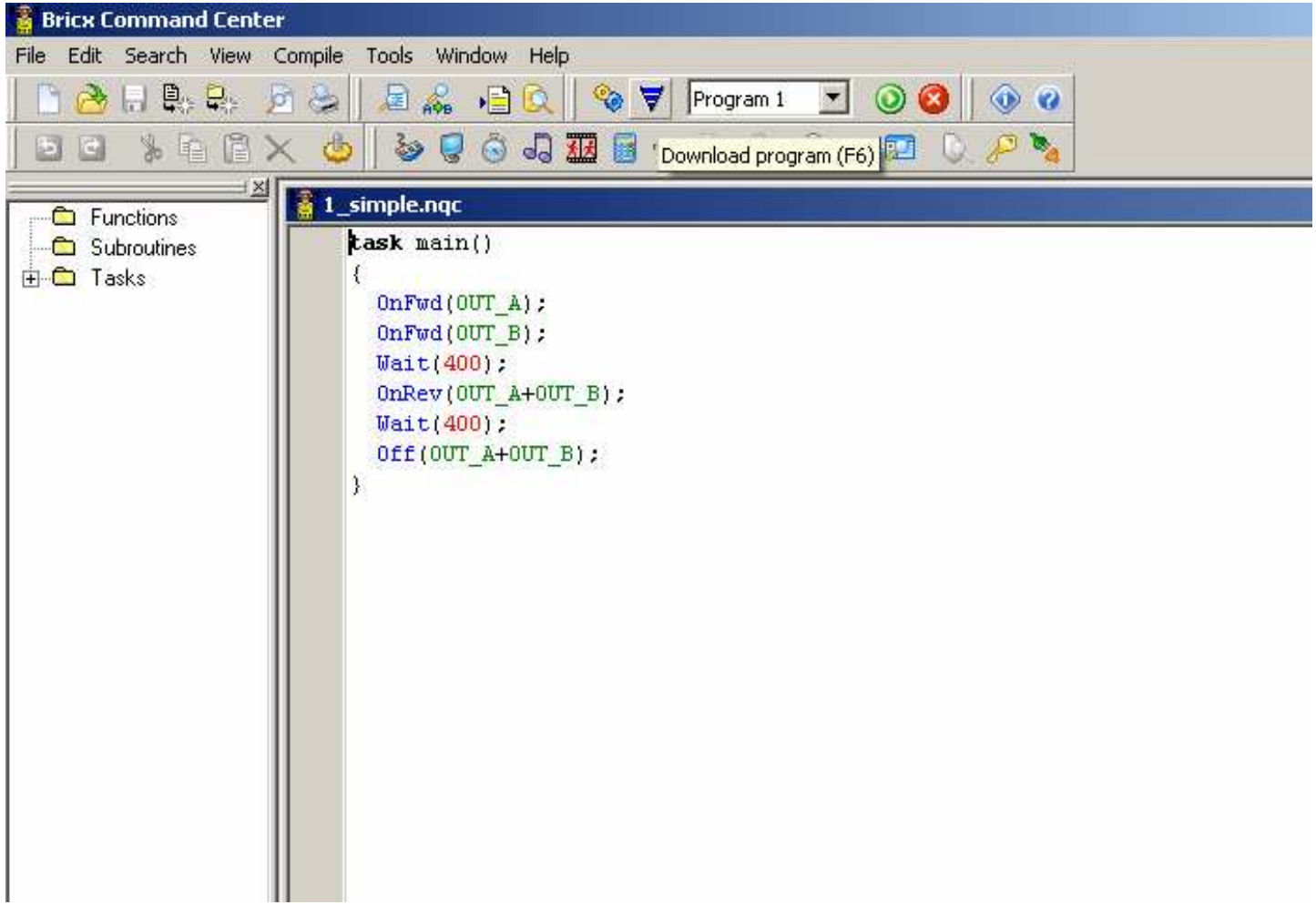
Bricx Command Center

- **Download der Firmware**



Bricx Command Center

- **Laden eines Programms**



Die Programmiersprache NQC

- **Bezeichner**
  Die Namen von Variablen, Funktionen, Subroutinen und Task werden Bezeichner genannt. In NQC muss ein gültiger Bezeichner mit einem Klein- oder Grossbuchstaben oder einen Unterstrich "_" beginnen. Die nachfolgenden Zeichen können Ziffern, Buchstaben oder Unterstriche sein.

- **Schlüsselwörter**

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

Programmstruktur

- **Codeblöcke**
  Ein Block oder Codeblock ist ein Bereich, der durch geschweifte Klammern { } zusammengefasst wird. Innerhalb dieses Blocks haben wir aufeinander folgende Anweisungen. In NQC können wir zwischen drei verschiedenen Codeblocks unterscheiden, nämlich den Tasks, den Funktionen und den Subroutinen.

Programmstruktur

- **Tasks**
  Jedes NQC Programm muss mindestens die Task *main()* enthalten. Das RCX ist aber im Stande, bis zu maximal 10 Tasks gleichzeitig auszuführen. Wir definieren Tasks durch das Schlüsselwort task und einen gültigen Bezeichner *name* mit folgender Syntax:

task *name()*
{
// Taskcode
}

Task können gestartet und gestoppt werden durch die Schlüsselwörter start und stop. Mit dem Befehl StopAllTasks können alle laufenden Tasks gestoppt werden.

Programmstruktur

- **Funktionen**
  In NQC sind alle Funktionen vom Typ void, dass heisst, das keine Rückgabewerte für Funktionen unterstützt werden. Es dürfen kein, ein oder mehrere Argumente der Funktion übergeben werden. Die einzelnen Argumente werden mittels Kommata getrennt. Die folgende Tabelle zeigt die möglichen Argumente, die Funktionen in NQC übergeben werden können.

Typ	Bedeutung	Restriktionen
int	Wert	keine
const int	Wert	nur Konstanten
int &	Referenz	nur Variablen
const int &	Referenz	Funktion darf das Argument nicht modifizieren
int*	Zeiger	Nur Zeiger
const int *	Zeiger	Funktion darf den Zeiger nicht modifizieren

Programmstruktur

- **Funktionen**
  Die Syntax für Funktionen sieht wie folgt aus:

void *name(Argumente)*
{
// Funktionsrumpf
}

Alle Funktionen in NQC sind implizit “inline”-Funktionen. Das heisst, dass bei jedem Aufruf einer Funktion der Compiler den kompletten Funktionscode in den Code einfügt. Der Vorteil liegt in der Ausführungszeit, der Nachteil im Platzverbrauch.

Programmstruktur

- **Subroutinen**
  Subroutinen erstellen im Gegensatz zu Funktionen nur eine Codeinstanz unabhängig von der Anzahl der Aufrufe. Damit sind diese viel effizienter im Bezug auf den Speicherverbrauch als Funktionen, haben aber – aufgrund des LEGO-Bytecode-Interpreters – viele Einschränkungen:

- Das RCX kann maximal nur bis zu 8 Subroutinen unterstützen.
- Subroutinen können keine Argumente annehmen.
- Subroutinen können keine anderen Subroutinen aufrufen.
- In der RCX-Version 1.0, wenn Subroutinen aus verschiedenen Task heraus aufgerufen werden, dürfen diese keine lokalen Variablen besitzen und keine Berechnungen auf temporären Variablen durchführen.

Programmstruktur

- **Makros**
  Eine andere Möglichkeit, kurze, oft benutzte Codefragmente auszulagern, bieten die so genannten Makros. Makros lassen sich mit dem Schlüsselwort #define definieren. Ein Makro endet normalerweise mit dem Ende einer Zeile. Dies lässt sich jedoch mit einem Backslash „\“ umgehen. Der Benutzer kann eine beliebige Anzahl Makros definieren. Makros können in ihrer Signatur auch Parameter enthalten. Das Schlüsselwort #undef kann dazu benutzt werden, die Definition eines Makros zu entfernen.

Beispiel:

```
#define turn_around(time) OnRev(OUT_C); Wait(time); \
                          OnFwd(OUT_A+OUT_C);
```

Programmstruktur

- **Variablen**
  Variablen werden mit dem Schlüsselwort int deklariert und können später oder auch in derselben Zeile mithilfe des Zuweisungsoperators „=“ initialisiert werden. Durch die Deklaration wird die Reservierung des nötigen Speichers veranlasst. Das reicht jedoch nicht aus. Die deklarierte Variable muss noch initialisiert werden, denn die Variable ist zwar nach der Deklaration dem Compiler bekannt, hat aber noch kein Startwert, mit dem man rechnen kann.

```
int x;          // x wird deklariert
int x,y;        // x und y werden deklariert
int x = 1;      // x wird initialisiert mit 1
int x = 1,y;    // x wird initialisiert mit 1 und y wird deklariert
```

Programmstruktur

- **Variablen**
  Wir unterscheiden zwischen globalen und lokalen Variablen. Globale Variablen werden außerhalb jeglicher Codeblöcke deklariert und können in allen Funktionen, Subroutinen und Task benutzt werden. Der Gültigkeitsbereich lokaler Variablen umfasst jeweils nur einen Codeblock – in welchem die Variable deklariert worden ist.

```
int x; // x ist eine globale Variable
task main()
{
int y; // y ist lokal und bezieht sich nur auf den Block task
x = y; // ok
{
int z; // z ist eine lokale Variable
y = z; // ok
}
y = z; // Fehler, z ist nicht sichtbar außerhalb seines Blocks
}
```

Programmstruktur

- **Zuweisungen**
  Es gibt außer dem Gleichheitsoperator „=“ noch andere Operatoren mit denen wir Variablen Werte zuweisen können.

Operator	Bedeutung
=	Zuweisung
+=	Zuweisung mit Addition
-=	Zuweisung mit Subtraktion
*=	Zuweisung mit Multiplikation
/=	Zuweisung mit Division
%=	Zuweisung des Restes der Division (modulo)
&=	Zuweisung mit bitweisem UND
|=	Zuweisung mit bitweisem ODER
^=	Zuweisung mit bitweisem XOR
||=	Zuweisung des Betrags
+-=	Zuweisung mit Signumfunktion
>>=	Zuweisung mit Rechtsschieben um konstanten Faktor
<<=	Zuweisung mit Linkschieben um konstanten Faktor

Programmstruktur

- **IF-Anweisung**
  Die If-Anweisung besteht aus dem Schlüsselwort if, einer Bedingung und einer nachfolgenden Anweisung, deren Ausführung von dem logischen Wert der Bedingung abhängt. Ist die Bedingung erfüllt, folgt die Abarbeitung der danach stehenden Anweisung. Ist die Bedingung nicht erfüllt, wird die alternative Anweisung bearbeitet. Der alternativen Anweisung muss immer das Schlüsselwort else vorausgehen, sonst wird die nächste Zeile des Programms ausgeführt. Die Syntax für die if-Anweisung sieht wie folgt aus:

if (*Bedingung) Anweisung* // Konsequenz der erfüllten Bedingung

if (*Bedingung) Anweisung* else *Anweisung* // Alternative

Programmstruktur

- **WHILE-Schleife**
  Die While-Schleife besteht aus einer Bedingung, die bei jedem Schleifendruchgang überprüft wird. Ist die Bedingung wahr, werden die Anweisungen, die im Schleifenrumpf stehen, abgearbeitet. Falls die Bedingung nicht erfüllt ist, wird im Programm mit der nächsten Anweisung nach der While-Schleife verfahren. Es ist von der Logik her möglich, dass die Bedingung der While-Schleife nie oder auch immer erfüllt ist. Es ist auch möglich, aus der Schleife mit dem Schlüsselwort break auszubrechen.

```
while (x<10)
{
   x = x-1;        // Rumpf, kann auch einen Codeblock enthalten
}
```

Programmstruktur

- **DO-WHILE-Schleife**
  Die Do-While-Schleife ist eine Variante der While-Schleife und unterscheidet sich von dieser in der Folge der Auswertung der Bedingung. Die While-Schleife prüft erst die Bedingung, die Do-While-Schleife tut dies erst nach der Abarbeitung des Rumpfes. Die Do-While-Schleife wird also, im Gegensatz zu While, mindestens ein Mal ausgeführt.

```
do
{
   x = x-1;       // Rumpf
}
while (x<10)
```

Programmstruktur

- **FOR-Schleife**
  Bei der For-Schleife wird genau so wie im Falle der While-Schleife der Rumpf erst abgearbeitet, wenn die Bedingung erfüllt ist. Die Syntax sieht wie folgt aus:

for(*Ausdruck1* ; *Bedingung* ; *Ausdruck2*) *Rumpf*

```
for (int i=0; i<10 ; i = i+1;)
{
  j = j +1 ; // j wird zehn Mal inkrementiert
}
```

Der erste Ausdruck wird einmal ausgeführt, dann folgt die Auswertung der Bedingung. Ist diese wahr, wird der Rumpf abgearbeitet und anschließend der zweite Ausdruck. Dies wird so lange wiederholt, bis die Bedingung falsch ist.

Programmstruktur

- **REPEAT-Schleife**
  Die Repeat-Schleife führt den Schleifenrumpf n-Mal aus. Die Syntax sieht wie folgt aus:

repeat (*Ausdruck*) *Rumpf*

```
Repeat (4)
{
 x = x +1; // wird vier Mal ausgeführt
}
```

Programmstruktur

- **SWITCH-Anweisung**
  Die Switch-Anweisung ermöglicht es, abhängig von einem Ausdruck einen bestimmten Codeblock auszuführen. Jedem Codeblock geht das Schlüsselwort case voraus. Dem Case-Label muss eine Zahl folgen, die für kein anderes Label verwendet worden ist. Die Switch-Anweisung wertet den Ausdruck aus und vergleicht das Ergebnis mit den Case-Labels. Entspricht das Ergebnis einer Zahl, die in einem der Labels vorkommt, wird der dem Label folgende Codeblock ausgeführt. Das Auswertungsverfahren endet wenn das Schlüsselwort break oder das Ende der Switch-Anweisung erreicht wird. Es ist auch möglich, ein Default-Case-Label zu definieren. Dieser wird immer dann ausgeführt, wenn kein passender Case-Label gefunden worden ist. Die Switch-Anweisung folgt der folgenden Syntax (Beispiel):

Programmstruktur

- **SWITCH-Anweisung**

```
switch(n)
{
case 1: // ausführen wenn n = 1 ist
        break;
case 2: // ausführen wenn n = 2 ist
        break;
case 3:
case 4: // ausführen wenn n = 3 oder n = 4 ist
        break;
default:// ausführen wenn n nicht 1, 2, 3 oder 4 ist
            break;
}
```

Programmstruktur

- **Ausdrücke und Operatoren**
  Der primitivste Ausdruck in NQC ist ein Wert. Unter Werten werden sowohl Variablen als auch numerische Konstanten verstanden. Numerische Konstanten können als Dezimalzahl angegeben werden (z.B. 215) oder als eine Hexadezimalzahl (z.B 0xD7). Komplexere Ausdrücke setzen sich aus Werten und Operatoren zusammen. Es gibt zwei vordefinierte Werte, nämlich true und false, wobei der Wert von true 1 ist und der Wert von false ungleich 0 ist.

Die folgende Tabelle listet, der Rangordnung nach (höchste bis niedrigste), die NQC-Operatoren auf.

Programmstruktur

- **Operatoren**

abs() sign()	Betragfunktion Signumfunktion		abs(x) sign(x)
++, --	In(de)krementieren	nur Variablen	x++
- ~ !	unäres Minus bitweises Komplement logische Komplement		-x ~123 !x
*, /, %	Multiplikation, Division, Modulo		x*y
+, -	Addition und Substraktion		x+y
<<, >>	Linksverschiebung, Rechtsverschiebung	Faktor muss konstant sein	x << 4

Programmstruktur

- **Operatoren**

<, >, <=, >=	numerische Vergleiche		x<y
==, !=	Gleichheit, Ungleichheit		x==1
&	bitweises UND		x&y
^	bitweises XOR		x^y
|	bitweises ODER		x|y
&&	logisches UND		x&&y
||	logisches ODER		x||y
?:	Bedingungsoperator		x==1 ? y:z

Programmstruktur

- **Bedingungen**

`True, False`	`immer wahr / immer falsch`
`Ausdruck`	`wahr wenn Ausdruck nicht gleich Null ist`
`ausdruck1 == ausdruck2`	`wahr wenn ausdruck1 gleich ausdruck2 ist`
`ausdruck1 != ausdruck2`	`wahr wenn ausdruck1 und ausdruck2 ungleich sind`
`ausdruck1 < ausdruck2` `ausdruck1 <= ausdruck2`	`wahr wenn ausdruck1 kleiner (gleich) als ausdruck2 ist`
`ausdruck1 > ausdruck2` `Ausdruck1 >= ausdruck2`	`wahr wenn ausdruck1 grösser (gleich) als ausdruck2 ist`
`! bedingung`	`wahr wenn bedingung falsch ist`
`bedingung1 && bedingung2`	`logische Und-Verknüpfung von zwei Bedingungen`
`bedingung1 || bedingung2`	`logische Oder-Verknüpfung von zwei Bedingungen`

NQC- Programmierschnittstelle (API)

Die API setzt sich aus Konstanten, Funktionen und Werten zusammen, mithilfe derer die Ressourcen von RCX angesprochen und benutzt werden können. Sie basiert auf der API aus „NQC Programmer's Guide".[4]

- **Sensoren**
  Die RCX-Einheit verfügt über drei Sensoranschlüsse, die intern mit 0, 1 und 2 bezeichnet sind. Damit es nicht zu Verwirrungen kommt, da auf der RCX-Einheit die Sensoren mit 1, 2 und 3 benannt sind, bietet NQC drei Konstanten an, SENSOR_1, SENSOR_2 und SENSOR_3. Wollen wir den Wert des intern mit 0 markierten Sensors auslesen, reicht es den folgenden Code zu schreiben:

```
x = SENSOR_1;
```

NQC- Programmierschnittstelle (API)

- **Sensortypen und –modi**
  In NQC ist es die Aufgabe des Programmierers, dem RCX mitzuteilen, welche Art von Sensor an welchen RCX-Port angeschlossen ist. Um erwartete Werte zu erhalten, muss der angeschlossene Sensor seiner Funktionalität entsprechend deklariert werden. Es gibt fünf Sensortypen die mit dem Funktionsaufruf SetSensorType deklariert werden können.

`SENSOR_TYPE_NONE`	`generischer passiver Sensor`
`SENSOR_TYPE_TOUCH`	`Berührungssensor`
`SENSOR_TYPE_TEMPERATURE`	`Temperatursensor`
`SENSOR_TYPE_LIGHT`	`Lichtsensor`
`SENSOR_TYPE_ROTATION`	`Rotationssensor`

NQC- Programmierschnittstelle (API)

- **Sensortypen und –modi**
  Für jeden Sensor muss auch sein Modus definiert werden. Ein Sensormodus gibt der RCX-Einheit vor, wie die RAW-Werte des Sensors bearbeitet werden sollen. Man kann den Modus des Sensors durch den Aufruf der Funktion SetSensorMode festlegen.

SENSOR_MODE_RAW	RAW-Werte von 0 bis 1023
SENSOR_MODE_BOOL	boolesche Werte 0 oder 1
SENSOR_MODE_EDGE	zählt die Anzahl der Flanken
SENSOR_MODE_PULSE	zählt die booleschen Perioden
SENSOR_MODE_PERCENT	Werte von 0 bis 100
SENSOR_MODE_FAHRENHEIT	Grad Fahrenheit
SENSOR_MODE_CELSIUS	Grad Celsius
SENSOR_MODE_ROTATION	Rotationsimpulse (16 mal pro Umdrehung)

NQC- Programmierschnittstelle (API)

- **Sensortypen und –modi**
  Eine andere Möglichkeit, den Sensortyp und den Sensormodus zu definieren, bietet die Funktion SetSensor. Sie ermöglicht es, mit nur einem Funktionsaufruf, einen gegebenen Sensor standardmäßig zu initialisieren.

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

NQC- Programmierschnittstelle (API)

- **Sensortypen und –modi, Zusammenfassung**

- **SetSensor(sensor, configuration)**

Setzt den Typ und den Modus des Sensors fest.
SetSensor(SENSOR_1, SENSOR_TOUCH);

- **SetSensorType(sensor, type)**

Setzt den Sensortyp fest.
SetSensorType(SENSOR_1, SENSOR_TYPE_TOUCH);

- **SetSensorMode(sensor, mode)**

Setzt den Sensormodus fest. Zusätzlich kann zum Modus ein Steigungsparameter addiert werden.
SetSensorMode(SENSOR_1, SENSOR_MODE_RAW);
SetSensorMode(SENSOR_1, SENSOR_MODE_RAW + 10);

- **ClearSensor(sensor)**

Löscht den Wert eines Sensors.
ClearSensor(SENSOR_1);

NQC- Programmierschnittstelle (API)

- **Sensorinformationen**

- **SensorValue(n)** Gibt den aktuell gelesenen Wert für Sensor n zurück, wobei n 0, 1 oder 2 sein muss.
  x = SensorValue(0); // lese Sensor 1
- **SensorType(n)** Gibt den Sensortyp für n zurück.
  x = SensorType(0);
- **SensorMode(n)** Gibt die Sensormodus für n zurück.
  x = SensorMode(0);
- **SensorValueBool(n)** Gibt den booleschen Sensorwert für n zurück.
  x = SensorValueBool(0);
- **SensorValueRaw(n)** Gibt den „rohen“ Sensorwert (0 – 1023) für n zurück.
  x = SensorValueRaw(0);

NQC- Programmierschnittstelle (API)

- **Ausgänge**

Jede Funktion, die die Ausgänge des RCX anspricht, erwartet als erstes Argument eine Konstante, welche einen der drei möglichen Ausgänge bestimmt. Die Konstanten heissen OUT_A, OUT_B und OUT_C. Um mehrere Ausgänge gleichzeitig anzusprechen, kann man diese mit Hilfe von "+" zusammenfassen, z.B. OUT_A + OUT_B. Drei Eigenschaften der Ausgänge können verändert werden, der Modus mit der Funktion SetOutput(), die Richtung mit SetDirection() und die Kraft mithilfe von SetPower(). Die folgenden Tabellen listen die möglichen Argumente auf.

NQC- Programmierschnittstelle (API)

- **Ausgänge**

OUT_OFF	Der Ausgang ist ausgeschaltet. Der Motor wird daran gehindert sich zu drehen.
OUT_ON	Der Ausgang ist angeschaltet. Der Motor wird angetrieben.
OUT_FLOAT	Der Motor kann ausrollen.

OUT_FWD	Vorwärtsrichtung.
OUT_REV	Rückwärtsrichtung.
OUT_TOGGLE	entgegengesetzte Richtung.

OUT_LOW	Minimale Motorkraft.
OUT_HALF	Normale Motorkraft.
OUT_FULL	Maximale Motorkraft

NQC- Programmierschnittstelle (API)

- **Ausgänge**

- **SetOutput(outputs, mode) // Bestimmt den Modus des Ausgangs.**
- **SetOutput(OUT_A + OUT_B, OUT_ON); // Schaltet A und B an**
- **SetDirection(outputs, direction) // Bestimmt die Drehrichtung der Motoren.**
- **SetDirection(OUT_A, OUT_REV); // A soll rückwerts drehen.**
- **SetPower(outputs, power) // Bestimmt die Motorkraft. Ausser den Konstanten sind auch Zahlen zwischen 0 und 7 zugelassen.**
- **SetPower(OUT_A, OUT_FULL); // volle Kraft.**
- **OutputStatus(n) // Gibt den aktuellen Status für Motor n zurück.**

NQC- Programmierschnittstelle (API)

- **Sound**

- **PlaySound(sound)**
  Spielt einen der sechs vordefinierten Klängen ab. Das Argument muss eins aus den folgenden, vordefinierten Konstanten sein:
- **SOUND_CLICK, SOUND_DOUBLE_BEEP, SOUND_DOWN, SOUND_UP, SOUND_LOW_BEEP, SOUND_FAST_UP.**
- **PlayTone(frequency, duration)**
  Spielt einen Ton ab mit der gegebenen Frequenz und Länge. Die Frequenz wird in Hertz angegeben und die Länge in hundertstel Sekunden.
- **PlayTone(440, 50); // Spielt 'A' für eine halbe Sekunde ab**

NQC- Programmierschnittstelle (API)

- **LCD-Anzeige**

Die RCX-Einheit hat sechs verschiedene Anzeigeoptionen. Standardmässig ist die RCX-Anzeige auf Systeminformationen eingestellt.

`DISPLAY_WATCH`	`Systeminformationen (der Einheit)`
`DISPLAY_SENSOR_1 (2, 3)`	`Wert von Sensor 1 (2, 3)`
`DISPLAY_OUT_A (B, C)`	`Einstellungen für Ausgang A (B, C)`

- **SelectDisplay(mode)**

Wählt den Anzeigemodus aus.
SelectDisplay(DISPLAY_SENSOR_1); // zeige Sensor 1 an

NQC- Programmierschnittstelle (API)

- **Kommunikation**

  Die RCX-Einheit kann durch die Infrarotschnittstelle Informationen senden und empfangen. Nachrichten können die Werte von 0 bis 255 haben, wobei der Wert 0 für „keine Nachricht wurde empfangen“ reserviert ist. Da Kommunikation durch die Infrarotschnittstelle stattfinden, können keine Informationen während des Sendevorgangs, empfangen werden.

- **Message()**
  Mit diesem Befehl kann die zuletzt empfangene Nachricht angesprochen werden. Falls keine Nachricht empfangen wurde, wird 0 zurückgeliefert.

NQC- Programmierschnittstelle (API)

- **Kommunikation**

- **ClearMessage() // Löscht den Nachrichtenspeicher. Der Befehl erleichtert das Abfangen einer neuen Nachricht, weil das Programm jetzt auf das Ergebnis des Aufrufs von Message() warten kann.**
- **ClearMessage(); // lösche den Nachrichtenspeicher**
- **until(Message() > 0); // warte auf die nächste Nachricht**
- **SendMessage(message) // Sendet eine Nachricht. message kann die Form eines Wertes oder eines Ausdruckes haben. Der Wert muss zwischen 0 und 255 liegen, nur die ersten 8 Bits des Argumentes werden gesendet.**
- **SendMessage(3); // sende Nachricht 3**
- **SendMessage(259); // andere Möglichkeit, Nachricht 3 zu senden**

NQC- Programmierschnittstelle (API)

- **Timer**

  Die RCX-Einheit verfügt über vier Timer. Die interne Zeitspanne zwischen den Ticks beträgt 100ms (10 Ticks pro Sekunde). Die maximale Laufzeit eines Timers beträgt 32767 Ticks (ungefähr 55 min), danach wird wieder von 0 gezählt.

- **ClearTimer(n)**
  Löscht und initialisiert den Timer n.
- **ClearTimer(0);**

- **Timer(n)**
  Gibt den Wert des Timers n zurück.

NQC- Programmierschnittstelle (API)

- **Allgemeine Funktionen**
- **Wait(time) // Hält die Ausführung einer Task für die in time angegebene Zeitspanne an. Das Argument kann eine Konstante oder ein Ausdruck sein.**
  Wait(100); // warte 1 Sekunde
- **Wait(Random(100)); // warte zufallsbedingt bis höchstens 1 Sekunde**
- **StopAllTasks() // Hält alle gerade laufenden Tasks an. Der Befehlt hält das Programm vollkommen an, so dass jeder anschliessender Befehl ignoriert wird.**
- **StopAllTasks(); // hält das Programm an**

NQC- Programmierschnittstelle (API)

- **Allgemeine Funktionen**
- **Random(n) // Gibt eine Zufallszahl zwischen 0 und n zurück.**
  x = Random(10);
- **SetSleepTime(minutes) //Setzt den Timer für die Zeitabschaltung, minutes muss eine Konstante sein.**
  SetSleepTime(5); // Abschalten nach 5 min.
  SetSleepTime(0); // deaktiviert die Zeitabschaltung.
- **SleepNow() // Erzwingt den Schlafmodus.**
  SleepNow(); // jetzt schlafen.

Literatur

[1] Lego-Mindstorms http://mindstorms.lego.com

[2] http://www.philohome.com/motors/motorcomp.htm

[3] Lego-Education-Center. http://www.mooreed.com.au.

[4] *NQC Programmer's Guide* Version 3.1 r5 by Dave Baum & John Hansen

Beispiele

```
// speed.nqc – Setzt Power, geht vorwärts, wartet,
// geht rückwärts
task main()
{
  SetPower(OUT_A+OUT_C,2);
  OnFwd(OUT_A+OUT_C);
  Wait(400);
  OnRev(OUT_A+OUT_C);
  Wait(400);
  Off(OUT_A+OUT_C);
}
```

Spirale

```
// spiral.nqc – Benutzt repeat & Variablen, um Roboter in einer
// Spirale laufen zu lassen
#define TURN_TIME   100
int move_time;                  // Definiere Variable
task main()
{  move_time = 20;              // Setze initialen Wert
  repeat(50)
  { OnFwd(OUT_A+OUT_C);
    Wait(move_time);            // Benutze Variable für Pause
    OnRev(OUT_C);
    Wait(TURN_TIME);
    move_time += 5;             // Erhöhe Variable
  }  Off(OUT_A+OUT_C); }
```

Benutzung der Berührungssensoren

```
task main()
{ SetSensor(SENSOR_1,SENSOR_TOUCH);
  OnFwd(OUT_A+OUT_C);
  while (true)
  { if (SENSOR_1 == 1)
    { OnRev(OUT_A+OUT_C); Wait(30);
      OnFwd(OUT_A); Wait(30);
      OnFwd(OUT_A+OUT_C);
    }
  }
}
```

Benutzung eines Lichtsensors

```
// Roboter läuft vorwärts bis er schwarz sieht
// Danach dreht er, bis er wieder weiß sieht
#define THRESHOLD 37
task main()
{ SetSensor(SENSOR_2,SENSOR_LIGHT);
  OnFwd(OUT_A+OUT_C);
  while (true)
  { if (SENSOR_2 < THRESHOLD)
    { OnRev(OUT_C);
      Wait(10);
      until (SENSOR_2 >= THRESHOLD);
      OnFwd(OUT_A+OUT_C);
    } } }
```

Tasks

```
task main()
{ SetSensor(SENSOR_1,SENSOR_TOUCH);
  start check_sensors;
  start move_square; }
task move_square()
{ while (true)
  { OnFwd(OUT_A+OUT_C); Wait(100);
    OnRev(OUT_C); Wait(85);  } }
task check_sensors()
{ while (true)
  { if (SENSOR_1 == 1)
    { stop move_square;
      OnRev(OUT_A+OUT_C); Wait(50);
      OnFwd(OUT_A); Wait(85);
      start move_square; } } }
```

Subroutines

```
sub turn_around()
{ OnRev(OUT_C); Wait(400);
  OnFwd(OUT_A+OUT_C);
}
task main()
{ OnFwd(OUT_A+OUT_C);
  Wait(100);
  turn_around();
  Wait(200);
  turn_around();
  Wait(100);
  turn_around();
  Off(OUT_A+OUT_C);}
```

Inline Funktionen, *call by reference*

```
void turn_around(int turntime)
{ OnRev(OUT_C); Wait(turntime);
  OnFwd(OUT_A+OUT_C); }
task main()
{ OnFwd(OUT_A+OUT_C);
  Wait(100);
  turn_around(200);
  Wait(200);
  turn_around(50);
  Wait(100);
  turn_around(300);
  Off(OUT_A+OUT_C); }
```

```
task main()
{  int count=0;
   while (count<=5)
   {  PlaySound(SOUND_CLICK);
      Wait(count*20);
      increment(count);
   }
}
void increment(int& n)
{ n++; }
```

Macros

```
#define turn_right(s,t)
SetPower(OUT_A+OUT_C,s);OnFwd(OUT_A);OnRev(OUT_C);Wait(t);
#define turn_left(s,t)
SetPower(OUT_A+OUT_C,s);OnRev(OUT_A);OnFwd(OUT_C);Wait(t);
#define forwards(s,t)
SetPower(OUT_A+OUT_C,s);OnFwd(OUT_A+OUT_C);Wait(t);
#define backwards(s,t)
SetPower(OUT_A+OUT_C,s);OnRev(OUT_A+OUT_C);Wait(t);
task main()
{ forwards(1,200);   turn_left(7,85);
  forwards(4,100);   backwards(1,200);
  forwards(7,100);   turn_right(4,85);
  forwards(1,200);   Off(OUT_A+OUT_C);}
```

Ende

Viel Spaß mit den Robotern im HAPRA!