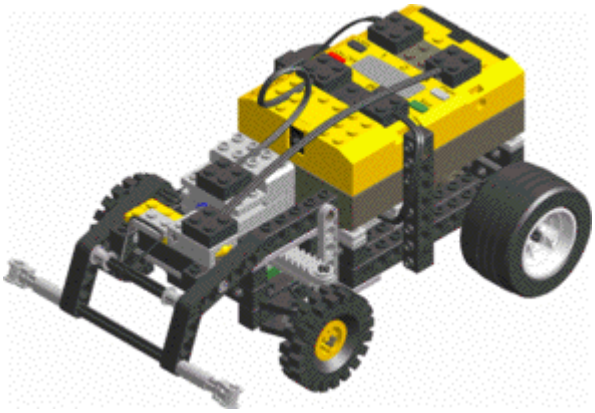
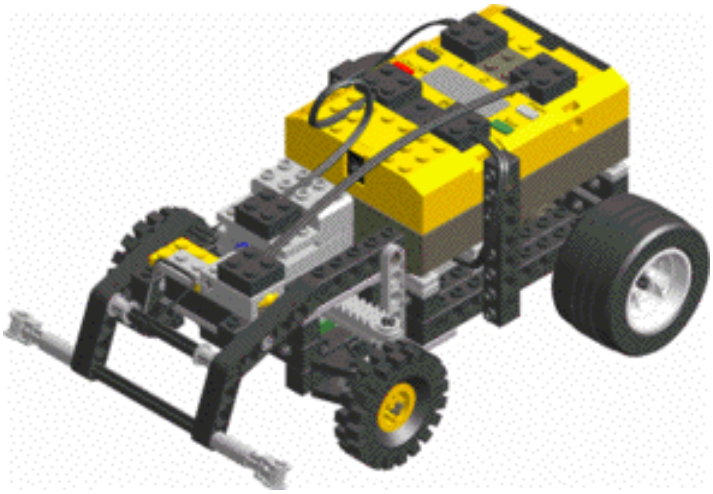
Projektwoche Lego Robotics

Dr. J. Boillat



Projektwoche Lego Robotics



Aufgabenstellung

Konstruieren Sie einen Lego-Roboter, der in der Lage ist, eine Wand in einem Zimmer zu verfolgen.
Der Roboter besteht aus 2 Sensoren und 2 Motoren Die Programmierung erfolgt mit NQC.

Unterlagen

- Kursunterlagen
  - Kursunterlagen in PDF-Format
- Mindstorms[tm]
  - Lego Mindstorms [www.legomindstorms.com/]
  - Lego Mindstorms Internals [www.crynwr.com/lego-robotics/]
  - RCX Internals [graphics.stanford.edu/~kekoa/rcx/]
- Robotics am MIT
  - The Epistemology and Learning Group [el.www.media.mit.edu/groups/el/]
  - The MIT Programmable Brick [el.www.media.mit.edu/groups/el/projects/programmable-brick/]
  - The Art of LEGO Design [/cherupakha.media.mit.edu/pub/people/fredm/artoflego.pdf]
  - Fred's 6.270 Home Page [lcs.www.media.mit.edu/people/fredm/projects/6270/]
- Software
  - NQC - Not Quite C [www.enteract.com/~dbaum/lego/nqc/]
  - RCX Command Center [www.cs.uu.nl/people/markov/lego/rcxcc/]
  - NQC-Tutorial [www.cs.uu.nl/people/markov/lego/]
- Literatur
  - Jonathan B. Knudsen, The Unofficial Guide to LEGO® MINDSTORMS[tm] Robots [www.oreilly.com/catalog/lmstorms/] , O'Reilly, Sebastopol, 1999, ISBN 1-56592-692-7.
  - Dave Baum, Dave Baum's Definitive Guide to LEGO® MINDSTORMS? [www.apress.com/Catalog/Mindstorms.htm] , Apress, Emeryville, 1999, ISBN 1-893115-09-7

This article was originally published in The Robotics Practitioner: The Journal for Robot Builders, *volume 1, number 2, Spring 1995;* trp@footfalls.com

The Art of LEGO Design

Fred G. Martin[1]

March 15, 1995

There is a real need for better resources for both fledgling and intermediate LEGO builders. The plans that the LEGO company distributes with its kits are very good at showing how to build specific models, but not so good at teaching how to design from one's own ideas. At the MIT Media Laboratory, we're working on a project we call the *LEGO Constructopedia*, a hypermedia resource for LEGO designers that will include LEGO building plans, design principles, textual descriptions, and rendered animations, all interlinked, indexed, and browsable. The project is just beginning and is still in the conceptual stages; this article is my attempt to present some of the content of our proposed LEGO Constructopedia in a more traditional form.

The article begins with an analysis of the structural principles of the LEGO system, continues with a discussion of gears, gear reduction, and geartrains, and finishes with a visual assortment of various building tricks or "clichés." Interspersed throughout are numerous diagrams and sample models to illustrate the ideas being presented. I hope that LEGO aficionados at all levels from novice to expert will find something of interest here.

Structure

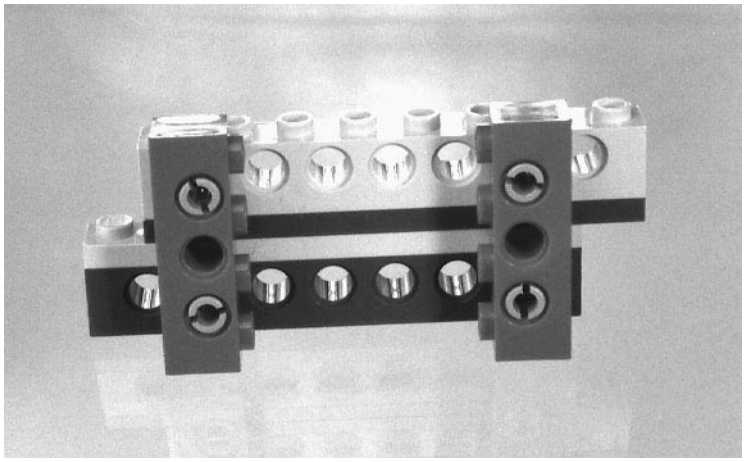
The Vertical Dimension Relation

Let's begin by examining the LEGO brick in detail. Most people realize that the LEGO brick is not based on a cubic form. The height of the brick is a larger measure than the length and width (assuming the normal viewpoint of studs on the top). But few people know the secret relationship between these dimensions: the vertical unit is precisely 6/5 times the horizontal ones. Put another way, a stack of five LEGO bricks is exactly equal in height as a six-stud LEGO beam is long.

The origins of this obscure relationship remain shrouded in mystery, but it has real practical value: by building structures with vertical heights equal to integral horizontal lengths, it is possible to use beams to brace LEGO constructions. This technique is greatly facilitated by the one-third-height plates, which allow a number of vertical spacing possibilities.

The most common trick is to create two horizontal units of space in the vertical dimension by separating two beams with two plates (Figure 1). This $1\frac{2}{3}$ vertical measure is two units of horizontal measure since $1\frac{2}{3}$ times the conversion factor of 6/5

[1]The Media Laboratory at the Massachusetts Institute of Technology, 20 Ames Street Room E15–320, Cambridge, MA 02139. E-mail: fredm@media.mit.edu. This document is Copyright © 1995 by Fred G. Martin. It may be distributed freely in verbatim form provided that no fee is collected for its distribution (other than reasonable reproduction costs) and this copyright notice is included. An electronic version of this document is available via anonymous FTP from cherupakha.media.mit.edu (Internet 18.85.0.47) in directory pub/people/fredm.



Two beams are separated by two $\frac{1}{3}$-height LEGO plates, creating a vertical interval of $1\frac{2}{3}$ units, which is equal to 2 horizontal units. Hence the beams can be locked into place using cross-beams and connector pegs—*the* way to make your LEGO construction quite sturdy.

Figure 1: Two Beams Locked Using $1\frac{2}{3}$ Vertical Spacing Relation

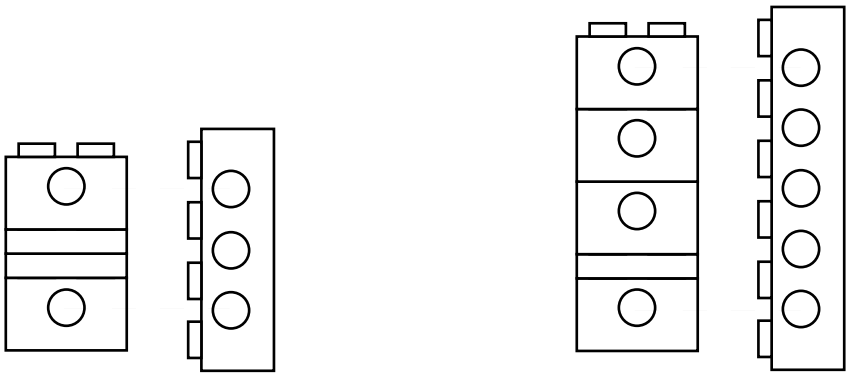


Figure 2: Creating Vertical Spacings with Two-Unit and Four-Unit Horizontal Measures

The black connector peg vs. the gray connector peg: what is the difference? The answer is that the black peg is *slightly larger*, so it fits quite snugly in the beam hole, while the smaller gray peg rotates freely. Use the black pegs to binding structures together, as suggested by the discussion on locking cross-beams, and use the gray peg when making hinged joints.

Figure 3: Black Connector Peg Versus Gray Connector Peg

equals 2. Another useful pairing is $3\frac{1}{3}$ vertical units (i.e., two beams separated by two beams/bricks and one plate) which equals 4 horizontal units (see Figure 2).

In addition to constructing perfect spacings vertically, it's possible to make diagonal braces. A 3-unit horizontal spacing with a 4-unit horizontal spacing vertically yields a 5-unit diagonal by the Pythagorean relation. This is an example of a perfect diagonal spacing, but near-perfect spacings that are "close enough" exist. Experiment, or spend some time thinking about the numbers.

Figure 1 shows the practical application of this dimensional relation: two beams locked together with cross-beams and connector pegs. You can use the vertical spacing trick for at least two purposes. First, use it to lock vertical structures on LEGO machines in place with beams and connector pegs (see more about connector pegs in Figure 3). Second, create vertical spacings that are the right intervals to allow gears to mesh properly (more on this later). This trick will go a long, long way in making sturdy, reliable LEGO designs.

Gearing

Turn on a small DC motor, like the stock LEGO motor, and what do you get? The shaft spins really fast, but with almost no torque (turning force). You can easily pinch the shaft with your fingertips and stop the motor from turning.

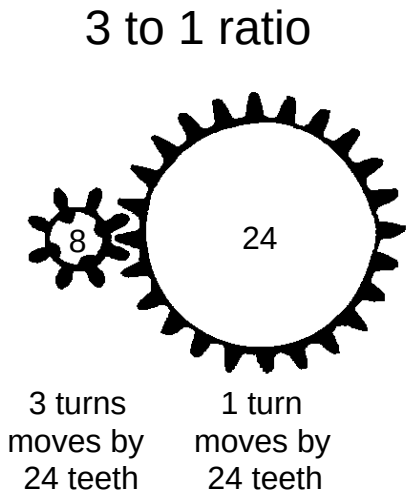
Through the magic of gear reduction, this fast-but-weak motor energy can be transformed into a strong but slow rotation, suitable for powering wheels, gripper hands, elbow joints, and any other mechanism. Along with structural issues, building effective geartrains is the other half of the challenge of creating working LEGO machines.

Counting Gear Teeth

Gear reduction is achieved by intermeshing gears of different sizes with compatible teeth. Figure 4 shows the effect of meshing an 8–tooth gear with a 24–tooth gear. When the 8–tooth gear rotates three times, it has advanced the 24–tooth gear one revolution. Hence this configuration produces a 3-to-1 gear reduction ratio.

More gear reduction can be achieved by meshing gears with greater disparities of teeth count. Using the LEGO 8–tooth and the LEGO 40–tooth gears produces a 5-to-1 reduction. But the more general solution is to gang together—or multiply—single pairs of gear reduction. Figure 5 shows how two 3-to-1 reductions may be ganged to produce a 9-to-1 reduction, by using a shaft that holds a 24–tooth input gear *and* an 8–tooth output gear.

The gear ganging concept is the foundation of gear trains. Figure 6 shows a model LEGO gear train that produces a 243-to-1 reduction from the motor shaft to the output wheel. The example is a bit of overkill—this much reduction will



When the 8–tooth gear rotates 3 times, it advances the meshed gear by a total of 24 teeth. Since the meshed gear has 24 teeth, it rotates exactly once. Hence this configuration produces a 3:1 ratio of gear reduction: three turns of the input gear causes one turn of the output gear.

Figure 4: 3-to-1 Gear Reduction Ratio

By ganging together—or multiplying—two 3-to-1 gear reductions, a 9-to-1 output reduction can be achieved. The key is to use intermediary shafts that hold large input gears (e.g., a 24–tooth) and small output gears (e.g., an 8–tooth).

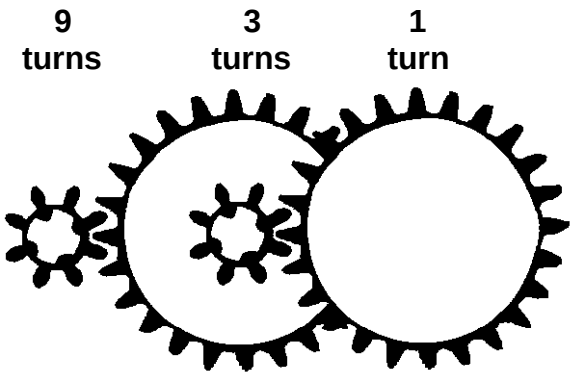
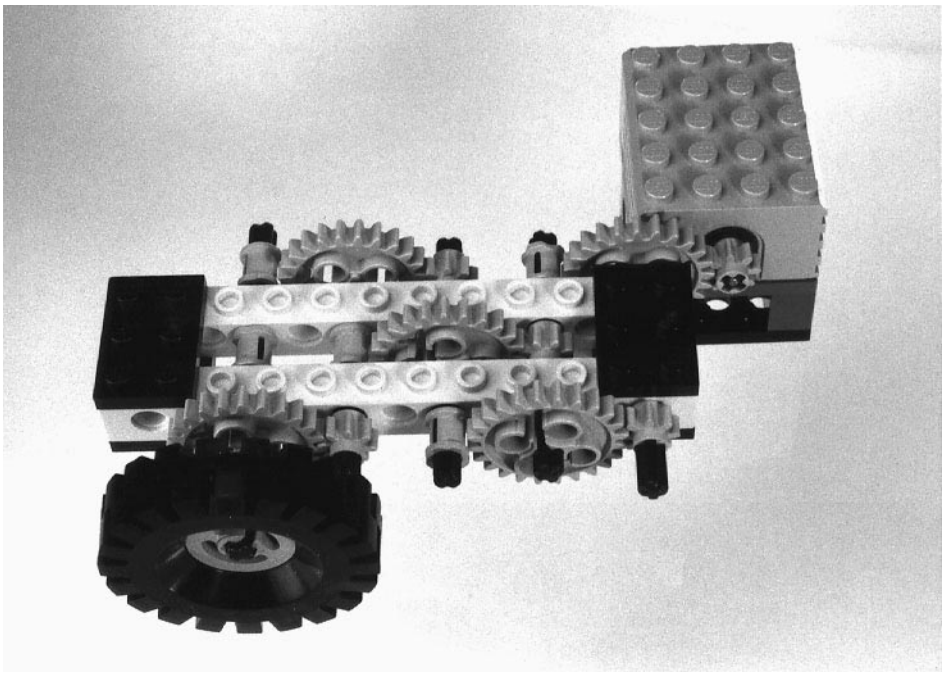


Figure 5: 9-to-1 Gear Reduction with Ganging



A five-stage reduction using 8– and 24–tooth gears creates a 243-to-1 reduction in this sample LEGO geartrain. Note the need for three parallel planes of motion to prevent the gears from interfering with one another. Four 2×3 LEGO plates are used to hold the beams square and keep the axles from binding.

Figure 6: Sample LEGO Geartrain

produce too slow a final rotation for the typical robot drive train—but it serves to illustrate the point.

When I present gear reduction to kids, I find it difficult to give a direct explanation of why it works. More precisely, by counting teeth it's evident enough that subsequent gears run slower, but why do they have correspondingly more torque? I generally appeal to a vague "energy must be conserved" line of reasoning. Ultimately, there's no substitute for holding a live geartrain in your hand and feeling the power as gear reduction does its work.

The Worm Gear

The worm gear is a fascinating invention, sort of a Mobius strip in the world of gears. When meshed with a conventional round gear, the worm creates an *n*-to-1 reduction: each revolution of the worm gear advances the opposing gear by just one tooth. So, for example, it takes 24 rotations of the worm to revolve the 24–tooth round gear once. This forms quite a

The worm gear is valuable because it acts as a gear with one tooth: each revolution of the worm gear advances the round gear it's driving by just one tooth. So the worm gear meshed with a 24-tooth gear (as pictured) yields a whopping 24 to 1 reduction. However, the worm gear loses a lot of power to friction, so it may not be suitable for high performance, main drive applications.

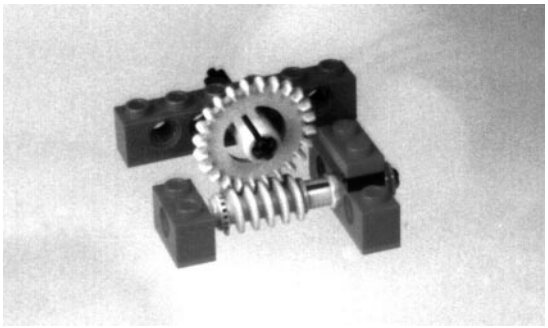
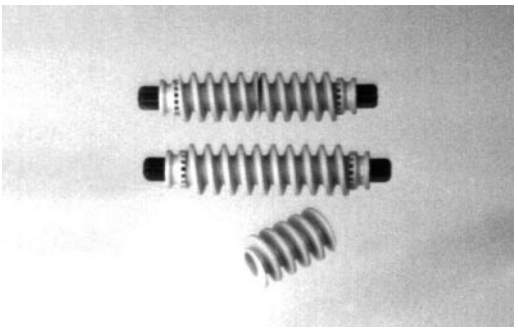


Figure 7: Using the Worm Gear



This diagram shows an arrangement of worm gears. At the bottom is the basic worm gear, two horizontal LEGO units in length. At the top is an unsuccessful attempt to put two worm gears on the same shaft. In the middle is the successful attempt. When placing multiple worm gears on a shaft, the trick is to try all four possible orientations to find the one that works.

Figure 8: Multiple Worm Gears on One Shaft

compact gear reduction—it would take about three gangs of the 3-to-1 reduction, forming a 27-to-1 relation, to do the same work as a single worm meshed with a 24–tooth round gear.

There is a drawback, however. The worm gear uses predominantly sliding friction when advancing the teeth of the round gear. The teeth of round gears are generally designed to minimize sliding effects when they are meshed with each other, but there's no getting around the problem with worm gears.

Thus worm gears create more frictional losses than round gears. At higher torques they have a tendency to cause a geartrain to stall. If your robot's too heavy, a worm gear drive may not work well as its main drive.

Worm gears have another interesting property: they can't be back-driven. That is, if you rotate the gear being driven by a worm, you'll just push the worm gear forward and back along its axle, but you won't get it to turn. Take advantage of this property. For example, if a worm gear is used to raise an arm lifting a weight, then the arm won't fall down after power is removed from the motor.

Figure 7 shows how to mesh a worm gear to a round gear, and Figure 8 illustrates putting two LEGO worm gears onto the same shaft.

Changing Axis of Rotation

In a geartrain with only round gears, all of the axles must be mutually parallel. With the worm gear, the output round gear's axis of rotation is at right angles to the worm's. Two other kinds of gears, the *crown gear* and *bevel gear* are available in the LEGO kit for changing the axis of rotation within a geartrain.

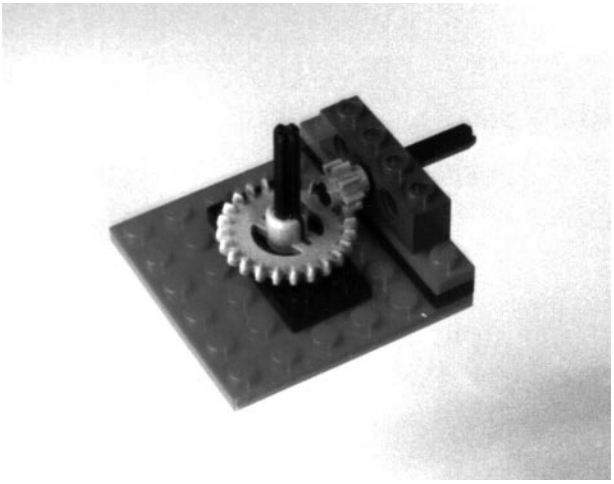
The Crown Gear

The crown gear is a round gear that is specially designed to mesh at right angles to the standard round gear (Figure 9). In the diagram, the crown gear is shown meshing with the 8–tooth gear. Meshing to the round 24–tooth and 40–tooth gear is also possible, though using the 8–tooth to drive the 24–tooth crown gear is an effective way to build in a gear reduction while changing the rotation axis.

The 24–tooth crown gear is the same size as the standard 24–tooth round gear, so it can be used as a replacement for that gear when a parts shortage occurs.

The Bevel Gear

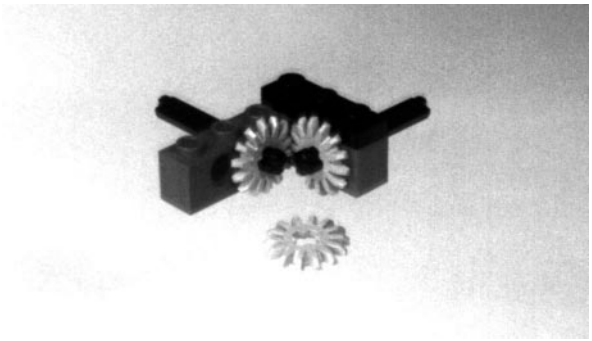
The bevel gear is used in pairs to provide a similar function to the crown gear, though without the capability for gear reduction. There are two styles of bevel gear: the older style (Figure 10), which is fairly flat, and a newer style, which is



The 8–tooth gear, in conjunction with the 24-tooth *crown gear*, is used to change the axis of rotation in a gear train.

In this instance, the configuration provides for a vertical shaft output. Horizontal output also possible.

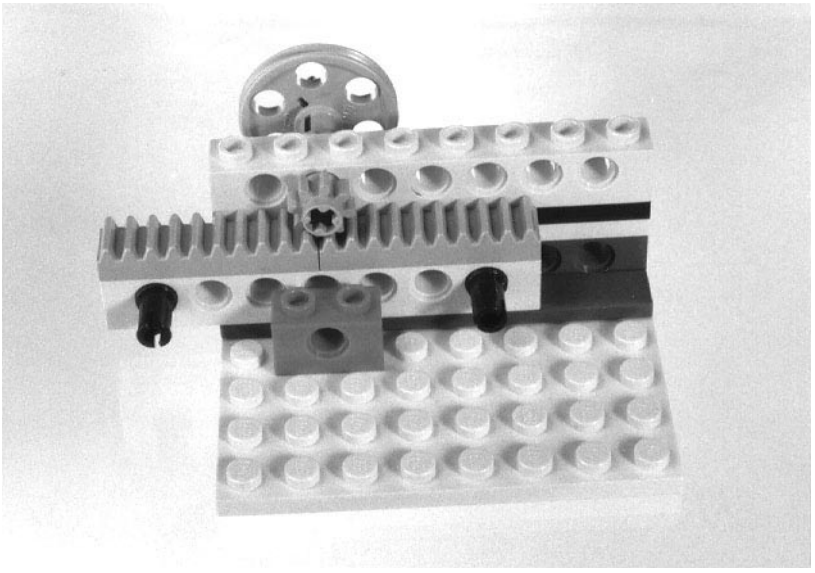
Figure 9: 8–Tooth Gear Meshing with Crown Gear



The bevel gears are used to change the angle of rotation of shafts in a gear train with a 1:1 ratio. In this case, they are used to effect a change in the horizontal plane.

This picture shows the older-style bevel gears, which have limited usefulness due to their relatively high friction and lack of strength. The newer bevel gears are thicker and perform much better.

Figure 10: The Bevel Gear



The gear driving the gear rack is often referred to as the "pinion," as in "rack-and-pinion steering," which uses the transverse motion of the gear rack to orient wheels. The 8–tooth gear is a good candidate to drive the rack because of the gear reduction it achieves—one revolution of the gear moves the rack by eight teeth.

Figure 11: Using the Gear Rack

the same diameter but thicker. The old style bevel gear is somewhat flimsy and lossy and is not suitable for delivering larger torques. The new bevel gear is a significant improvement.

Old style bevel gears can be put to good use by serving as stop bushes (Figure 15).

The Gear Rack

The gear rack is a like a round gear unrolled and laid out flat. When driven by a round gear (the 8–tooth usually works best), it traverses back and forth in a linear motion (Figure 11).

Gear racks can be laid end-to-end to make longer stretches of motion. Underneath a beam driven by gear racks, use the smooth-topped LEGO plates as a surface for the beam to slide on.

Practical Hints

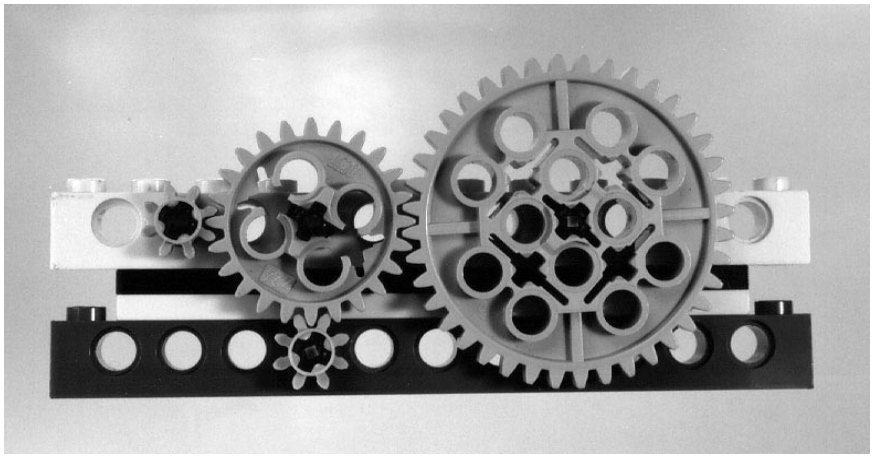
For the remainder of this section on LEGO gearing, I'll present a number of assorted tips to assist in your geartrain designs.

Gear Sizing

It is helpful to know the sizes of the standard gears. This links back to the earlier section on the LEGO dimensional relation—creating unit horizontal spacings in the vertical dimension can be used not only to lock structures into place, but to mesh gears properly above and below one another.

Of the four round gears, three of them—the 8–tooth, the 24–tooth, and the 40–tooth, have a radius that is a whole number of horizonal LEGO units *plus one-half of a unit*. Therefore, these three gears form an whole-unit spacing when their radii are added—i.e., when they are meshed together in a geartrain.

For example, the 8–tooth gear has a radius of one-half of a unit, and the 24–tooth, $1\frac{1}{2}$ units, so when properly meshed together, their centers are spaced at two horizontal units. A spacing of two horizontal units is readily available on LEGO beams, or can be constructed using the $1\frac{2}{3}$ vertical spacing relationship discussed earlier. Figure 12 shows how the three half-radii gears mesh with one another.



The 8–tooth, 24–tooth, and 40–tooth round gears all mesh properly along a horizontal beam because they have "half unit" radii. For example, the 8–tooth gear has a radius of $\frac{1}{2}$ LEGO units, and the 24–tooth gear, $1\frac{1}{2}$ units, so they mesh at a spacing of two horizontal LEGO units.

The example shows the 8– and 24–tooth gears meshed horizontally at two units, and, using the $1\frac{2}{3}$ vertical spacing trick, vertically as well (a common and useful configuration).

Figure 12: The Half-Radius Round Gears

The 16–tooth gear, on the other hand, has a radius of one LEGO unit, so it meshes properly only with itself (at the standard horizontal unit spacing). A pair of two 16–tooth gears thus requires a space of two LEGO units, which happens to be the same interval as the pair of an 8–tooth and a 24–tooth gear. Thus these respective pairs of gears may be easily interchanged—a useful trick for adjusting the performance of an existing geartrain without a performing major overhaul (Figure 13).

Odd Gear Spacings

It's possible to mesh gears at odd spacings using diagonal mounting. Generally, the gears work better when slightly too far apart than when too tight, which causes them to bind.

Many combinations are possible when creating a space diagonally, and some of them work. For example, and 8– and 16–tooth gear will function when spaced along the diagonal of one horizontal unit and one vertical unit.

An interesting exercise is to calculate the effective spacings (in horizontal units) of various diagonal measures using the Pythagorean formula, and then see which come close enough to a pairing of gears to be useful. But I suggest experimentation.

Supporting Axles

It's important to keep axles well-supported in a geartrain. Practically, this means using at least two beams to carry the axles. More importantly, all beams with common axles running through them must be held together squarely. If the beams not held together, the axles will bind and lock up inside the beam-hole bearing mounts.

As suggested in Figure 14, use the $2\times$ parts, not the $1\times$ parts, to hold beams in place. Figure 6, the sample geartrain, uses 2×3 plates to squarely support the beams.

Using Stop Bushes

The stop bushes, or axle holders, are to keep axles from sliding back and forth in their mounts. In addition to the standard full-width stop bush, the small pulley wheel and the bevel gear may be put into service (Figure 15).

The 16–tooth gear has a radius of 1 LEGO unit, so two of them mesh properly together at a spacing of two units (left side of diagram). Since an 8– and 24–tooth gear also mesh at two-unit spacing, these respective pairs of gears can be swapped for one another in an existing geartrain—a handy way to change the performance of a geartrain without rebuilding it from scratch.

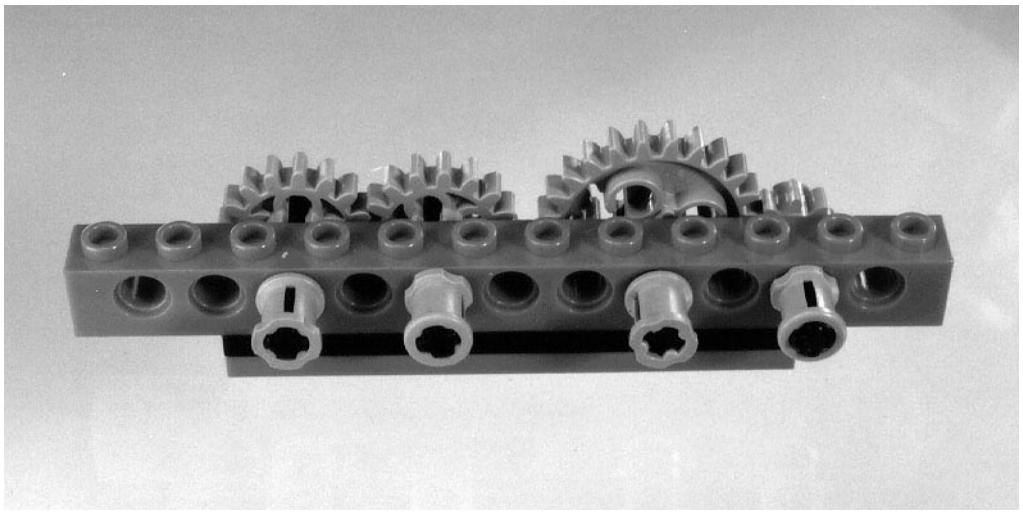


Figure 13: The 16–Tooth Gears

It seems like an obvious point, but using the 2× parts to hold beams parallel is quite important when the beams will be carrying common axles. If beams are not held squarely, the axles will bind and freeze inside the beam-hold bearing supports.

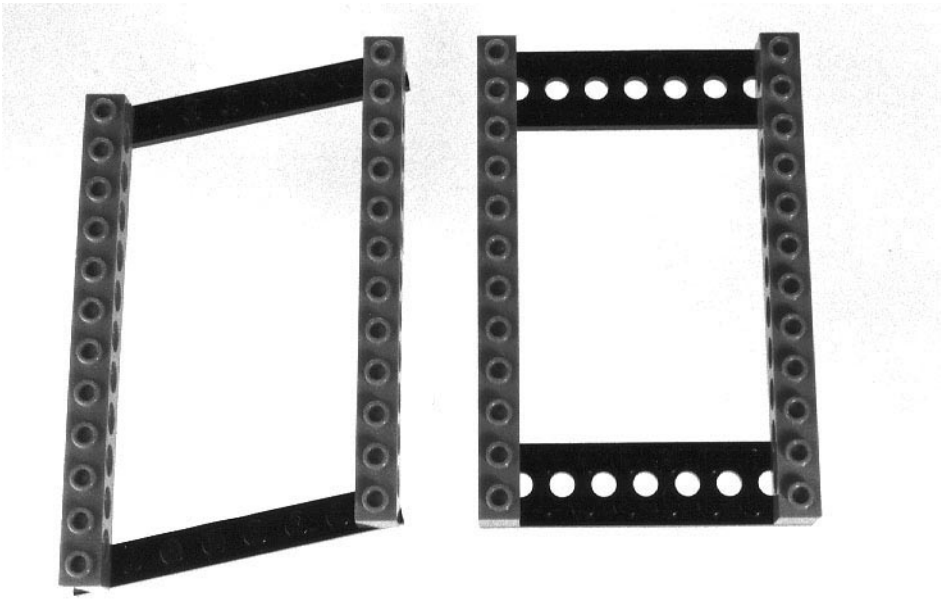
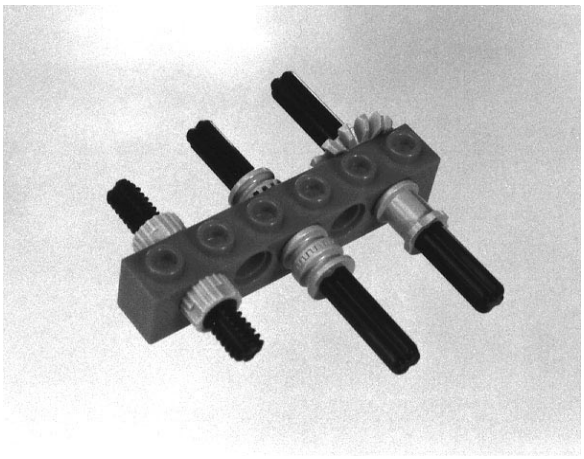


Figure 14: Locking Parallel Beams Together with 2× Parts



The standard 1-LEGO-long stop bush (upper axle, front) is not the only part that can act as a bushing (axle holder). Use the small pulley wheel (middle axle) to act as a half-sized spacer—it also grabs tighter than the full bush. In a pinch, the bevel gear (upper axle, back) makes a great bushing. Finally, the nut-and-bolt parts (lower axle) can be used to make a tight connection (if you can find them).

Figure 15: The Stop Bushes and Other Parts

Reducing Noise with Pulley Wheels

Sometimes a geartrain will be quite noisy. Usually most of the noise is generated by the very first meshing of gears from the motor. Here is the ideal place to use a pulley wheel drive (Figure 16).

Use the small pulley wheel on the motor shaft, and the medium or large pulley wheel on the driven shaft. The ratio of the circumferences of two pulleys creates a reduction just like the ratio of the gear teeth of a pair of meshed gears.

LEGO pulley drive belts—thin rubber bands—are best when used in high speed, low torque situations, because they can't transmit a lot of force. So the first stage is really the best place to use a pulley drive.

Be careful, though, about using pulleys in a competitive situation. They have a penchant for breaking or falling off at the most inopportune moment.

Chain Link Drive

Chain link drives are best suited for the final stage of a geartrain—transmitting power down to the axles holding the wheels, for example. This is because they can easily deliver the necessary torques, and they impose frictional losses that are minimized when rotational speeds are low.

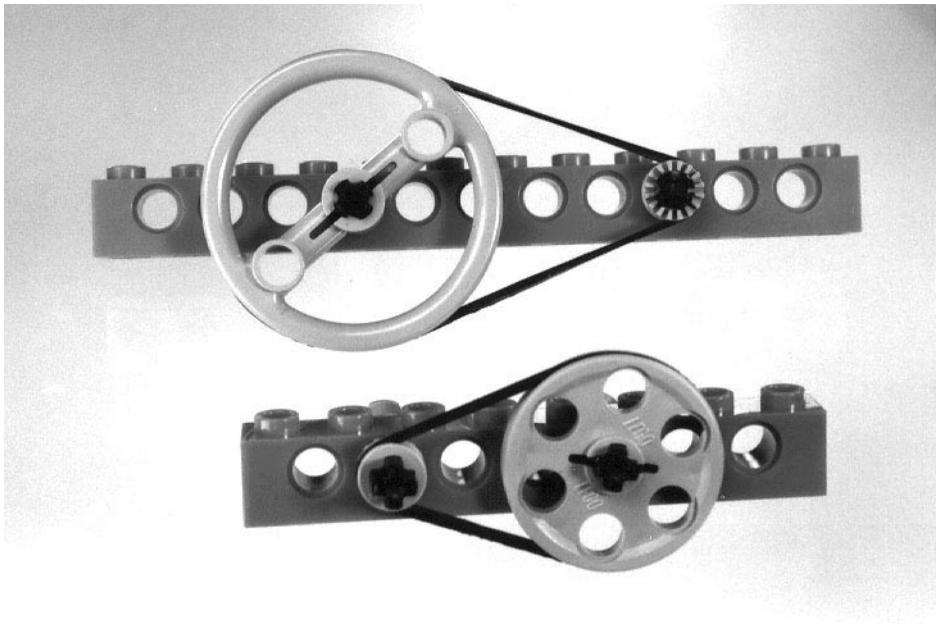
Getting the right amount of chain links can be tricky. Generally a looser chain works better—chains that are too tight will bind up. But too loose a chain will skip when the going gets tough.

Design Strategy

When designing a new geartrain into a model, I find it best to work *backward from the final drive*, rather than forward from the motor. This makes sense because usually there is a fair bit of flexibility about where the motor is ultimately mounted, but much less in the placement of drive wheels or leg joints (for example).

So start by mounting the axle shaft that will carry the final drive, put a wheel and gear on it, and start working backward, adding gearing until there is enough, and finally mount the motor in a convenient spot.

When designing a vehicle, don't forget about the role of the tire in determining the relationship between the rotational speed of the final drive axle and the linear speed that is achieved. Small tires act as gear reductions with respect to large tires, and this may have an effect on how much gear reduction is necessary. Experiment!



There are three sizes of pulley wheel: the tiny one, which doubles as a stop bush, the medium-sized one, which doubles as a tire hub, and the large-sized one, which is sometimes used as a steering wheel in official LEGO plans.

Figure 16: Using Pulley Wheels

Chain link can be an effective way to deliver large amounts of torque to a final drive, while providing a gear reduction if needed.

Chain link works best at the slower stages of gearing, and with a somewhat slack linkage. Use the larger gears—the 8–tooth one won't work very well.

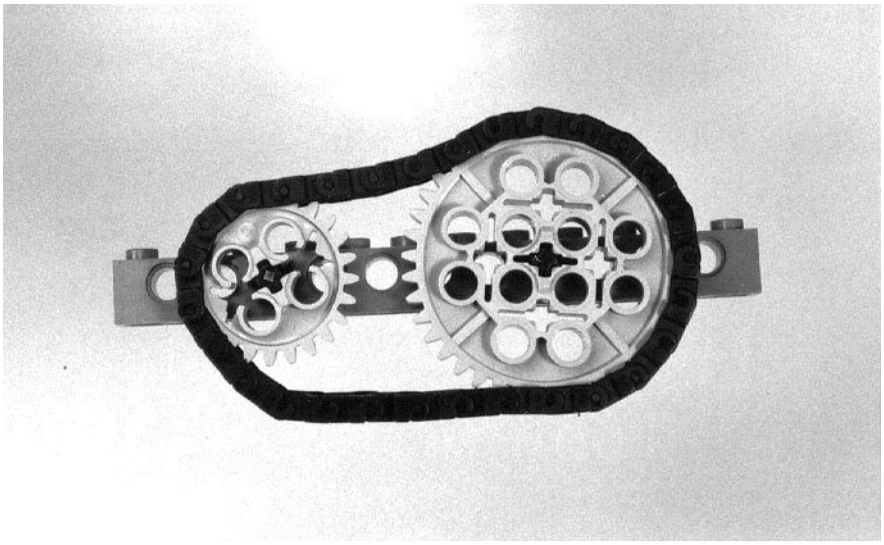
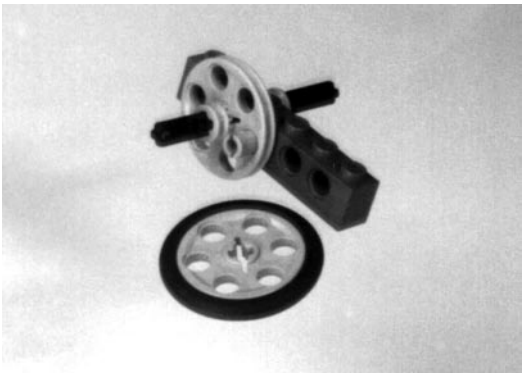


Figure 17: Chain Link Drive



On occasion it is necessary to lock a beam to an axle. This figure shows how to use a medium pulley wheel, which rigidly locks to an axle, to hold the beam in place.

Figure 18: Axle Locked Through Beam Using Pulley Wheel

The special “gear mounter” piece is an axle on one side and a loose connector peg on the other. It can be used to mount gears used as idlers in a gear train—used simply to transmit motion or to reverse the direction of rotation.

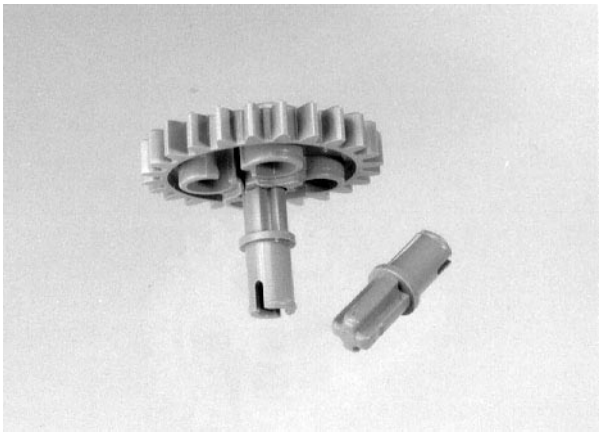


Figure 19: The Gear Mounter Part

If a geartrain seems to be performing badly, there are a few things to check. Make sure the stop bushes aren’t squeezing too hard—there should be some room for the axles to shift back and forth in their mounts. Check that all beams holding the axles are squarely locked together. The most common cause of poorly performing geartrains is beam mounts that aren’t square.

To test a geartrain, try driving it backward. Remove the motor, and gently but firmly turn the final drive wheel or shaft. If there isn’t too much friction, all of the gears in the train will start moving, with the motor’s gear spinning around rapidly. If your geartrain can be readily back-driven, it’s a sure sign that it’s performing well.

LEGO Design Clichés

This section presents a miscellaneous assortment of ideas in a visual fashion. I’ve come to call these LEGO ideas “clichés” because I hope that they become common, everyday knowledge, rather than secrets held by some small group of LEGO experts. I find myself inventing them time and again, on the spot, when working with kids helping them with their LEGO designs. Part of my intention in writing this article is to collect these clichés and share them with others.

Browse through this collection and perhaps you will find one or more of these techniques useful in your own LEGO designs.

This configuration of parts can be used as a compact axle joiner. LEGO now produces a part designed for this purpose, but in lieu of that part, this is a useful trick.

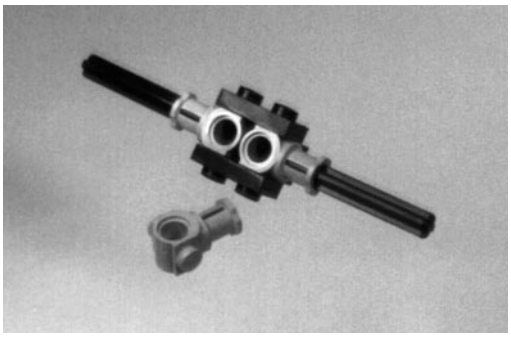


Figure 20: An Axle Joiner

In order to build outward from a vertical wall of axle holes, a smaller beam may be mounted with its top studs in the holes of the beam wall.

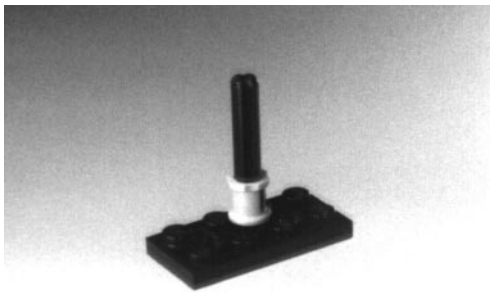
You will not see this configuration in LEGO's model plans, because the top studs are *slightly* too big for the axle holes, and a model left in this state will gradually experience solid flow as the stressed plastic expands. The official LEGO solution is to use the "connector peg with stud" parts (see Figure 22), but this method is actually stronger (or at least until the LEGO parts deform).

Figure 21: One Method of Building Outward from a Vertical Wall

The recommended way to build outward from a beam wall is to use the connector-peg-with-stud piece, which is a loose-style connector peg on one end and a top stud on the other.

This method is somewhat weaker than the method of simply plugging top studs into axle holes (Figure 21), but will not deform the plastic.

Figure 22: The Connector-Peg-With-Stud Part



The full-size stop bush can be used in one orientation to hold an axle through a plate hole so that the axle can freely rotate. In Figure 24, an additional plate is used to trap the axle, but allow it to rotate freely.

Figure 23: Using a Stop Bush to Retain an Axle

By using the stop bush to hold an axle in place between two plates, a vertical axle mount can easily be created. Depending on the orientation of the stop bush, it can be made to either lock the axle in place or allow it to rotate freely. In this diagram, the axle is allowed to rotate.

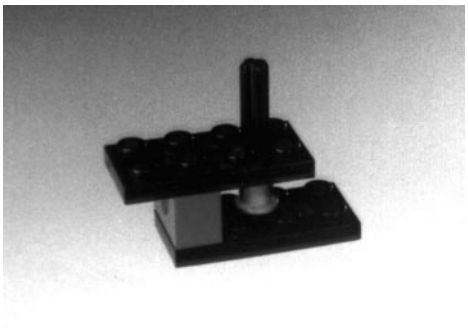
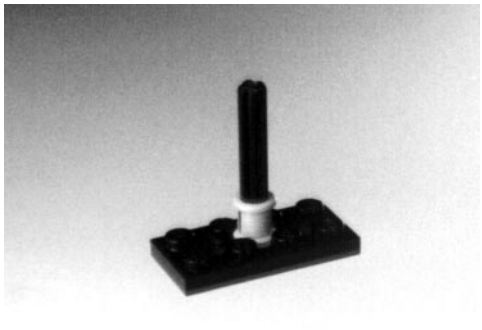


Figure 24: Trapping an Axle Between Two Plates Using Stop Bush



In the other orientation, the stop bush locks between four top studs, perfectly centered over the axle holes in flat plates. This allows the stop bush to lock a plate to an axle.

Figure 25: Using a Stop Bush to Lock an Axle to a Plate

The “toggle joint” can be used to lock two axles at a variety of odd angles. The short axle running through the two toggle joints is equipped with stop bushes on either end to hold the joint together.

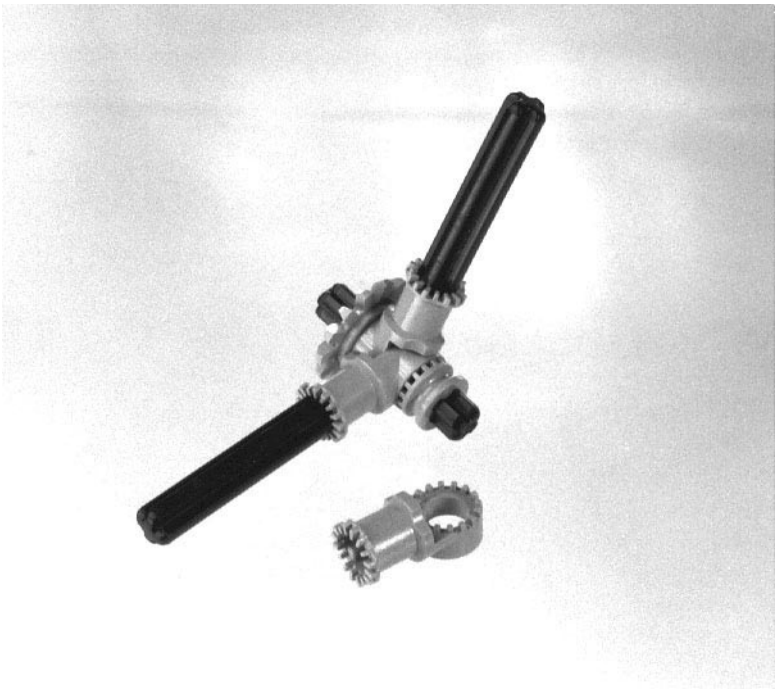


Figure 26: Two Toggle Joints

Here the toggle joint is used to connect two axles at right angles. The small pulley wheel is deployed on the axle that runs through the toggle joint to either lock the axle or allow it to rotate.

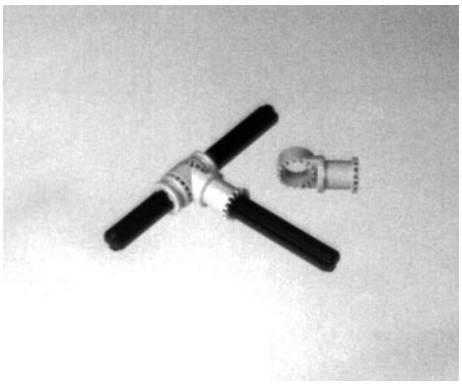
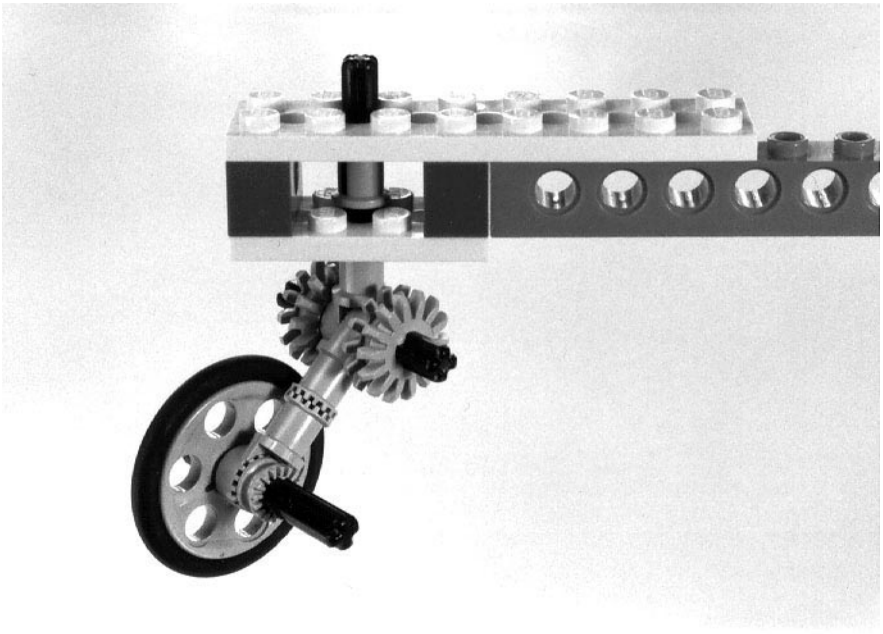


Figure 27: Toggle Joint With Free or Locked Axle



Several clichés are used to construct this caster wheel. The vertical axle is trapped between two plates, but allowed to rotate, using the trick shown in Figure 24. The angled joint down to the wheel is done using toggle joints in the configuration suggested in Figure 26, and the final mounting of the wheel is done using the toggle joint per Figure 27.

Figure 28: A Caster Design

The "piston rod" part (shown in the left foreground) is used twice in each mechanism to create a LEGO leg. By using a chain drive or gear linkage to lock legs in sync, a multi-legged creature can be designed.

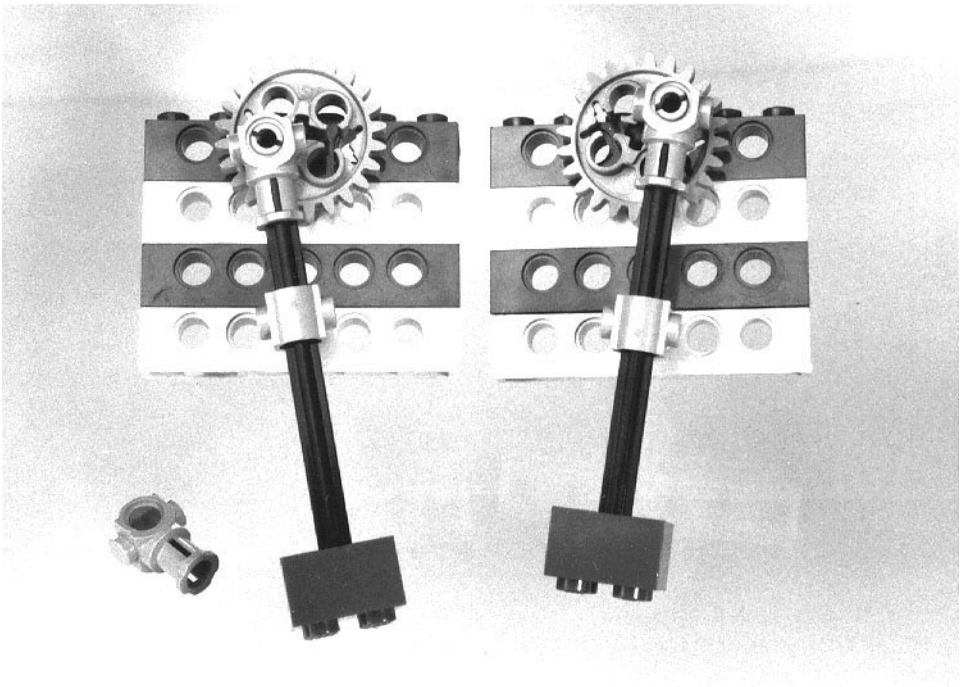
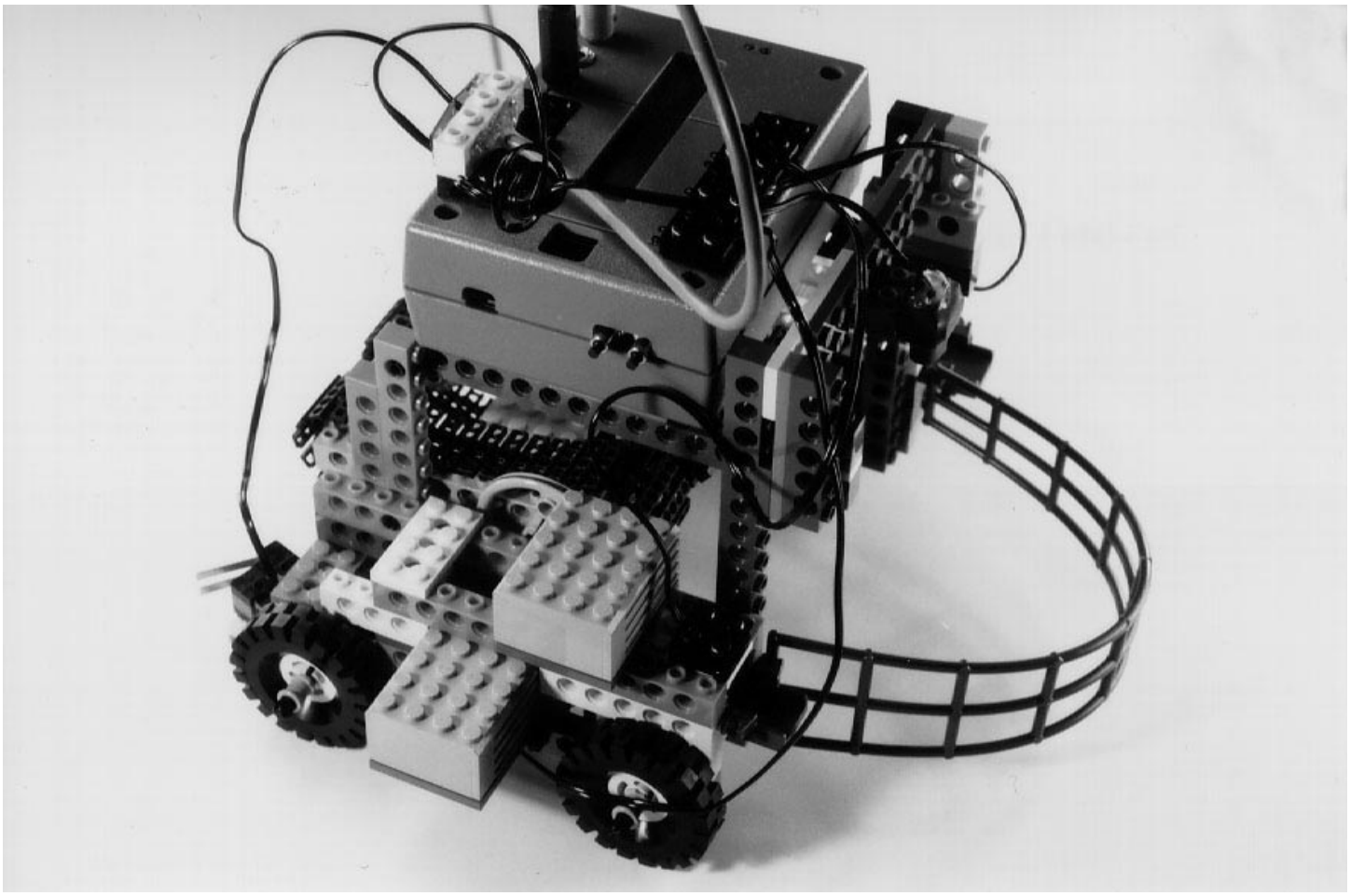


Figure 29: LEGO Legs



How many of the design clichés can you find in this robot? Look especially for the vertical spacing trick used to provide rigidity to the robot structure.

The robot, a ping-pong ball collector, was designed by the author and Brian Silverman. Sitting on top of it is the Programmable Brick, a robotics controller for kids recently developed by the author and his colleagues at the MIT Media Laboratory.

Figure 30: LEGO Ping-Pong Ball Collecting Robot

Closing

I hope that this article inspires others to contribute LEGO design clichés from their own vocabulary to a larger collection of resources for LEGO builders. By the time this article is printed, we will open a World Wide Web site representing successive versions of our LEGO Constructopedia, presenting these ideas in a hypermedia format and soliciting the contributions of others.

Acknowledgments

Steve Ocko guided my early work with the LEGO Technic system, and continues to inspire me with his LEGO creativity and his dedication to developing rich learning environments for kids. Mike Plusch and Randy Sargent, both LEGO geniuses, shared numerous ideas and insights with me, which I've incorporated into this article.

The LEGO Group is a continuing sponsor of our work at the Media Laboratory. They have provided both generous research funding and valued intellectual discourse on a wide range of topics related to children's learning and effective use of the LEGO materials. We are deeply grateful to their support of our work. Any criticism I make about their products should be taken in the spirit of making a great thing even better.

About the Author

Fred Martin has been developing educational robotics technologies at the MIT Media Laboratory since 1986, and is a co-founder of the annual MIT LEGO Robot Design Competition.

Fred is interested in robotics for fun and as a vehicle for exploring sensing, control, and engineering design. He has been teaching robotics to kids, teenagers, college students, and adults for nearly ten years and hasn't gotten tired of it yet.

Currently a Postdoctoral Fellow at the Media Lab, Fred is working on a robotics textbook aimed at undergraduate engineering students. Tentatively entitled *The Art of Robotics: A Hands-On Introduction to Engineering*, the book is scheduled to be published in the fall of 1996 by Benjamin/Cummings. This article will form the basis of an expanded chapter on LEGO design in the text.

Fred welcomes any comments on this article. He can be reached at `fredm@media.mit.edu`. The URL for the LEGO Constructopedia WWW site is:

```
http://el.www.media.mit.edu/groups/el/Projects/constructopedia
```

NQC Programmer's Guide

Version 2.0 rev 2, written by Dave Baum

Introduction

NQC stands for Not Quite C, and is a simple language for programming the LEGO RCX. The preprocessor and control structures of NQC are very similar to C. NQC is not a general purpose language - there are many restrictions that stem from limitations of the standard RCX firmware.

Although NQC was created specifically for the RCX, there still is a logical separation between the NQC language and the RCX API used to control the RCX. This division tends to get blurred is a few special cases such as multi-tasking support. In general, the compiler implements the language, and a special NQC file defines the RCX API in terms of the primitives of the language itself.

This document describes the NQC language and the RCX API. In short, it provides the information needed to write NQC programs. Since there are several different interfaces for NQC, this document does not describe how to use an NQC implementation. Refer to the documentation provided with the NQC tool, such as the *NQC User Manual* for information specific to that implementation.

For up-to-date information and documentation for NQC, visit the NQC Web Site at http://www.enteract.com/~dbaum/nqc

Lexical Considerations

Comments

Two forms of comments are supported in NQC. The first form (traditional C comments) begin with `/*` and end with `*/`. They may span multiple lines, but do not nest:

```
/* this is a comment */

/* this is a two
```

```
   line comment */

/* another comment...
   /* trying to nest...
      ending the inner comment...*/
   this text is no longer a comment! */
```

The second form of comments begins with `//` and ends with a newline (sometimes known as C++ style comments).

```
// a single line comment
```

Comments are ignored by the compiler. Their only purpose is to allow the programmer to document the source code.

Whitespace

Whitespace (spaces, tabs, and newlines) is used to separate tokens and to make programs more readable. As long as the tokens are distinguishable, adding or subtracting whitespace has no effect on the meaning of a program. For example, the following lines of code both have the same meaning:

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

Some of the C++ operators consist of multiple characters. In order to preserve these tokens whitespace must not be inserted within them. In the example below, the first line uses a right shift operator ('>>'), but in the second line the added space causes the '>' symbols to be interpreted as two separate tokens and thus generate an error.

```
x = 1 >> 4; // set x to 1 right shifted by 4 bits
x = 1 > > 4; // error
```

Numerical Constants

Numerical constants may be written in either decimal or hexidecimal form. Decimal constants consist of one or more decimal digits. Hexidecimal constants start with `0x` or `0X` followed by one or more hexidecimal digits.

```
x = 10;  // set x to 10
x = 0x10; // set x to 16 (10 hex)
```

Identifiers and Keywords

Identifiers are used for variable, task, and function names. The first character of an identifier must be an upper or lower case letter or the underscore ('_'). Remaining characters may be letters, numbers, an underscore.

A number of potential identifiers are reserved for use in the NQC language itself. These reserved words are call keywords and may not be used as identifiers. A complete list of keywords appears below:

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

Program Structure

An NQC program is composed of code blocks and global variables. There are three distinct types of code blocks: tasks, inline functions, and subroutines. Each type of code block has its own unique features and restrictions, but they all share a common structure.

Tasks

The RCX implicitly supports multi-tasking, thus an NQC task directly corresponds to an RCX task. Tasks are defined using the `task` keyword using the following syntax:

```
task name()
{
      // the task's code is placed here
}
```

The name of the task may be any legal identifier. A program must always have at least one task - named "main" - which is started whenever the program is run. A program may also contain up to 9 additional tasks.

The body of a task consists of a list of statements. Tasks may be started and stopped using the `start` and `stop` statements (described in the section titled *Statements*). There is also an RCX API command, `StopAllTasks`, which stops all currently running tasks.

Inline Functions

It is often helpful to group a set of statements together into a single function, which can then be called as needed. NQC supports functions with arguments, but not return values. Functions are defined using the following syntax:

```
void name(argument_list)
{
      // body of the function
}
```

The keyword `void` is an artifact of NQC's heritage - in C functions are specified with the type of data they return. Functions that do not return data are specified to return `void`. Returning data is not supported in NQC, thus all functions are declared using the `void` keyword.

The argument list may be empty, or may contain one or more argument definitions. An argument is defined by its *type* followed by its *name*. Multiple arguments are separated by commas. All values in the RCX are represented as 16 bit signed integers. However NQC supports four different argument types which correspond to different argument passing semantics and restrictions:

Type	**Meaning**	**Restriction**
int	pass by value	none
const int	pass by value	only constants may be used
int&	pass by reference	only variables may be used
const int &	pass by reference	function cannot modify argument

Arguments of type `int` are passed by value from the calling function to the callee. This usually means that the compiler must allocate a temporary variable to hold the argument. There are no restrictions on the type of value that may be used. However, since the function is working with a copy of the actual argument, any changes it makes to the value will not be seen by the caller. In the example below, the function `foo` attempts to set the value of its argument to 2. This is perfectly legal, but since `foo` is working on a copy of the original argument, the variable `y` from main task remains unchanged.

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

task main()
{
      int y = 1;  // y is now equal to 1
      foo(y);           // y is still equal to 1!
}
```

The second type of argument, `const int`, is also passed by value, but with the restriction that only constant values (e.g. numbers) may be used. This is rather important since there are a number of RCX functions that only work with constant arguments.

```
void foo(const int x)
{
```

```
        PlaySound(x);     // ok
        x = 1;      // error - cannot modify argument
    }

    task main()
    {

        foo(2);     // ok
        foo(4*5);   // ok - expression is still constant
        foo(x);     // error - x is not a constant
    }
```

The third type, `int &`, passes arguments by reference rather than by value. This allows the callee to modify the value and have those changes visible in the caller. However, only variables may be used when calling a function using `int &` arguments:

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

    task main()
    {
        int y = 1;  // y is equal to 1

        foo(y);     // y is now equal to 2
        foo(2);     // error - only variables allowed
    }
```

The last type, `const int &`, is rather unusual. It is also passed by reference, but with the restriction that the callee is not allowed to modify the value. Because of this restriction, the compiler is able to pass anything (not just variables) to functions using this type of argument. In general this is the most efficient way to pass arguments in NQC.

There is one important difference between `int` arguments and `const int &` arguments. An `int` argument is passed by value, so in the case of a dynamic expression (such as a sensor reading), the value is read once then saved. With `const int &` arguments, the expression will be re-read each time it is used in the function:

```
void foo(int x)
{
      if (x==x)  // this will always be true
            PlaySound(SOUND_CLICK);
}

void bar(const int x)
{
      if (x==x) // may not be true..value could change
            PlaySound(SOUND_CLICK);
}


task main()
{
      foo(SENSOR_1);    // will play sound
      bar(2);           // will play sound
      bar(SENSOR_1);    // may not play sound
}
```

Functions must be invoked with the correct number (and type) of arguments. The example below shows several different legal and illegal calls to function `foo`:

```
void foo(int bar, const int baz)
{
      // do something here...
}

task main()
{
      int x;      // declare variable x

      foo(1, 2);  // ok
      foo(x, 2);  // ok
      foo(2, x);  // error - 2nd argument not constant!
      foo(2);     // error - wrong number of arguments!
}
```

NQC functions are always expanded as inline functions. This means that each call to a function results in another copy of the function's code being included in the program. Unless used judiciously, inline functions can lead to excessive code size.

Subroutines

Unlike inline functions, subroutines allow a single copy of some code to be shared between several different callers. This makes subroutines much more space efficient than inline functions, but due to some limitations in the RCX, subroutines have some significant restrictions. First of all, subroutines cannot use any arguments. Second, a subroutine cannot call another subroutine. Last, a maximum of 8 subroutines may be defined in a program. In addition, if the subroutine is called from multiple tasks then it cannot have any local variables (or temporary variables). These significant restrictions make subroutines less desirable than inline functions, therefore their use should be minimized to those situations where the resultant savings in code size is absolutely necessary. The syntax for a subroutine appears below:

```
sub name()
{
      // body of subroutine
}
```

Variables

All variables in the RCX are of the same type - specifically 16 bit signed integers. The RCX supports up to 32 such variables. This pool of variables is utilized by NQC in several different ways. Variables are declared using the `int` keyword followed by a comma separated list of variable names and terminated by a semicolon (';'). Optionally, an initial value for each variable may be specified using an equals sign ('=') after the variable name. Several examples appear below:

```
int x;      // declare x
int y,z;    // declare y and z
int a=1,b;  // declare a and b, initialize a to 1
```

Global variables are declared at the program scope (outside any code block). Once declared, they may be used within all tasks, functions, and subroutines. Their scope begins at declaration and ends at the end of the program.

Local variables may be declared within tasks, functions, and sometimes within subroutines. Such variables are only accessible within the code block in which they are

defined. Specifically, their scope begins with their declaration and ends at the end of their code block. In the case of local variables, a compound statement (a group of statements bracketed by { and }) is considered a block:

```
int x;  // x is global

task main()
{
      int y;  // y is local to task main
      x = y; // ok
      {     // begin compound statement
            int z;  // local z declared
            y = z; // ok
      }
      y = z; // error - z no longer in scope
}

task foo()
{
      x = 1; // ok
      y = 2; // error - y is not global
}
```

In many cases NQC must allocate one or more temporary variables for its own use. In some cases a temporary variable is used to hold an intermediate value during a calculation. In other cases it is used to hold a value as it is passed to a function. These temporary variables deplete the pool of available variables in the RCX. NQC attempts to be as efficient as possible with temporary variables (including reusing them when possible).

Statements

The body of a code block (task, function, or subroutine) is composed of statements. Statements are terminated with a semi-colon (';').

Variable Delcaration

Variable declaration, as described in the previous section, is one type of statement. It declares a local variable (with optional initialization) for use within the code block. The syntax for a variable declaration is:

```
int variables;
```

where variables is a comma separated list of names with optional initial values:

```
name[=expression]
```

Assignment

Once declared, variables may be assigned the value of an expression:

```
variable assign_operator expression;
```

There are nine different assignment operators. The most basic operator, '=', simply assigns the value of the expression to the variable. The other operators modify the variable's value in some other way as shown in the table below

Operator	**Action**
=	Set variable to expression
+=	Add expression to variable
-=	Subtract expression from variable
*=	Multiple variable by expression
/=	Divide variable by expression
&=	Bitwise AND expression into variable
|=	Bitwise OR expression into variable
||=	Set variable to absolute value of expression
+-=	Set variable to sign (-1,+1,0) of expression

Some examples:

```
x = 2;      // set x to 2
y = 7;      // set y to 7
x += y;     // x is 9, y is still 7
```

Control Structures

The simplest control structure is a compound statement. This is a list of statements enclosed within curly braces ('{' and '}'):

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

Although this may not seem very significant, it plays a crucial role in building more complicated control structures. Many control structures expect a single statement as their body. By using a compound statement, the same control structure can be used to control multiple statements.

The `if` statement evaluates a condition. If the condition is true it executes one statement (the consequence). An optional second statement (the alternative) is executed if the condition is false. The two syntaxes for an `if` statement is shown below.

```
if (condition) consequence
if (condition) consequence else alternative
```

Note that the condition is enclosed in parentheses. Examples are shown below. Note how a compound statement is used in the last example to allow two statements to be executed as the consequence of the condition.

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

The `while` statement is used to construct a conditional loop. The condition is evaluated, and if true the body of the loop is executed, then the condition is tested again. This process continues until the condition becomes false (or a `break` statement is executed). The syntax for a `while` loop appears below:

```
while (condition) body
```

It is very common to use a compound statement as the body of a loop:

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

```
}
```

A variant of the `while` loop is the `do-while` loop. Its syntax is:

```
do body while (condition)
```

The difference between a `while` loop and a `do-while` loop is that the `do-while` loop always executes the body at least once, whereas the `while` loop may not execute it at all.

The `repeat` statement executes a loop a specified number of times:

```
repeat (expression) body
```

The expression determines how many times the body will be executed. Note that it is only evaluated a single time, then the body is repeated that number of times. This is different from both the `while` and `do-while` loops which evaluate their condition each time through the loop.

NQC also defines the `until` macro which provides a convenient alternative to the `while` loop. The actual definition of `until` is:

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

In other words, `until` will continue looping until the condition becomes true. It is most often used in conjunction with an empty body statement:

```
until(SENSOR_1 == 1);   // wait for sensor to be pressed
```

Other Statements

A function (or subroutine) call is a statement of the form:

```
name(arguments);
```

The arguments list is a comma separated list of expressions. The number and type of arguments supplied must match the definition of the function itself.

Tasks may be started or stopped with the following statements:

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

Within loops (such as a `while` loop) the `break` statement can be used to exit the loop and the `continue` statement can be used to skip to the top of the next iteration of the loop.

```
break;
continue;
```

It is possible to cause a function to return before it reaches the end of its code using the `return` statement.

```
return;
```

Any expression is also a legal statement when terminated by a semicolon. It is rare to use such a statement since the value of the expression would then be discarded. The one notable exception is expressions involving the increment (`++`) or decrement (`--`) operators.

```
x++;
```

The empty statement (just a bare semicolon) is also a legal statement.

Expressions and Conditions

In C there is no distinction between expressions and conditions. However, within NQC they are two syntactically different entities. The legal operations for expressions cannot be used on conditions and vice versa.

Expressions can be assigned to variables, used as arguments in a function, or as the count in a `repeat` statement. Conditions are used in most of the conditional control structures (`if`, `while`, etc.).

Expressions

Values are the most primitive type of expressions. More complicated expressions are formed from values using various operators. The NQC language only has two built in

kinds of values: numerical constants and variables. The RCX API defines other values corresponding to various RCX features such as sensors and timers.

Numerical constants in the RCX are represented as 16 bit signed integers. NQC internally uses 32 bit signed math for constant expression evaluation, then reduces to 16 bits when generating RCX code. Numeric constants can be written as either decimal (e.g. `123`) or hexidecimal (e.g. `0xABC`). Presently, there is very little range checking on constants, so using a value larger than expected may have unusual effects.

Values may be combined using operators. Several of the operators may only be used in evaluating constant expressions, which means that their operands must either be constants, or expressions involving nothing but constants. The operators are listed here in order of precedence (highest to lowest).

Operator	**Description**	**Associativity**	**Restriction**	**Example**
`abs()` `sign()`	Absolute value Sign of operand	n/a n/a		`abs(x)` `sign(x)`
`++` `--`	Increment Decrement	left left	variables only variables only	`x++ or ++x` `x-- or --x`
`-` `~`	Unary minus Bitwise negation (unary)	right right	constant only	`-x` `~123`

`*`
`/`
`%`
	Multiplication
	Division
	Modulo
		left
		left
		left
			constant only	`x * y`
				`x / y`
				`123 % 4`

`+` `-`	Addition Subtraction	left left		`x + y` `x - y`
`<<` `>>`	Left shift Right shift	left left	constant only constant only	`123 << 4` `123 >> 4`
`&`	Bitwise AND	left		`x & y`
`^`	Bitwise XOR	left	constant only	`123 ^ 4`

`|`	Bitwise OR	left		`x | y`
`&&`	Logical AND	left	constant only	`123 && 4`
`||`	Logical OR	left	constant only	`123 || 4`

Where needed, parentheses may be used to change the order of evaluation:

```
x = 2 + 3 * 4;    // set x to 14
y = (2 + 3) * 4;  // set y to 20
```

Conditions

Conditions are generally formed by comparing two expressions. There are also two constant conditions - `true` and `false` - which always evaluate to true or false respectively. A condition may be negated with the negation operator, or two conditions combined with the AND and OR operators. The table below summarizes the different types of conditions.

Condition	**Meaning**
`true`	always true
`false`	always false
`expr1 == expr2`	true if expr1 equals expr2
`expr1 != expr2`	true if expr1 is not equal to expr2
`expr1 < expr2`	true if one expr1 is less than expr2
`expr1 <= expr2`	true if expr1 is less than or equal to expr2
`expr1 > expr2`	true if expr1 is greater than expr2
`expr1 >= expr2`	true if expr1 is greater than or equal to expr2
`! condition`	logical negation of a condition - true if condition is false
`cond1 && cond2`	logical AND of two conditions (true if and only if both conditions are true)
`cond1 || cond2`	logical OR of two conditions (true if and only if at least one of the conditions are true)

The Preprocessor

The preprocessor implements the following directives: `#include`, `#define`, `#ifdef`, `#ifndef`, `#if`, `#elif`, `#else`, `#endif`, `#undef`. Its implementation is fairly close to a standard C preprocessor, so most things that work in a generic C preprocessor should have the expected effect in NQC. Significant deviations are listed below.

#include

The `#include` command works as expected, with the caveat that the filename must be enclosed in double quotes. There is no notion of a system include path, so enclosing a filename in angle brackets is forbidden.

```
#include "foo.nqh"  // ok

#include <foo.nqh> // error!
```

#define

The `#define` command is used for simple macro substitution. Redefinition of a macro is an error (unlike in C where it is a warning). Macros are normally terminated by the end of the line, but the newline may be escaped with the backslash ('\') to allow multi-line macros:

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

The `#undef` directive may be used to remove a macro's definition.

Conditional Compilation

Conditional compilation works similar to the C preprocessor. The following preprocessor directives may be used:

```
#if condition
#ifdef symbol
#ifndef symbol
#else
```

```
#elif condition
#endif
```

Conditions in `#if` directives use the same operators and precedence as in C. The `defined()` operator is supported.

Program Initialization

The compiler will insert a call to a special initialization function, `_init`, at the start of a program. This default function is part of the RCX API and sets all three outputs to full power in the forward direction (but still turned off). The initialization function can be disabled using `the #pragma noinit` directive:

```
#pragma noinit    // don't do any program initialization
```

The default initialization function can be replaced with a different function using the `#pragma init` directive.

```
#pragma init function   // use custom initialization
```

RCX API

The RCX API defines a set of constants, functions, values, and macros that provide access to features of the RCX such as sensors and outputs.

Sensors

The names `SENSOR_1`, `SENSOR_2`, and `SENSOR_3` are used to identify the RCX's sensor ports. Before a sensor's value can be read, it must be configured properly. A sensor has two different settings: its *type* and its *mode*. The type determines how the RCX reads the sensor electrically, while the mode determines how the sensors value is interpreted. For some sensors types only one mode makes sense, but for others (such as the temperature sensor) it can be read equally well in multiple modes (e.g. Fahrenheit and Celsius). The type and mode may be set using `SetSensorType`(*sensor*, *type*) and `SetSensorMode`(*sensor*, *mode*).

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

For convenience both the mode and type may be set using the `SetSensor`(*sensor*, *configuration*) command. This is the easiest and most common way to configure a sensor.

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

Valid constants for a sensor's type, mode, and configuration are given below.

Sensor Type	**Meaning**
SENSOR_TYPE_TOUCH	a touch sensor
SENSOR_TYPE_TEMPERATURE	a temperature sensor
SENSOR_TYPE_LIGHT	a light sensor
SENSOR_TYPE_ROTATION	a rotation sensor

Sensor Mode	**Meaning**
SENSOR_MODE_RAW	raw value from 0 to 1023
SENSOR_MODE_BOOL	boolean value (0 or 1)
SENSOR_MODE_EDGE	counts number of boolean transitions
SENSOR_MODE_PULSE	counts number of boolean periods
SENSOR_MODE_PERCENT	value from 0 to 100
SENSOR_MODE_FAHRENHEIT	degrees F
SENSOR_MODE_CELSIUS	degrees C

Sensor Configuration	**Type**	**Mode**
SENSOR_TOUCH	SENSOR_TYPE_TOUCH	SENSOR_MODE_BOOL
SENSOR_LIGHT	SENSOR_TYPE_LIGHT	SENSOR_MODE_PERCENT
SENSOR_ROTATION	SENSOR_TYPE_ROTATION	SENSOR_MODE_ROTATION
SENSOR_CELSIUS	SENSOR_TYPE_TEMPERATURE	SENSOR_MODE_CELSIUS
SENSOR_FAHRENHEIT	SENSOR_TYPE_TEMPERATURE	SENSOR_MODE_FAHRENHEIT
SENSOR_PULSE	SENSOR_TYPE_TOUCH	SENSOR_MODE_PULSE
SENSOR_EDGE	SENSOR_TYPE_TOUCH	SENSEO_MODE_EDGE

A sensor's value can be read by using its name within an condition. For example, the following code checks to see if the value of sensor 1 is greater than 20:

```
if (SENSOR_1 > 20)
```

```
        // do something...
```

Some sensor types (such as SENSOR_TYPE_ROTATION) allow you to reset the sensor's internal counter with the following command:

```
ClearSensor(expression sensor);
```

Outputs

The names `OUT_A`, `OUT_B`, and `OUT_C` are used to identify the RCX's three outputs. All of the commands to control outputs can work on multiple outputs at the same time. In order to specify more than one output for a command, add the names of the outputs together. For example, use "`OUT_A + OUT_B`" to specify outputs A and B together.

Each output has three different attributes: mode, direction, and power level. The mode can be set with the `SetOutput`(*outputs*, *mode*) command. The mode parameter should be one of the following constants:

Output Mode	**Meaning**
OUT_OFF	output is off (motor is prevented from turning)
OUT_ON	output is on (motor will be powered)
OUT_FLOAT	motor can "coast"

The other two attributes, direction and power level, may be set at any time, but only have an effect when the output is on. The direction is set with the `SetDirection`(*outputs*, *direction*) command. The direction parameter should be one of the following constants:

Direction	**Meaning**
OUT_FWD	Set to forward direction
OUT_REV	Set to reverse direction
OUT_TOGGLE	Switch direction to the opposite of what it is presently

The power level can range 0 (lowest) to 7 (highest). The names OUT_LOW, OUT_HALF, and OUT_FULL are defined for use in setting power level. The level is set using the `SetPower`(*outputs*, *power*) command.

Be default, all three motors are set to full power and the forward direction (but still turned off) when a program starts.

Since control of outputs is such a common feature of programs, a number of convenience functions are provided that make it easier to work with the outputs. It should be noted that these commands do not provide any new functionality above the `SetOutput` and `SetDirection` commands. They are merely convenient ways to make programs more concise.

Command	**Action**
On(outputs)	turns motors on
Off(outputs)	turns motors off
Float(outputs)	makes outputs "float"
Fwd(outputs)	sets outputs to forward direction
Rev(outputs)	sets outputs to reverse direction
Toggle(outputs)	toggles direction of outputs
OnFwd(outputs)	sets direction to forward, then turns on
OnRev(outputs)	sets direction to reverse, then turns on
OnFor(outputs, time)	turns outputs on for specified amount of time (in 100ths of a second)

Some examples of using the output commands are shown below:

```
OnFwd(OUT_A);     // turn on A in the forward direction
OnRev(OUT_B);     // turn on B in the reverse direction
Toggle(OUT_A + OUT_B);  // flip directions of A and B
Off(OUT_A + OUT_B);     // turn off A and B
OnFor(OUT_C, 100);      // turn on C for 1 second
```

All of the output functions require constants for their arguments with the following exceptions:

`OnPower` - an expression may be used for the power level

`OnFor` - and expression may be used for the time

Miscellaneous

`Wait`(*time*) - Make a task sleep for specified amount of time (in 100ths of a second). The time argument may be an expression or a constant:

```
Wait(100);  // wait 1 second
```

```
Wait(Random(100)); // wait random time up to 1 second
```

`PlaySound`(*sound*) - Play one of the 6 preset RCX sounds. The sound argument must be a constant. The following constants are pre-defined for use with `PlaySound`: `SOUND_CLICK`, `SOUND_DOUBLE_BEEP`, `SOUND_DOWN`, `SOUND_UP`, `SOUND_LOW_BEEP`, `SOUND_FAST_UP`.

```
PlaySound(SOUND_CLICK);
```

`PlayTone`(*frequency*, *duration*) - Play a single tone of the specified frequency and duration. Both arguments must be constant. The frequency is in Hz, the duration is in 100ths of a second.

```
PlayTone(440, 50);      // Play 'A' for one half second
```

`SelectDisplay`(*mode*) - Select which mode the LCD display should use. There are seven different display modes as shown below. The RCX defaults to `DISPLAY_WATCH`.

Mode	**LCD Contents**
DISPLAY_WATCH	show the system "watch"
DISPLAY_SENSOR_1	show value of sensor 1
DISPLAY_SENSOR_2	show value of sensor 2
DISPLAY_SENSOR_3	show value of sensor 3
DISPLAY_OUT_A	show setting for output A
DISPLAY_OUT_B	show setting for output B
DISPLAY_OUT_C	show setting for output C

```
SelectDisplay(DISPLAY_SENSOR_1);    // view sensor 1
```

`ClearMessage()` - Clear the message buffer for RCX to RCX communication. This allows detection of the next received IR message. The `Message()` expression may be used to read the current contents of the receive buffer.

```
ClearMessage();   // clear out the received message
until(Message() > 0);   // wait for next message
```

`SendMessage`(*message*) - Send an IR message to another RCX. 'message' may be any expression, but the RCX can only send messages with a value between 0 and 255, so only the lowest 8 bits of the argument are used.

```
SendMessage(3);   // send message 3
SendMessage(259); // another way to send message 3
```

`SetWatch`(*hours*, *minutes*) - Set the system watch to the specified number of hours and minutes. Hours must be a constant between 0 and 23 inclusive. Minutes must be a constant between 0 and 59 inclusive.

```
SetWatch(3, 15);  // set watch to 3:15
```

`ClearTimer`(*timer*) - Clear one of the RCX's four internal timers. The timer parameter must be a constant between 0 and 3 inclusive.

```
ClearTimer(0);    // clear the first RCX timer
```

`StopAllTasks`() - Stop all currently running tasks. This will halt the program completely, so any code following this command will be ignored.

```
StopAllTasks();   // stop the program
```

`SetTxPower`(power) - Set the power level for the RCX's IR transmitter. The power level should be either `TX_POWER_LO` or `TX_POWER_HI`.

```
SetTxPower(TX_POWER_LO);      // set IR to lower power
```

Data Logging

The RCX contains a datalog which can be used to store readings from sensors, timers, variables, and the system watch. Before adding data, the datalog first needs to be created

using the `CreateDatalog`(*size*) command. The 'size' parameter must be a constant and determines how many data points the datalog can hold.

```
CreateDatalog(100);     // datalog for 100 points
```

Values can then be added to the datalog using `AddToDatalog`(*value*). When the datalog is uploaded to a computer it will show both the value itself and the source of the value (timer, variable, etc). The datalog directly supports the following data sources: timers, sensor values, variables, and the system watch. Other data types (such as a constant or random number) may also be logged, but in this case NQC will first move the value into a variable and then log the variable. The values will still be captured faithfully in the datalog, but the sources of the data may be a bit misleading.

```
AddToDatalog(Timer(0)); // add timer 0 to datalog
AddToDatalog(x);  // add variable 'x'
AddToDatalog(7);  // add 7 - will look like a variable
```

The RCX itself cannot read values back out of the datalog. The datalog must be uploaded to a host computer . The specifics of uploading the datalog depend on the NQC environment being used. For example, in the command line version of NQC, the following commands will upload and print the datalog:

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

Values

The RCX has several different sources of data: sensors, variables, timers, etc. Within NQC, these data sources can be accessed using special expressions.

The RCX contains four timers which measure time in increments of a tenth of a second (100ms). To read the value of a timer, use the `Timer(n)` expression where `n` is a constant between 0 and 3 and specifies which timer to read.

Sensors have a number of different values associated with them. The sensor's raw value, its boolean value, or its normal value can be read. In addition, it is possible for a program to read a sensor's configured type and mode. All of the expressions for sensor values require an argument specifying which sensor to use. This argument should be between 0

and 2. Note that the names `SENSOR_1`, `SENSOR_2`, and `SENSOR_3` are really just macros for `SensorValue(n)` expressions:

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

The RCX API also provides expressions to generate a random number, read the system watch, or read the last received IR message. All of the RCX API expressions are summarized below:

Expression	**Meaning**
`Timer(n)`	value of timer n
`Random(n)`	random number between 0 and n
`Watch()`	value of system watch (time in minutes)
`Message()`	value of last IR message received
`SensorValue(n)`	value of sensor n
`SensorType(n)`	type of sensor n
`SensorMode(n)`	mode of sensor n
`SensorValueRaw(n)`	raw value of sensor n
`SensorValueBool(n)`	boolean value of sensor n

Appendix A - Quick Reference

Statements

Statment	**Description**
`while (`*`cond`*`)` *`body`*	Execute body zero or more times while condition is true
`do` *`body`* `while (`*`cond`*`)`	Execute body one or more times while condition is true
`until (`*`cond`*`)` *`body`*	Execute body zero or more times until condition is true
`break`	Break out from while/do/until body
`continue`	Skip to next iteration of while/do/until body
`repeat (`*`expression`*`)` *`body`*	Repeat body a specified number of times
`if (`*`cond`*`)` *`stmt1`* `if (`*`cond`*`)` *`stmt1`* `else` *`stmt2`*	Execute stmt1 if condition is true. Execute stmt2 (if present) if condition is false.
`start` *`task_name`*	Start the specified task
`stop` *`task_name`*	Stop the specified task
`function``(`*`args`*`)`	Call a function using the supplied arguments
`var = expression`	Evaluate expression and assign to variable
`var += expression`	Evaluate expression and add to variable
`var -= expression`	Evaluate expression and subtract from variable
`var *= expression`	Evaluate expression and multiply into variable
`var /= expression`	Evaluate expression and divide into variable
`var |= expression`	Evaluate expression and perform bitwise OR into variable
`var &= expression`	Evaluate expression and perform bitwise AND into variable
`return`	Return from function to the caller
`expression`	Evaluate expression

Conditions

Conditions are used within control statements to make decisions. In most cases, the condition will involve a comparison between expressions.

Condition	**Meaning**
`true`	always true
`false`	always false
`expr1 == expr2`	true if expressions are equal
`expr1 != expr2`	true if expressions are not equal
`expr1 < expr2`	true if expr1 is less than expr2
`expr1 <= expr2`	true if expr1 is less than or equal to expr2
`expr1 > expr2`	true if expr1 is greater than expr2
`expr1 >= expr2`	true if expr1 is greater than or equal to expr2
`!` *`condition`*	logical negation of a condition
`cond1 && cond2`	logical AND of two conditions (true if and only if both conditions are true)
`cond1 || cond2`	logical OR of two conditions (true if and only if at least one of the conditions is true)

Expressions

There are a number of different values that can be used within expressions including constants, variables, and sensor values. Note that SENSOR_1, SENSOR_2, and SENSOR_3 are macros that expand to SensorValue(0), SensorValue(1), and SensorValue(2) respectively.

Value	**Description**
`number`	A constant value (e.g. 123)
`variable`	A named variable (e.g x)
`Timer(`*`n`*`)`	Value of timer n, where n is between 0 and 3
`Random(`*`n`*`)`	Random number between 0 and n

`SensorValue(`*n*`)`	Current value of sensor n, where n is between 0 and 2
`Watch()`	Value of system watch
`Message()`	Value of last received IR message

Values may be combined by using operators. Several of the operators may only be used in evaluating constant expressions, which means that their operands must be either constants or expressions involving nothing but constants. The operators are listed here in order of precedence (highest to lowest).

Operator	**Description**	**Associativity**	**Restriction**	**Example**
`abs()` `sign()`	Absolute value Sign of operand	n/a n/a	none none	`abs(x)` `sign(x)`
`++` `--`	Increment Decrement	left left	variables only variables only	`x++ or ++x` `x-- or --x`
`-` `~`	Unary minus Bitwise negation (unary)	right right	none constant only	`-x` `~123`

`*`
`/`
`%`
	Multiplication
	Division
	Modulo
		left
		left
		left
			none
			none
			constant only
				`x * y`
				`x / y`
				`123 % 4`

`+` `-`	Addition Subtraction	left left	none none	`x + y` `x - y`
`<<` `>>`	Left shift Right shift	left left	constant only constant only	`123 << 4` `123 >> 4`
`&`	Bitwise AND	left	none	`x & y`
`^`	Bitwise XOR	left	constant only	`123 ^ 4`
`|`	Bitwise OR	left	none	`x | y`
`&&`	Logical AND	left	constant only	`123 && 4`
`||`	Logical OR	left	constant only	`123 || 4`

RCX Functions

Most of the functions require all arguments to be constant expressions (numbers or operations involving other constant expressions). The exceptions are functions that use a sensor as an argument and those that can use any expression. In the case of sensors, the argument should be a sensor name: SENSOR_1, SENSOR_2, or SENSOR_3. In some cases there are predefined names (e.g. SENSOR_TOUCH) for appropriate constants.

Function	**Description**	**Example**
`SetSensor(`*`sensor`*`, `*`config`*`)`	Configure a sensor.	`SetSensor(SENSOR_1, SENSOR_TOUCH)`
`SetSensorMode(`*`sensor`*`, `*`mode`*`)`	Set sensor's mode	`SetSensor(SENSOR_2, SENSOR_MODE_PERCENT)`
`SetSensorType(`*`sensor`*`, `*`type`*`)`	Set sensor's type	`SetSensor(SENSOR_2, SENSOR_TYPE_LIGHT)`
`ClearSensor(`*`sensor`*`)`	Clear a sensor's value	`ClearSensor(SENSOR_3)`
`On(`*`outputs`*`)`	Turn on one or more outputs	`On(OUT_A + OUT_B)`
`Off(`*`outputs`*`)`	Turn off one or more outputs	`Off(OUT_C)`
`Float(`*`outputs`*`)`	Let the outputs float	`Float(OUT_B)`
`Fwd(`*`outputs`*`)`	Set outputs to forward direction	`Fwd(OUT_A)`
`Rev(`*`outputs`*`)`	Set outputs to reverse direction	`Rev(OUT_B)`
`Toggle(`*`outputs`*`)`	Flip the direction of outputs	`Toggle(OUT_C)`
`OnFwd(`*`outputs`*`)`	Turn on in forward direction	`OnFwd(OUT_A)`
`OnRev(`*`outputs`*`)`	Turn on in reverse direction	`OnRev(OUT_B)`
`OnFor(`*`outputs`*`, `*`time`*`)`	Turn on for specified number of 100ths of a second. Time may be an expression.	`OnFor(OUT_A, x)`
`SetOutput(`*`outputs`*`, `*`mode`*`)`	Set output mode	`SetOutput(OUT_A, OUT_ON)`
`SetDirection(`*`outputs`*`, `*`dir`*`)`	Set output direction	`SetDirection(OUT_A, OUT_FWD)`
`SetPower(`*`outputs`*`, `*`power`*`)`	Set output power level (0-7).	`SetPower(OUT_A, x)`

	Power may be an expression.	
`Wait(`*`time`*`)`	Wait for the specified amount of time in 100ths of a second. Time may be an expression.	`Wait(x)`
`PlaySound(`*`sound`*`)`	Play the specified sound (0-5).	`PlaySound(SOUND_CLICK)`
`PlayTone(`*`freq, duration`*`)`	Play a tone of the specified frequency for the specified amount of time (in 10ths of a second)	`PlayTone(440, 5)`
`ClearTimer(`*`timer`*`)`	Reset timer (0-3) to value 0	`ClearTimer(0)`
`StopAllTasks()`	Stop all currently running tasks	`StopAllTasks()`
`SelectDisplay(`*`mode`*`)`	Select one of 7 display modes: 0: system watch, 1-3: sensor value, 4-6: output setting. Mode may be an expression.	`SelectDisplay(1)`
`SendMessage(`*`message`*`)`	Send an IR message (1-255). Message may be an expression.	`SendMessage(x)`
`ClearMessage()`	Clear the IR message buffer	`ClearMessage()`
`CreateDatalog(`*`size`*`)`	Create a new datalog of the given size	`CreateDatalog(100)`
`AddToDatalog(`*`value`*`)`	Add a value to the datalog. The value may be an expression.	`AddToDatalog(Timer(0))`
`SetWatch(`*`hours, minutes`*`)`	Set the system watch value	`SetWatch(1,30)`
`SetTxPower(`*`hi_lo`*`)`	Set the infrared transmitter power level to low or high power	`SetTxPower(TX_POWER_LO)`

RCX Constants

Many of the values for RCX functions have named constants that can help make code more readable. Where possible, use a named constant rather than a raw value.

Sensor configurations for SetSensor()	SENSOR_TOUCH, SENSOR_LIGHT, SENSOR_ROTATION, SENSOR_CELSIUS, SENSOR_FAHRENHEIT, SENSOR_PULSE, SENSOR_EDGE
Modes for SetSensorMode()	SENSOR_MODE_RAW, SENSOR_MODE_BOOL, SENSOR_MODE_EDGE, SENSOR_MODE_PULSE, SENSOR_MODE_PERCENT, SENSOR_MODE_CELSIUS, SENSOR_MODE_FAHRENHEIT, SENSOR_MODE_ROTATION
Types for SetSensorType()	SENSOR_TYPE_TOUCH, SENSOR_TYPE_TEMPERATURE, SENSOR_TYPE_LIGHT, SENSOR_TYPE_ROTATION
Outputs for On(), Off(), etc.	OUT_A, OUT_B, OUT_C
Modes for SetOutput()	OUT_ON, OUT_OFF, OUT_FLOAT
Directions for SetDirection()	OUT_FWD, OUT_REV, OUT_TOGGLE
Output power for SetPower()	OUT_LOW, OUT_HALF, OUT_FULL
Sounds for PlaySound()	SOUND_CLICK, SOUND_DOUBLE_BEEP, SOUND_DOWN, SOUND_UP, SOUND_LOW_BEEP, SOUND_FAST_UP
Modes for SelectDisplay()	DISPLAY_WATCH, DISPLAY_SENSOR_1, DISPLAY_SENSOR_2, DISPLAY_SENSOR_3, DISPLAY_OUT_A, DISPLAY_OUT_B, DISPLAY_OUT_C
Tx power level for SetTxPower()	TX_POWER_LO, TX_POWER_HI

Keywords

Keywords are those words reserved by the NQC compiler for the language itself. It is an error to use any of these as the names of functions, tasks, or variables.

```
__sensor        const           false
abs             continue        if
asm             do              inline
break           else            int
```

repeat	stop	void
return	sub	while
sign	task	
start	true	

Appendix B - rcx.nqh

The rcx.nqc file is a special file internal to the NQC compiler which defines the RCX API. A complete listing of rcx.nqh is provided below for reference. Note that because of line wrapping issues in this document, the actual rcx.nqh file included with the NQC release should be viewed directly.

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

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


// these are the standard system definitions for 2.0

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

// constants for selecting sensors
```

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

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

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


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

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

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

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

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

#endif



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

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

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

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

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

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

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

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

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

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


#endif



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

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

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

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

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


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

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

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

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

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


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

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

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

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

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

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

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

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

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

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

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

NQC User Manual

Version 2.0, written by Dave Baum

Introduction

NQC stands for Not Quite C, and is a simple language for programming the LEGO RCX. The preprocessor and control structures of NQC are very similar to C. NQC is not a general purpose language - there are many restrictions that stem from limitations of the standard RCX firmware.

This document describes how to use the command line version of the NQC compiler. For information about the NQC language and/or the RCX API refer to the *NQC Programmer's Gudie*.

For up-to-date information and documentation for NQC, visit the NQC Web Site at http://www.enteract.com/~dbaum/nqc

Usage

Invoking NQC without any argument will print the version number and usage information:

```
nqc version 2.0
     Copyright (C) 1998,1999 David Baum.  All Rights Reserved.
Usage: nqc [options] [actions] [ - | filename ] [actions]
   - : read from stdin instead of a source_file
Options:
   -1: use NQC 1.x compatability mode
   -c: target CyberMaster instead of RCX
   -d: download program
   -n: prevent the system file (rcx.nqh) from being included
   -D<sym>[=<value>] : define macro <sym>
   -E[<filename>] : write compiler errors to <filename> (or stdout)
   -I<path>: search <path> for include files
   -L[<filename>] : generate code listing to <filename> (or stdout)
   -O<outfile>: specify output file
   -S<portname>: specify serial port
   -U<sym>: undefine macro <sym>
Actions:
   -run: run current program
   -pgm <number>: select program number
   -datalog | -datalog_full: upload datalog
   -near : set IR to near mode
   -far : set IR to far mode
   -watch <time> | now : set RCX time
```

```
-firmware <filename> : download firmware
-sleep <timeout> : set RCX sleep timeout
-msg <number> : send IR message to RCX
-raw <data> : format data as a packet and send to RCX
-clear : erase all programs and datalong in RCX
```

The most typical use of NQC is to compile a source program and download it to the RCX:

```
nqc –d your_file
```

Source and Output Files

NQC will only accept a single filename for compilation. If '`-`' is used instead of a filename, stdin is compiled. If a filename ends in `.rcx`, then it is assumed to be an RCX image file (from a previous compile) and will be used as is (for downloading or listing).

NQC will automatically create an output file for you if all of the following conditions are met:

- You are compiling a file (not stdin)
- Neither `-d` or `-L` have been specified
- `-O` has not been used.

In this case the output file is named the same as the input file, but with an extension of "`.rcx`" instead of "`.nqc`".

If the `-O` option is used and a source file (or stdin) is compiled, then the output file is generated even if `-d` or `-L` are used.

Preprocessor Options

NQC always searches the current directory for include files. Additional directories may be specified using either the NQC_INCLUDE environment variable or the `–I` option. Multiple `–I` options may be specified. Searching is done in the following order:

- Current Directory
- Directory specified by NQC_INCLUDE (if any)

• Directories specified by `-I` options (if any)

When specifying additional include directories, you must include the proper directory separator, for example under Windows you might do something like this to add C:\NQC to the include search path:

```
nqc -IC:\NQC\ foo.nqc
```

Most of the RCX API is defined in a special include file named "`rcx.nqh`". This file is stored within the compiler itself and is usually included automatically before any compilation. The `-n` option disables this feature and compiles the source file without the RCX API being defined.

Two other preprocessor options are available:

`-D`*symbol*[=*value*] defines symbol in the preprocessor

`-U`*symbol* removes symbol's definition

Serial Ports

The default serial port used by NQC is system dependent:

• Windows: COM1

• Macintosh: modem port

• Linux: /dev/ttyS0

This default can be overridden either by setting the system environment variable RCX_PORT, or by using the `-S` option on the command line. If both are specified, the command line option takes precedence.

Miscellaneous Options

The `-c` option makes NQC compatible with CyberMaster. This has several effects:

• Defines the compile time symbol `__CM` (instead of `__RCX`)

• Targets CyberMaster during compilation (different task/subroutine limits, etc.)

• Uses the CyberMaster serial protocol for downloading instead of the

The `-d` option can be used to automatically download the compiled file to the RCX.

Compiler error messages are normally reported to stderr. If you want to redirect these messages, use the `-E` option. If you specify a filename (e.g. `-Eerror_file`) the messages will be redirected into a file. If no filename is specified (e.g. `-E`) then the errors will be placed on stdout instead of stderr.

A program listing may be generated by using the `-L` option. If a filename is specified (e.g. `-Llist_file`), then the listing is written to the named file. Otherwise the listing is written to stdout.

The `-1` option make NQC compatible with version 1.3 of the NQC compiler. This includes some changes in syntax as well as the 1.3 APIs for the RCX.

Actions

Actions look similar to options, but they have some subtle differences. In general, options setup things (such as a serial port) for later use, while actions cause something to happen. Actions are executed in the order that they appear on the command line. In addition, actions appearing before the source file happen before compilation, while actions after the source file happen after compilation. For historical reasons, downloading the compiled file is treated as an option and not an action.

For example, if you want to compile and download foo.nqc to program slot #2 and run it after downloading, you'd use the following command:

```
nqc -d -pgm 2 foo.nqc -run
```

Note that the `-d` option appears before any actions. The `-pgm` action must appear before the source file, otherwise the program change would happen after the program was downloaded. Finally, the `-run` action appears at the end of the line so that the program is started after download is complete

A complete list of appears below:

Action	**Meaning**
`-run`	Run the currently selected program.
`-pgm` *number*	Select a program number on the RCX.
`-datalog`	Upload the RCX's datalog and print it to stdout
`-datalog_full`	Same as `-datalog`, but with more verbose output
`-near`	Set the RCX's IR mode to short-range
`-far`	set the RCX's IR mode to long-range
`-watch` *time*	Set the RCX's clock to the specified time. If "now" is specified, then the host's current time is used.
`-firmware` *file*	Download a firmware file to the RCX.
`-sleep` *timeout*	Set the auto shutoff timeout (in minutes) for the RCX.
`-msg` *number*	Send a message to the RCX. This is useful if you write a program that allows external control via IR messages.
`-raw` *data*	Send an arbitrary message to the RCX and print the reply (if any) to stdout. The data should be a hexidecimal string, with no spaces, a zero-padded so that it is an even number of characters (although it may be an odd number of bytes). For example, to read the contents of variable 1, you could use `-raw 120001`. The bytecodes for raw messages can be found on web sites that document the RCX protocol.
`-clear`	Erase all programs and any datalog present in the RCX.

Programming Lego Robots using NQC

(Version 3.03, Oct 2, 1999)

by Mark Overmars

Department of Computer Science
Utrecht University
P.O. Box 80.089, 3508 TB Utrecht
the Netherlands

Preface

The Lego MindStorms and CyberMaster robots are wonderful new toys from which a wide variety of robots can be constructed, that can be programmed to do all sorts of complicated tasks. Unfortunately, the software that comes with the robots is, although visually attractive, rather limited in functionality. Hence, it can only be used for simple tasks. To unleash the full power of the robots, you need a different programming environment. NQC is a programming language, written by Dave Baum, that was especially designed for the Lego robots. If you have never written a program before, don't worry. NQC is really easy to use and this tutorial will tell you all about it. Actually, programming the robots in NQC is a lot easier than programming a normal computer, so this is a chance to become a programmer in an easy way.

To make writing programs even easier, there is the RCX Command Center. This utility helps you to write your programs, to send them to the robot, and to start and stop the robot. RCX Command Center works almost like a text processor, but with some extras. This tutorial will use RCX Command Center (version 3.0 or higher) as programming environment. You can download it for free from the web at the address

http://www.cs.uu.nl/people/markov/lego/

RCX Command Center runs on Windows PC's ('95, '98, NT). (Make sure that you ran the software that comes with the Lego set at least once, before using RCX Command Center. The Lego software installs certain components that RCX Command Center uses.) The language NQC can also be used on other platforms. You can download it from the web at address

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

Most of this tutorial also applies to the other platforms (assuming you use NQC version 2.0 or higher), except that you loose some of the tools and the color-coding.

In this tutorial I assume that you have the MindStorms robot. Most of the contents also applies to the CyberMaster robots although some of the functionality is not available for those robots. Also the names of e.g. the motors are different so you will have to change the examples a little bit to make them work.

Acknowledgements

I would like to thank Dave Baum for developing NQC. Also many thanks to Kevin Saddi for writing a first version of the first part of this tutorial.

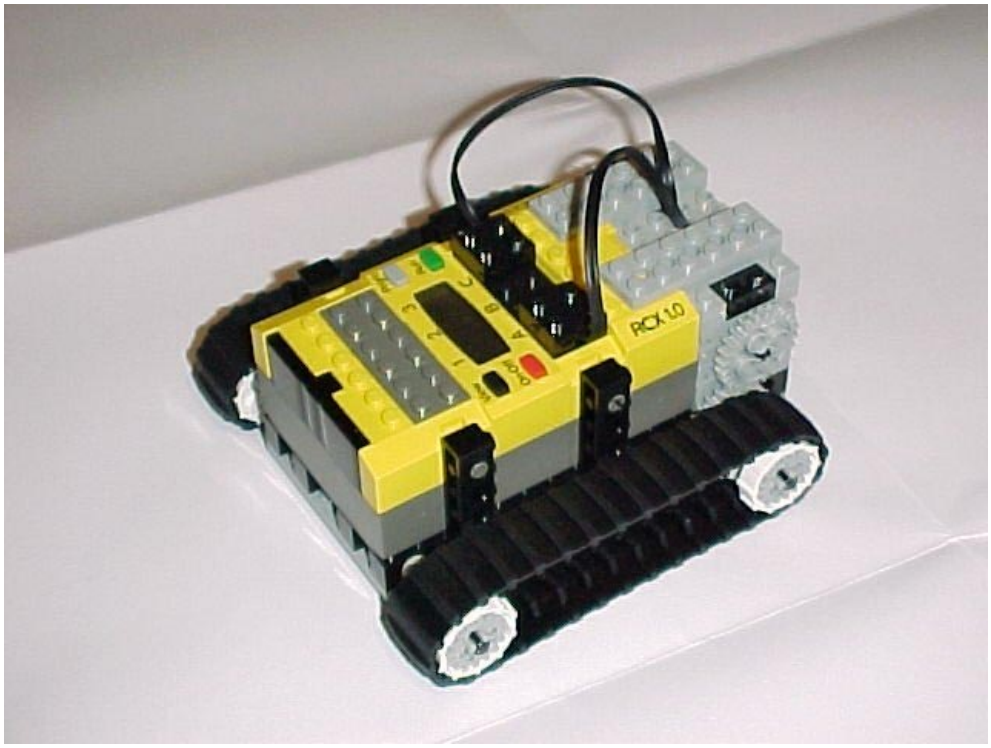
Contents

I. Writing your first program

In this chapter I will show you how to write an extremely simple program. We are going to program a robot to move forwards for 4 seconds, then backwards for another 4 seconds, and then stop. Not very spectacular but it will introduce you to the basic idea of programming. And it will show you how easy this is. But before we can write a program, we first need a robot.

Building a robot

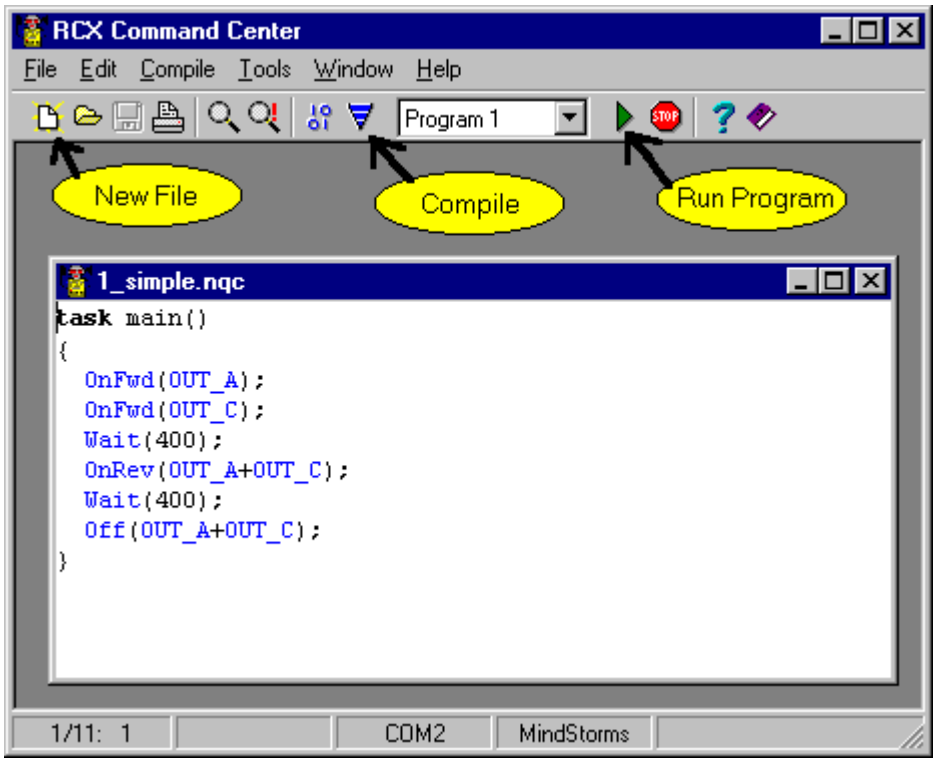
The robot we will use throughout this tutorial is a simple version of the top-secret robot that is described on page 39-46 of your constructopedia. We will only use the basis chassis. Remove the whole front with the two arms and the touch sensors. Also, connect the motors slightly different such that the wires are connected to the RCX at the outside. This is important for your robot to drive in the correct direction. Your robot should look like this:



Also make sure that the infra-red port is correctly connected to your computer and that it is set to long range. (You might want to check with the RIS software that the robot is functioning well.)

Starting RCX Command Center

We write our programs using RCX Command Center. Start it by double clicking on the icon RcxCC. (I assume you already installed RCX Command Center. If not, download it from the web site (see the preface), unzip it, and place it in any directory you like.) The program will ask you where to locate the robot. Switch the robot on and press **OK**. The program will (most likely) automatically find the robot. Now the user interface appears as shown below (without a window).



The interface looks like a standard text editor, with the usual menu's, and buttons to open and save files, print files, edit files, etc. But there are also some special menus for compiling and downloading programs to the robot and for getting information from the robot. You can ignore these for the moment.

We are going to write a new program. So press the **New File** button to create a new, empty window.

Writing the program

Now type in the following program:

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

It might look a bit complicated at first, so let us analyze it. Programs in NQC consist of tasks. Our program has just one task, named `main`. Each program needs to have a task called `main` which is the one that will be executed by the robot. You will learn more about tasks in Chapter VI. A task consists of a number of commands, also called statements. There are brackets around the statements such that it is clear that they all belong to this task. Each statement ends with a semicolon. In this way it is clear where a statement ends and where the next statement begins. So a task looks in general as follows:

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

Our program has six statements. Let us look at them one at the time:

```
OnFwd(OUT_A);
```
This statement tells the robot to start output A, that is, the motor connected to the output labeled A on the RCX, to move forwards. It will move with maximal speed, unless you first set the speed. We will see later how to do this.

```
OnFwd(OUT_C);
```
Same statement but now we start motor C. After these two statement, both motors are running, and the robot moves forwards.

```
Wait(400);
```
Now it is time to wait for a while. This statement tells us to wait for 4 seconds. The argument, that is, the number between the parentheses, gives the number of "ticks". Each tick is 1/100 of a second. So you can very precisely tell the program how long to wait. So for 4 seconds, the program does do nothing and the robot continues to move forwards.

```
OnRev(OUT_A+OUT_C);
```
The robot has now moved far enough so we tell it to move in reverse direction, that is, backwards. Note that we can set both motors at once using `OUT_A+OUT_C` as argument. We could also have combined the first two statements this way.

```
Wait(400);
```
Again we wait for 4 seconds.

```
Off(OUT_A+OUT_C);
```
And finally we switch both motors off.

That is the whole program. It moves both motors forwards for 4 seconds, then backwards for 4 seconds, and finally switches them off.

You probably noticed the colors when typing in the program. They appear automatically. Everything in blue is a command for the robot, or an indication of a motor or other thing that the robot knows about. The word **`task`** is in bold because it is an important (reserved) word in NQC. Other important words appear in bold as well as we will see later. The colors are useful to see that you did not make any errors while typing.

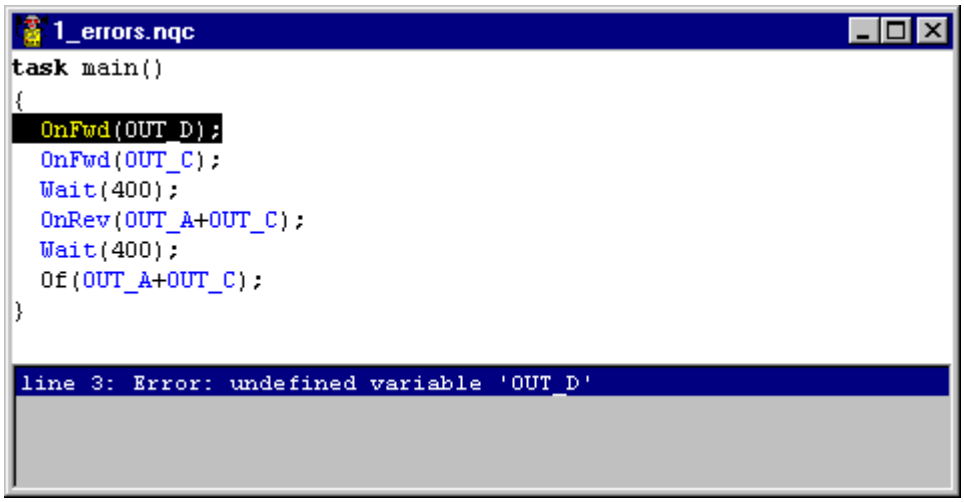
Running the program

Once you have written a program, it needs to be compiled (that is, changed into code that the robot can understand and execute) and send to the robot using the infra red link (called "downloading" the program). There is a button that does both at once (see the figure above). Press this button and, assuming you made no errors when typing in the program, it will correctly compile and be downloaded. (If there are errors in your program you will be notified; see below.)

Now you can run your program. To this end press the green run button on your robot or, more easily, press the run button on your window (see the figure above). Does the robot do what you expected? If not, the wires are probably connected wrong.

Errors in your program

When typing in programs there is a reasonable chance that you make some errors. The compiler notices the errors and reports them to you at the bottom of the window, like in the following figure:



It automatically selects the first error (we mistyped the name of the motor). When there are more errors, you can click on the error messages to go to them. Note that often errors at the beginning of the program cause other errors at other places. So better only correct the first few errors and then compile the program again. Also note that the color-coding helps a lot in avoiding errors. For example, on the last line we typed `Of` rather than `Off`. Because this is an unknown command it is not colored blue.

There are also errors that are not found by the compiler. If we had typed `OUT_B` this would have gone unnoticed because that motor exists (even though we do not use it in the robot). So if the robot exhibits unexpected behavior, there is most likely something wrong in your program.

Changing the speed

As you noticed, the robot moved rather fast. Default the robot moves as fast as it can. To change the speed you can use the command `SetPower()`. The power is a number between 0 and 7. 7 is the fastest, 0 the slowest (but the robot will still move). Here is a new version of our program in which the robot moves slow:

```
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);
}
```

Summary

In this chapter you wrote your first program in NQC, using RCX Command Center. You should now know how to type in a program, how to download it to the robot and how to let the robot execute the program. RCX Command Center can do many more things. To find out about them, read the documentation that comes with it. This tutorial will primarily deal with the language NQC and only mention features of RCX Command Center when you really need them.

You also learned some important aspects of the language NQC. First of all, you learned that each program has one task named `main` that is always executed by the robot. Also you learned the four most important motor commands: `OnFwd()`, `OnRev()`, `SetPower()` and `Off()`. Finally, you learned about the `Wait()` statement.

II. A more interesting program

Our first program was not very spectacular. So let us try to make it more interesting. We will do this in a number of steps, introducing some important features of our programming language NQC.

Making turns

You can make your robot turn by stopping or reversing the direction of one of the two motors. Here is an example. Type it in, download it to your robot and let it run. It should drive a bit and then make a 90-degree right turn.

```
task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(100);
  OnRev(OUT_C);
  Wait(85);
  Off(OUT_A+OUT_C);
}
```

You might have to try some slightly different numbers than 85 in the second `Wait()` command to make a precise 90-degree turn. This depends on the type of surface on which the robot runs. Rather than changing this in the program it is easier to use a name for this number. In NQC you can define constant values as shown in the following program.

```
#define MOVE_TIME   100
#define TURN_TIME    85

task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(MOVE_TIME);
  OnRev(OUT_C);
  Wait(TURN_TIME);
  Off(OUT_A+OUT_C);
}
```

The first two lines define two constants. These can now be used throughout the program. Defining constants is good for two reasons: it makes the program more readable, and it is easier to change the values. Note that RCX Command Center gives the define statements its own color. As we will see in Chapter VI, you can also define things other than constants.

Repeating commands

Let us now try to write a program that makes the robot drive in a square. Going in a square means: driving forwards, turning 90 degrees, driving forwards again, turning 90 degrees, etc. We could repeat the above piece of code four times but this can be done a lot easier with the **`repeat`** statement.

```
#define MOVE_TIME   100
#define TURN_TIME    85

task main()
{
  repeat(4)
  {
    OnFwd(OUT_A+OUT_C);
    Wait(MOVE_TIME);
    OnRev(OUT_C);
    Wait(TURN_TIME);
  }
  Off(OUT_A+OUT_C);
}
```

The number behind the **repeat** statement, between parentheses, indicates how often something must be repeated. The statements that must be repeated are put between brackets, just like the statements in a task. Note that, in the above program, we also indent the statements. This is not necessary, but it makes the program more readable.

As a final example, let us make the robot drive 10 times in a square. Here is the program:

```
#define MOVE_TIME   100
#define TURN_TIME    85

task main()
{
  repeat(10)
  {
    repeat(4)
    {
      OnFwd(OUT_A+OUT_C);
      Wait(MOVE_TIME);
      OnRev(OUT_C);
      Wait(TURN_TIME);
    }
  }
  Off(OUT_A+OUT_C);
}
```

There is now one repeat statement inside the other. We call this a “nested” repeat statement. You can nest repeat statements as much as you like. Take a careful look at the brackets and the indentation used in the program. The task starts at the first bracket and ends at the last. The first repeat statement starts at the second bracket and ends at the fifth. The second, nested repeat statement starts at the third bracket and ends at the fourth. As you see the brackets always come in pairs, and the piece between the brackets we indent.

Adding comment

To make your program even more readable, it is good to add some comment to it. Whenever you put `//` on a line, the rest of that line is ignored and can be used for comments. A long comment can be put between `/*` and `*/`. Comments are colored green in the RCX Command Center. The full program could look as follows:

```
/*  10 SQUARES

    by Mark Overmars

This program make the robot run 10 squares
*/

#define MOVE_TIME   100     // Time for a straight move
#define TURN_TIME    85     // Time for turning 90 degrees

task main()
{
  repeat(10)                // Make 10 squares
  {
    repeat(4)
    {
      OnFwd(OUT_A+OUT_C);
      Wait(MOVE_TIME);
      OnRev(OUT_C);
      Wait(TURN_TIME);
    }
  }
  Off(OUT_A+OUT_C);         // Now turn the motors off
}
```

Summary

In this chapter you learned the use of the **`repeat`** statement and the use of comment. Also you saw the function of nested brackets and the use of indentation. With all you know so far you can make the robot move along all sorts of paths. It is a good exercise to try and write some variations of the programs in this chapter before continuing with the next chapter.

III. Using variables

Variables form a very important aspect of every programming language. Variables are memory locations in which we can store a value. We can use that value at different places and we can change it. Let me describe the use of variables using an example.

Moving in a spiral

Assume we want to adapt the above program in such a way that the robot drives in a spiral. This can be achieved by making the time we sleep larger for each next straight movement. That is, we want to increase the value of `MOVE_TIME` each time. But how can we do this? `MOVE_TIME` is a constant and constants cannot be changed. We need a variable instead. Variables can easily be defined in NQC. You can have 32 of these, and you can give each of them a separate name. Here is the spiral program.

```
#define TURN_TIME   85

int move_time;              // define a variable

task main()
{
  move_time = 20;           // set the initial value
  repeat(50)
  {
    OnFwd(OUT_A+OUT_C);
    Wait(move_time);       // use the variable for sleeping
    OnRev(OUT_C);
    Wait(TURN_TIME);
    move_time += 5;         // increase the variable
  }
  Off(OUT_A+OUT_C);
}
```

The interesting lines are indicated with the comments. First we define a variable by typing the keyword **`int`** followed by a name we choose. (Normally we use lower-case letters for variable names and uppercase letters for constants, but this is not necessary.) The name must start with a letter but can contain digits and the underscore sign. No other symbols are allowed. (The same applied to constants, task names, etc.) The strange word **`int`** stands for integer. Only integer numbers can be stored in it. In the second interesting line we assign the value 20 to the variable. From this moment on, whenever you use the variable, it stands for 20. Now follows the repeat loop in which we use the variable to indicate the time to sleep and, at the end of the loop we increase the value of the variable with 5. So the first time the robot sleeps 20 ticks, the second time 25, the third time 30, etc.

Besides adding values to a variable we can also multiply a variable with a number using `*=`, subtract using `-=` and divide using `/=`. (Note that for division the result is rounded to the nearest integer.) You can also add one variable to the other, and write down more complicated expressions. Here are some examples:

```
int aaa;
int bbb, ccc;

task main()
{
  aaa = 10;
  bbb = 20 * 5;
  ccc = bbb;
  ccc /= aaa;
  ccc -= 5;
  aaa = 10 * (ccc + 3);  // aaa is now equal to 80
}
```

Note on the first two lines that we can define multiple variables in one line. We could also have combined all three of them in one line.

Random numbers

In all the above programs we defined exactly what the robot was supposed to do. But things get a lot more interesting when the robot is going to do things that we don't know. We want some randomness in the motions. In NQC you can create random numbers. The following program uses this to let the robot drive around in a random way. It constantly drives forwards for a random amount of time and then makes a random turn.

```
int move_time, turn_time;

task main()
{
  while(true)
  {
    move_time = Random(60);
    turn_time = Random(40);
    OnFwd(OUT_A+OUT_C);
    Wait(move_time);
    OnRev(OUT_A);
    Wait(turn_time);
  }
}
```

The program defines two variables, and then assigns random numbers to them. `Random(60)` means a random number between 0 and 60 (it can also be 0 or 60). Each time the numbers will be different. (Note that we could avoid the use of the variables by writing e.g. `Wait(Random(60))`.)

You also see a new type of loop here. Rather that using the repeat statement we wrote **`while(true)`**. The while statement repeats the statements below it as long as the condition between the parentheses is true. The special word **`true`** is always true, so the statements between the brackets are repeated forever, just as we want. You will learn more about the while statement in Chapter IV.

Summary

In this chapter you learned about the use of variables. Variables are very useful but, due to restrictions of the robots, they are a bit limited. You can define only 32 of them and they can store only integers. But for many robot tasks this is good enough.

You also learned how to create random numbers, such that you can give the robot unpredictable behavior. Finally we saw the use of the while statement to make an infinite loop that goes on forever.

IV. Control structures

In the previous chapters we saw the repeat and while statements. These statements control the way the other statements in the program are executed. They are called “control structures”. In this chapter we will see some other control structures.

The if statement

Sometimes you want that a particular part of your program is only executed in certain situations. In this case the if statement is used. Let me give an example. We will again change the program we have been working with so far, but with a new twist. We want the robot to drive along a straight line and then either make a left or a right turn. To do this we need random numbers again. We pick a random number between 0 and 1, that is, it is either 0 or 1. If the number is 0 we make a right turn; otherwise we make a left turn. Here is the program:

```
#define MOVE_TIME   100
#define TURN_TIME    85

task main()
{
  while(true)
  {
    OnFwd(OUT_A+OUT_C);
    Wait(MOVE_TIME);
    if (Random(1) == 0)
    {
      OnRev(OUT_C);
    }
    else
    {
      OnRev(OUT_A);
    }
    Wait(TURN_TIME);
  }
}
```

The if statement looks a bit like the while statement. If the condition between the parentheses is true the part between the brackets is executed. Otherwise, the part between the brackets after the word **`else`** is executed. Let us look a bit better at the condition we use. It reads `Random(1) == 0`. This means that `Random(1)` must be equal to 0 to make the condition true. You might wonder why we use == rather than =. The reason is to distinguish it from the statement that put a value in a variable. You can compare values in different ways. Here are the most important ones:

==	equal to
<	smaller than
<=	smaller than or equal to
>	larger than
>=	larger than or equal to
!=	not equal to

You can combine conditions use `&&`, which means “and”, or `||`, which means “or”. Here are some examples of conditions:

`true`	always true
`false`	never true
`ttt != 3`	true when ttt is not equal to 3
`(ttt >= 5) && (ttt <= 10)`	true when ttt lies between 5 and 10
`(aaa == 10) || (bbb == 10)`	true if either aaa or bbb (or both) are equal to 10

Note that the if statement has two parts. The part immediately after the condition, that is executed when the condition is true, and the part after the else, that is executed when the condition is false. The keyword else and the part after it are optional. So you can leave them away if there is nothing to do when the condition is false.

The do statement

There is another control structure, the do statement. It has the following form:

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

The statements between the brackets after the do part are executed as long as the condition is true. The condition has the same form as in the if statement described above. Here is an example of a program. The robots runs around randomly for 20 seconds and then stops.

```
int move_time, turn_time, total_time;

task main()
{
  total_time = 0;
  do
  {
    move_time = Random(100);
    turn_time = Random(100);
    OnFwd(OUT_A+OUT_C);
    Wait(move_time);
    OnRev(OUT_C);
    Wait(turn_time);
    total_time += move_time; total_time += turn_time;
  }
  while (total_time < 2000);
  Off(OUT_A+OUT_C);
}
```

Note in this example that we placed two statements on one line. This is allowed. You can place as many statements on a line as you like (as long as there are semicolons in between). But for readability of the program this is often not a good idea.

Note also that the do statement behaves almost the same as the while statement. But in the while statement the condition is tested before executing the statements, while in the do statement the condition is tested at the end. For the while statement, the statements might never be executed, but for the do statement they are executed at least once.

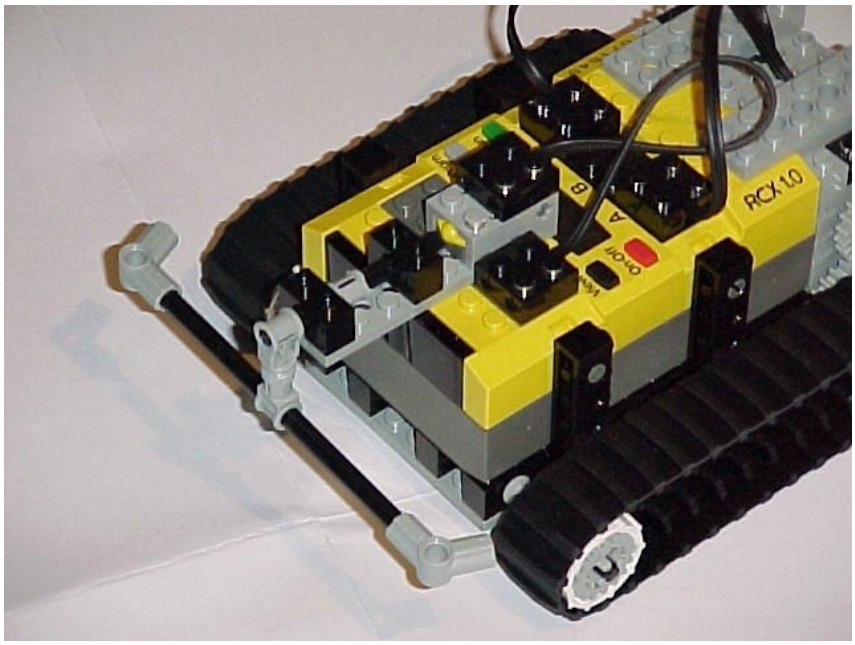
Summary

In this chapter we have seen two new control structures: the if statement and the do statement. Together with the repeat statement and the while statement they are the statements that control the way in which the program is executed. It is very important that you understand what they do. So better try some more examples yourself before continuing.

We also saw that we can place multiple statements on a line.

V. Sensors

One of the nice aspects of the Lego robots is that you can connect sensors to them and that you can make the robot react to the sensors. Before I can show how to do this we must change the robot a bit by adding a sensor. To this end, build the sensor construction shown in figure 4 on page 28 of the constructopedia. You might want to make it slightly wider, such that your robot looks as follows:



Connect the sensor to input 1 on the RCX.

Waiting for a sensor

Let us start with a very simple program in which the robot drives forwards until it hits something. Here it is:

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  OnFwd(OUT_A+OUT_C);
  until (SENSOR_1 == 1);
  Off(OUT_A+OUT_C);
}
```

There are two important lines here. The first line of the program tells the robot what type of sensor we use. `SENSOR_1` is the number of the input to which we connected the sensor. The other two sensor inputs are called `SENSOR_2` and `SENSOR_3`. `SENSOR_TOUCH` indicates that this is a touch sensor. For the light sensor we would use `SENSOR_LIGHT`. After we specified the type of the sensor, the program switches on both motors and the robot starts moving forwards. The next statement is a very useful construction. It waits until the condition between the brackets is true. This condition says that the value of the sensor `SENSOR_1` must be 1, which means that the sensor is pressed. As long as the sensor is not pressed, the value is 0. So this statement waits until the sensor is pressed. Then we switch off the motors and the task is finished.

Acting on a touch sensor

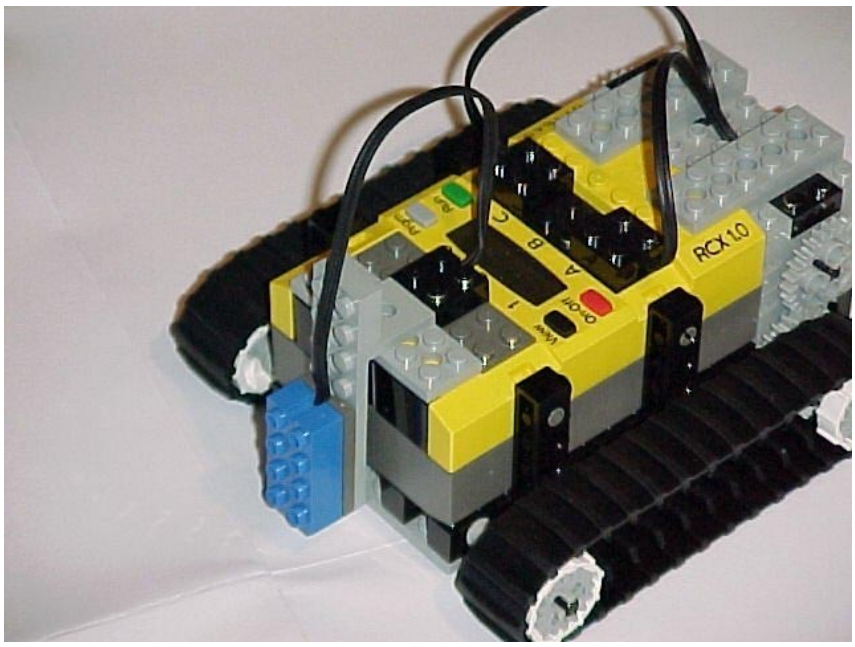
Let us now try to make the robot avoid obstacles. Whenever the robot hits an object, we let it move back a bit, make a turn, and then continue. Here is the program:

```
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);
    }
  }
}
```

As in the previous example, we first indicate the type of the sensor. Next the robot starts moving forwards. In the infinite while loop we constantly test whether the sensor is touched and, if so, move back for 1/3 of a second, turn right for 1/3 of a second, and then continue forwards again.

Light sensors

Besides touch sensors, you also get a light sensor with your MindStorms system. The light sensor measures the amount of light in a particular direction. The light sensor also emits light. In this way it is possible to point the light sensor in a particular direction and make a distinction between the intensity of the object in that direction. This is in particular useful when trying to make a robot follow a line on the floor. This is what we are going to do in the next example. We first need to attach the light sensor to the robot such that it is in the middle of the robot, at the front, and points downwards. Connect it to input 2. For example, make a construction as follows:



We also need the race track that comes with the RIS kit (This big piece of paper with the black track on it.) The idea now is that the robot makes sure that the light sensor stays above the track. Whenever the intensity of the light goes up, the light sensor is off the track and we need to adapt the direction. Here is a very simple program for this that only works if we travel around the track in clockwise direction.

```
#define THRESHOLD 40

task main()
{
  SetSensor(SENSOR_2,SENSOR_LIGHT);
  OnFwd(OUT_A+OUT_C);
  while (true)
  {
    if (SENSOR_2 > THRESHOLD)
    {
      OnRev(OUT_C);
      until (SENSOR_2 <= THRESHOLD);
      OnFwd(OUT_A+OUT_C);
    }
  }
}
```

The program first indicates that sensor 2 is a light sensor. Next it sets the robot to move forwards and goes into an infinite loop. Whenever the light value is bigger than 40 (we use a constant here such that this can be adapted easily, cause it depends a lot on the surrounding light) we reverse one motor and wait till we are on the track again.

As you will see when you execute the program, the motion is not very smooth. Try adding a `Wait(10)` command before the **until** command to make the robot move better. Note that the program does not work for moving counter-clockwise. To enable motion along arbitrary path a much more complicated program is required.

Summary

In this chapter you have seen how to work with touch sensors and light sensors. We also saw the **until** command that is useful when using sensors.

I recommend you to write a number of programs yourself at his stage. You have all the ingredients to give your robots pretty complicated behavior now. For example, try to put two touch sensors on your robot, one on the left front and the other on the right front, and make the robot move away from the obstacles it hits. Also, try to make a robot that stays within an area indicated by a thick black border line on the floor.

VI. Tasks and subroutines

Up to now all our programs consisted of just one task. But NQC programs can have multiple tasks. It is also possible to put pieces of code in so-called subroutines that you can use at different places in your program. Using tasks and subroutines makes your programs easier to understand and more compact. In this chapter we will look at the various possibilities.

Tasks

An NQC program consists of at most 10 tasks. Each task has a name. One task must have the name `main`, and this task will be executed. The other tasks will only be executed when a running tasks tells them to be executed using a start command. From this moment on both tasks are running simultaneously (so the first task continues running). A running task can also stop another running task by using the stop command. Later this task can be restarted again, but it will start from the beginning; not from the place where it was stopped.

Let me demonstrate the use of tasks. Put your touch sensor again on your robot. We want to make a program in which the robot drives around in squares, like before. But when it hits an obstacle it should react to it. It is difficult to do this in one task, because the robot must do two things at the same moment: drive around (that is, switching on and off motors at the right moments) and watch for sensors. So it is better to use two tasks for this, one task that drives the squares; the other that reacts to the sensors. Here is the program.

```
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;
    }
  }
}
```

The main task just sets the sensor type and then starts both other tasks. After this task main is finished. Task `move_square` moves the robot forever in squares. Task `check_sensors` checks whether the touch sensor is pushed. If so it takes the following actions: First of all it stops task `move_square`. This is very important. `check_sensors` now takes control over the motions of the robot. Next it moves the robot back a bit and makes it turn. Then it can start `move_square` again to let the robot again drive in squares.

It is very important to remember that tasks that you start are running at the same moment. This can lead to unexpected results. Chapter X explains these problems in detail and gives solutions for them.

Subroutines

Sometimes you need the same piece of code at multiple places in your program. In this case you can put the piece of code in a subroutine and give it a name. Now you can execute this piece of code by simply calling its name from within task. NQC (or actually the RCX) allows for at most 8 subroutines. Let us look at an example.

```
sub turn_around()
{
  OnRev(OUT_C); Wait(340);
  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);
}
```

In this program we have defined a subroutine that makes the robot rotate around its center. The main task calls the subroutine three times. Note that we call the subroutine by writing down its name with parentheses behind it. So it looks the same as many of the commands we have seen. Only there are no parameters, so there is nothing between the parentheses.

Some warnings are in place here. Subroutines are a bit weird. For example, subroutines cannot be called from other subroutines. Subroutines can be called from different tasks but this is not encouraged. It very easily leads to problems because the same subroutine might actually be run twice at the same moment by different tasks. This tends to give unwanted effects. Also, when calling a subroutine from different tasks, due to a limitation in the RCX firmware, you cannot use complicated expressions anymore. So, unless you know precisely what you are doing, *don't call a subroutine from different tasks!*

Inline functions

As indicated above, subroutines cause certain problems. The nice part is that they are stored only once in the RCX. This saves memory and, because the RCX does not have so much free memory, this is useful. But when subroutines are short, better use inline functions instead. These are not stored separately but copied at each place they are used. This costs more memory but problems like the ones with using complicated expressions, are no longer present. Also there is no limit on the number of inline functions.

Defining and calling inline functions goes exactly the same way as with subroutines. Only use the keyword **void** rather than **sub**. (The word **void** is used because this same word appears in other languages like C.) So the above example, using inline functions, looks as follows:

```
void turn_around()
{
  OnRev(OUT_C); Wait(340);
  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 functions have another advantage over subroutines. They can have arguments. Arguments can be used to pass a value for certain variables to an inline function. For example, assume, in the above example, we can make the time to turn an argument of the function, as in the following examples:

```
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);
}
```

Note that in the parenthesis behind the name of the inline function we specify the argument(s) of the function. In this case we indicate that the argument is an integer (there are some other choices) and that its name is turntime. When there are more arguments, you must separate them with commas.

Defining macro's

There is yet another way to give small pieces of code a name. You can define macro's in NQC (not to be confused with the macro's in RCX Command Center). We have seen before that we can define constants, using #define, by giving them a name. But actually we can define any piece of code. Here is the same program again but now using a macro for turning around.

```
#define turn_around OnRev(OUT_C);Wait(340);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);
}
```

After the #define statement the word turn_around stands for the text behind it. Now wherever you type turn_around, this is replaced by this text. Note that the text should be on one line. (Actually there are ways of putting a #define statement on multiple lines, but this is not recommended.)

Define statements are actually a lot more powerful. They can also have arguments. For example, we can put the time to turn as an argument in the statement. Here is an example in which we define four macro's; one to move forwards, one to move backwards, one to turn left and one to turn right. Each has two arguments: the speed and the time.

```
#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(3,200);
  turn_left(7,85);
  forwards(7,100);
  backwards(7,200);
  forwards(7,100);
  turn_right(7,85);
  forwards(3,200);
  Off(OUT_A+OUT_C);
}
```

It is very useful to define such macro's. It makes your code more compact and readable. Also, you can more easily change your code when you e.g. change the connections to the motors.

Summary

In this chapter you saw the use of tasks, subroutines, inline functions, and macro's. They have different uses. Tasks normally run at the same moment and take care of different things that have to be done at the same moment. Subroutines are useful when larger pieces of code must be used at different places in the same task. Inline functions are useful when pieces of code must be used a many different places in different tasks, but they use more memory. Finally macro's are very useful for small pieces of code that must be used a different places. They can also have parameters, making them even more useful.

Now that you have worked through the chapters up to here, you have all the knowledge you need to make your robot do complicated things. The other chapters in this tutorial teach you about other things that are only important in certain applications.

VII. Making music

The RCX has a built-in speaker that can make sounds and even play simple pieces of music. This is in particular useful when you want to make the RCX tell you that something is happening. But it can also be funny to have the robot make music while it runs around.

Built-in sounds

There are six built-in sounds in the RCX, numbered from 0 to 5. They sound as follows:

0	Key click
1	Beep beep
2	Decreasing frequency sweep
3	Increasing frequency sweep
4	'Buhhh' Error sound
5	Fast increasing sweep

You can play them using the commands `PlaySound()`. Here is a small program that plays all of them.

```
task main()
{
  PlaySound(0); Wait(100);
  PlaySound(1); Wait(100);
  PlaySound(2); Wait(100);
  PlaySound(3); Wait(100);
  PlaySound(4); Wait(100);
  PlaySound(5); Wait(100);
}
```

You might wonder why there are these wait commands. The reason is that the command that plays the sound does not wait for it to finish. It immediately executes the next command. The RCX has a little buffer in which it can store some sounds but after a while this buffer get full and sounds get lost. This is not so serious for sounds but it is very important for music, as we will see below.

Note that the argument to `PlaySound()` must be a constant. You cannot put a variable here!

Playing music

For more interesting music, NQC has the command `PlayTone()`. It has two arguments. The first is the frequency, and the second the duration (in ticks of 1/100h of a second, like in the wait command). Here is a table of useful frequencies:

Sound	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
G#	52	104	208	415	831	1661	3322	
G	49	98	196	392	784	1568	3136	
F#	46	92	185	370	740	1480	2960	
F	44	87	175	349	698	1397	2794	
E	41	82	165	330	659	1319	2637	
D#	39	78	156	311	622	1245	2489	
D	37	73	147	294	587	1175	2349	
C#	35	69	139	277	554	1109	2217	
C	33	65	131	262	523	1047	2093	4186
B	31	62	123	247	494	988	1976	3951
A#	29	58	117	233	466	932	1865	3729
A	28	55	110	220	440	880	1760	3520

As we noted above for sounds, also here the RCX does not wait for the note to finish. So if you use a lot in a row better add (slightly longer) wait commands in between. Here is an example:

```
task main()
{
  PlayTone(262,40);  Wait(50);
  PlayTone(294,40);  Wait(50);
  PlayTone(330,40);  Wait(50);
  PlayTone(294,40);  Wait(50);
  PlayTone(262,160); Wait(200);
}
```

You can create pieces of music very easily using the RCX Piano that is part of the RCX Command Center.

If you want to have the RCX play music while driving around, better use a separate task for it. Here you have an example of a rather stupid program where the RCX drives back and forth, constantly making music.

```
task music()
{
  while (true)
  {
    PlayTone(262,40);  Wait(50);
    PlayTone(294,40);  Wait(50);
    PlayTone(330,40);  Wait(50);
    PlayTone(294,40);  Wait(50);
  }
}

task main()
{
  start music;
  while(true)
  {
    OnFwd(OUT_A+OUT_C); Wait(300);
    OnRev(OUT_A+OUT_C); Wait(300);
  }
}
```

Summary

In this chapter you learned how to let the RCX make sounds and music. Also you saw how to use a separate task for music.

VIII. More about motors

There are a number of additional motor commands that you can use to control the motors more precisely. In this chapter we discuss them.

Stopping gently

When you use the `Off()` command, the motor stops immediately, using the brake. In NQC it is also possible to stop the motors in a more gentle way, not using the brake. For this you use the `Float()` command. Sometimes this is better for your robot task. Here is an example. First the robot stops using the brakes; next without using the brakes. Note the difference. (Actually the difference is very small for this particular robot. But it makes a big difference for some other robots.)

```
task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(200);
  Off(OUT_A+OUT_C);
  Wait(100);
  OnFwd(OUT_A+OUT_C);
  Wait(200);
  Float(OUT_A+OUT_C);
}
```

Advanced commands

The command `OnFwd()` actually does two things: it switches the motor on and it sets the direction to forwards. The command `OnRev()` also does two things: it switches the motor on and sets the direction to reverse. NQC also has commands to do these two things separately. If you only want to change one of the two things, it is more efficient to use these separate commands; it uses less memory in the RCX, it is faster, and it can result in smoother motions. The two separate commands are `SetDirection()` that sets the direction (`OUT_FWD`, `OUT_REV` or `OUT_TOGGLE` which flips the current direction) and `SetOutput()` that sets the mode (`OUT_ON`, `OUT_OFF` or `OUT_FLOAT`). Here is a simple program that makes the robot drive forwards, backwards and forwards again.

```
task main()
{
  SetPower(OUT_A+OUT_C,7);
  SetDirection(OUT_A+OUT_C,OUT_FWD);
  SetOutput(OUT_A+OUT_C,OUT_ON);
  Wait(200);
  SetDirection(OUT_A+OUT_C,OUT_REV);
  Wait(200);
  SetDirection(OUT_A+OUT_C,OUT_TOGGLE);
  Wait(200);
  SetOutput(OUT_A+OUT_C,OUT_FLOAT);
}
```

Note that, at the start of every program, all motors are set in forward direction and the speed is set to 7. So in the above example, the first two commands are not necessary.

There are a number of other motor commands, which are shortcuts for combinations of the commands above. Here is a complete list:

`On('motors')`	Switches the motors on
`Off('motors')`	Switches the motors off
`Float('motors')`	Switches the motors of smoothly
`Fwd('motors')`	Switches the motors forward (but does not make them drive)
`Rev('motors')`	Switches the motors backwards (but does not make them drive)
`Toggle('motors')`	Toggles the direction of the motors (forward to backwards and back)
`OnFwd('motors')`	Switches the motors forward and turns them on
`OnRev('motors')`	Switches the motors backwards and turns them on

`OnFor('motors','ticks')`	Switches the motors on for ticks time
`SetOutput('motors','mode')`	Sets the output mode (`OUT_ON`, `OUT_OFF` or `OUT_FLOAT`)
`SetDirection('motors','dir')`	Sets the output direction (`OUT_FWD`, `OUT_REV` or `OUT_TOGGLE`)
`SetPower('motors','power')`	Sets the output power (0-9)

Varying motor speed

As you probably noticed, changing the speed of the motors does not have much effect. The reason is that you are mainly changing the torque, not the speed. You will only see an effect when the motor has a heavy load. And even then, the difference between 2 and 7 is very small. If you want to have better effects the trick is to turn the motors on and off in rapid succession. Here is a simple program that does this. It has one task, called run_motor that drives the motors. It constantly checks the variable speed to see what the current speed is. Positive is forwards, negative backwards. It sets the motors in the right direction and then waits for some time, depending on speed, before switching the motors off again. The main task simply sets speeds and waits.

```
int speed, __speed;

task run_motor()
{
  while (true)
  {
    __speed = speed;
    if (__speed > 0) {OnFwd(OUT_A+OUT_C);}
    if (__speed < 0) {OnRev(OUT_A+OUT_C); __speed = -__speed;}
    Wait(__speed);
    Off(OUT_A+OUT_C);
  }
}

task main()
{
  speed = 0;
  start run_motor;
  speed = 1;   Wait(200);
  speed = -10; Wait(200);
  speed = 5;   Wait(200);
  speed = -2;  Wait(200);
  stop run_motor;
  Off(OUT_A+OUT_C);
}
```

This program can be made much more powerful, allowing for rotations, and also possibly incorporating a waiting time after the `Off()` command. Experiment yourself.

Summary

In this chapter you learned about the extra motor commands that are available: `Float()` that stops the motor gently, `SetDirection()` that sets the direction (`OUT_FWD`, `OUT_REV` or `OUT_TOGGLE` which flips the current direction) and `SetOutput()` that sets the mode (`OUT_ON`, `OUT_OFF` or `OUT_FLOAT`). You saw the complete list of motor commands available. You also learned a trick to control the motor speed in a better way.

IX. More about sensors

In Chapter V we discussed the basic aspects of using sensors. But there is a lot more you can do with sensors. In this chapter we will discuss the difference between sensor mode and sensor type, we will see how to use the rotation sensor (a type of sensor that is not provided with the RIS but can be bought separately and is very useful), and we will see some tricks to use more than three sensors and to make a proximity sensor.

Sensor mode and type

The `SetSensor()` command that we saw before does actually two things: it sets the type of the sensor, and it sets the mode in which the sensor operates. By setting the mode and type of the a sensor separately, you can control the behavior of the sensor more precisely, which is useful for particular applications.

The type of the sensor is set with the command `SetSensorType()`. There are four different types: `SENSOR_TYPE_TOUCH`, which is the touch sensor, `SENSOR_TYPE_LIGHT`, which is the light sensor, `SENSOR_TYPE_TEMPERATURE`, which is the temperature sensor (this type of sensor is not part of the RIS but can be bought separately), and `SENSOR_TYPE_ROTATION`, which is the rotation sensor (also not part of the RIS but available separately). Setting the type sensor is in particular important to indicate whether the sensor needs power (like e.g. for the light of the light sensor). I know of no uses for setting a sensor to a different type than it actually is.

The mode of the sensor is set with the command `SetSensorMode()`. There are eight different modes. The most important one is `SENSOR_MODE_RAW`. In this mode, the value you get when checking the sensor is a number between 0 and 1023. It is the raw value produced by the sensor. What it means depends on the actual sensor. For example, for a touch sensor, when the sensor is not pushed the value is close to 1023. When it is fully pushed, it is close to 50. When it is pushed partially the value ranges between 50 and 1000. So if you set a touch sensor to raw mode you can actually find out whether it is touched partially. When the sensor is a light sensor, the value ranges from about 300 (very light) to 800 (very dark). This gives a much more precise value than using the `SetSensor()` command.

The second sensor mode is `SENSOR_MODE_BOOL`. In this mode the value is 0 or 1. When the raw value is above about 550 the value is 0, otherwise it is 1. `SENSOR_MODE_BOOL` is the default mode for a touch sensor. The modes `SENSOR_MODE_CELSIUS` and `SENSOR_MODE_FAHRENHEIT` are useful with temperature sensors only and give the temperature in the indicated way. `SENSOR_MODE_PERCENT` turns the raw value into a value between 0 and 100. Every raw value of 400 or lower is mapped to 100 percent. If the raw value gets higher, the percentage slowly goes down to 0. `SENSOR_MODE_PERCENT` is the default mode for a light sensor. `SENSOR_MODE_ROTATION` seems to be useful only for the rotation sensor (see below).

There are two other interesting modes: `SENSOR_MODE_EDGE` and `SENSOR_MODE_PULSE`. They count transitions, that is changes from a low to a high raw value or opposite. For example, when you touch a touch sensor this causes a transition from high to low raw value. When you release it you get a transition the other direction. When you set the sensor mode to `SENSOR_MODE_PULSE`, only transitions from low to high are counted. So each touch and release of the touch sensor counts for one. When you set the sensor mode to `SENSOR_MODE_EDGE`, both transitions are counted. So each touch and release of the touch sensor counts for two. So you can use this to count how often a touch sensor is pushed. Or you can use it in combination with a light sensor to count how often a (strong) lamp is switched on and off. Of course, when you are counting things, you should be able to set the counter back to 0. For this you use the command `ClearSensor()`. It clears the counter for the indicated sensor(s).

Let us look at an example. The following program uses a touch sensor to steer the robot. Connect the touch sensor with a long wire to input one. If touch the sensor quickly twice the robot moves forwards. It you touch it once it stops moving.

```
task main()
{
  SetSensorType(SENSOR_1,SENSOR_TYPE_TOUCH);
  SetSensorMode(SENSOR_1,SENSOR_MODE_PULSE);
  while(true)
  {
    ClearSensor(SENSOR_1);
    until (SENSOR_1 >0);
    Wait(100);
    if (SENSOR_1 == 1) {Off(OUT_A+OUT_C);}
    if (SENSOR_1 == 2) {OnFwd(OUT_A+OUT_C);}
  }
}
```

Note that we first set the type of the sensor and then the mode. It seems that this is essential because changing the type also effects the mode.

The rotation sensor

The rotation sensor is a very useful type of sensor that is unfortunately not part of the standard RIS. It can though be bought separately from Lego. The rotation sensor contains a hole through which you can put an axle. The rotation sensor measures the amount the axle is rotated. One full rotation of the axle is 16 steps (or –16 if you rotate it the other way). Rotation sensors are very useful to make the robot make precisely controlled movements. You can make an axle move the exact amount you want. If you need finer control than 16 step, you can always use gears to connect it to an axle that moves faster, and use that one for counting steps.

One standard application is to have two rotation sensors connected to the two wheels of the robot that you control with the two motors. For a straight movement you want both wheels to turn equally fast. Unfortunately, the motors normally don't run at exactly the same speed. Using the rotation sensors you can see that one wheel turns faster. You can then temporarily stop that motor (best using `Float()`) until both sensors give the same value again. The following program does this. It simply lets the robot drive in a straight line. To use it, change your robot by connecting the two rotation sensors to the two wheels. Connect the sensors to input 1 and 3.

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_ROTATION); ClearSensor(SENSOR_1);
  SetSensor(SENSOR_3,SENSOR_ROTATION); ClearSensor(SENSOR_3);
  while (true)
  {
    if (SENSOR_1 < SENSOR_3)
      {OnFwd(OUT_A); Float(OUT_C);}
    else if (SENSOR_1 > SENSOR_3)
      {OnFwd(OUT_C); Float(OUT_A);}
    else
      {OnFwd(OUT_A+OUT_C);}
  }
}
```

The program first indicates that both sensors are rotation sensors, and resets the values to zero. Next it start an infinite loop. In the loop we check whether the two sensor readings are equal. If they are the robot simply moves forwards. If one is larger, the correct motor is stopped until both readings are again equal.

Clearly this is only a very simple program. You can extend this to make the robot drive exact distances, or to let it make very precise turns.

Putting multiple sensors on one input

The RCX has only three inputs so you can connect only three sensors to it. When you want to make more complicated robots (and you bought some extra sensors) this might not be enough for you. Fortunately, with some tricks, you can connect two (or even more) sensors to one input.

The easiest is to connect two touch sensors to one input. If one of them (or both) is touched, the value is 1, otherwise it is 0. You cannot distinguish the two but sometimes this is not necessary. For example, when you put one touch sensor at the front and one at the back of the robot, you know which one is touched based on the

direction the robot is driving in. But you can also set the mode of the input to raw (see above). Now you can get a lot more information. If you are lucky, the value when the sensor is pressed is not the same for both sensors. If this is the case you can actually distinguish between the two sensors. And when both are pressed you get a much lower value (around 30) so you can also detect this.

You can also connect a touch sensor and a light sensor to one input. Set the type to light (otherwise the light sensor won't work). Set the mode to raw. In this case, when the touch sensor is pushed you get a raw value below 100. If it is not pushed you get the value of the light sensor which is never below 100. The following program uses this idea. The robot must be equipped with a light sensor pointing down, and a bumper at the front connected to a touch sensor. Connect both of them to input 1. The robot will drive around randomly within a light area. When the light sensor sees a dark line (raw value > 750) it goes back a bit. When the touch sensor touches something (raw value below 100) it does the same. Here is the program:

```
int ttt,tt2;

task moverandom()
{
  while (true)
  {
    ttt = Random(50) + 40;
    tt2 = Random(1);
    if (tt2 > 0)
      { OnRev(OUT_A); OnFwd(OUT_C); Wait(ttt); }
    else
      { OnRev(OUT_C); OnFwd(OUT_A);Wait(ttt); }
    ttt = Random(150) + 50;
    OnFwd(OUT_A+OUT_C);Wait(ttt);
  }
}

task main()
{
  start moverandom;
  SetSensorType(SENSOR_1,SENSOR_TYPE_LIGHT);
  SetSensorMode(SENSOR_1,SENSOR_MODE_RAW);
  while (true)
  {
    if ((SENSOR_1 < 100) || (SENSOR_1 > 750))
    {
      stop moverandom;
      OnRev(OUT_A+OUT_C);Wait(30);
      start moverandom;
    }
  }
}
```

I hope the program is clear. There are two tasks. Task `moverandom` makes the robot move around in a random way. The main task first starts `moverandom`, sets the sensor and then waits for something to happen. If the sensor reading gets too low (touching) or too high (out of the white area) it stops the random moves, backs up a little, and start the random moves again.

It is also possible to connect two light sensors to the same input. The raw value is in some way related to the combined amount of light received by the two sensors. But this is rather unclear and seems hard to use. Connecting other sensors with rotation or temperature sensors seems not to be useful.

Making a proximity sensor

Using touch sensors, your robot can react when it hits something. But it would be a lot nicer when the robot could react just before it hits something. It should know that it is near to some obstacle. Unfortunately there are no sensors for this available. There is though a trick we can use for this. The robot has an infra-red port with which it can communicate with the computer, or with other robots. (We will see more about the communication between robots in Chapter XI.) It turns out that the light sensor that comes with the robot is very sensitive to infra-red light. We can build a proximity sensor based on this. The idea is that one tasks sends out infra-red

messages. Another task measures fluctuations in the light intensity that is reflected from objects. The higher the fluctuation, the closer we are to an object.

To use this idea, place the light sensor above the infra-red port on the robot, pointing forwards. In this way it only measures reflected infra-red light. Connect it to input 2. We use raw mode for the light sensor to see the fluctuations as good as possible. Here is a simple program that lets the robot run forwards until it gets near to an object and then makes a 90 degree turn to the right.

```
int lastlevel;           // To store the previous level

task send_signal()
{
  while(true)
    {SendMessage(0); Wait(10);}
}

task check_signal()
{
  while(true)
  {
    lastlevel = SENSOR_2;
    if(SENSOR_2 > lastlevel + 200)
      {OnRev(OUT_C); Wait(85); OnFwd(OUT_A+OUT_C);}
  }
}

task main()
{
  SetSensorType(SENSOR_2, SENSOR_TYPE_LIGHT);
  SetSensorMode(SENSOR_2, SENSOR_MODE_RAW);
  OnFwd(OUT_A+OUT_C);
  start send_signal;
  start check_signal;
}
```

The task `send_signal` send out 10 IR signals every seconds, using the command `SendMessage(0)`. The task `check_signal` repeatedly saves the value of the light sensor. Then it checks whether it (slightly later) has become at least 200 higher, indicating a large fluctuation. If so, it lets the robot make a 90-degree turn to the right. The value of 200 is rather arbitrary. If you make it smaller, the robot turns further away from obstacles. If you make it larger, it gets closer to them. But this also depends on the type of material and the amount of light available in the room. You should experiment or use some more clever mechanism for learning the correct value.

A disadvantage of the technique is that it only works in one direction. You probably still need touch sensors at the sides to avoid collisions there. But the technique is very useful for robots that must drive around in mazes. Another disadvantage is that you cannot communicate from the computer to the robot because it will interfere with the infra-red commands send out by the robot. (Also the remote control on your television might not work.)

Summary

In this chapter we have seen a number of additional issues about sensors. We saw how to separately set the type and mode of a sensor and how this could be used to get additions information. We learned how to use the rotation sensor. And we saw how multiple sensors can be connected to one input of the RCX. Finally, we saw a trick to use the infra red connection of the robot in combination with a light sensor, to create a proximity sensor. All these tricks are extremely useful when constructing more complicated robots. Sensors always play a crucial role there.

X. Parallel tasks

As has been indicated before, tasks in NQC are executed simultaneously, or in parallel as people usually say. This is extremely useful. In enables you to watch sensors in one task while another task moves the robot around, and yet another task plays some music. But parallel tasks can also cause problems. One task can interfere with another.

A wrong program

Consider the following program. Here one task drives the robot around in squares (like we did so often before) and the second task checks for the touch sensor. When the sensor is touched, it moves a bit backwards, and makes a 90-degree turn.

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

task check_sensors()
{
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      OnRev(OUT_A+OUT_C);
      Wait(50);
      OnFwd(OUT_A);
      Wait(85);
      OnFwd(OUT_C);
    }
  }
}
```

This probably looks like a perfectly valid program. But if you execute it you will most likely find some unexpected behavior. Try the following: Make the robot touch something while it is turning. It will start going back, but immediately moves forwards again, hitting the obstacle. The reason for this is that the tasks may interfere. The following is happening. The robot is turning right, that is, the first task is in its second sleep statement. Now the robot hits the sensor. It start going backwards, but at that very moment, the main task is ready with sleeping and moves the robot forwards again; into the obstacle. The second task is sleeping at this moment so it won't notice the collision. This is clearly not the behavior we would like to see. The problem is that, while the second task is sleeping we did not realize that the first task was still running, and that its actions interfere with the actions of the second task.

Stopping and restarting tasks

One way of solving this problem is to make sure that at any moment only one task is driving the robot. This was the approach we took in Chapter VI. Let me repeat the program here.

```
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;
    }
  }
}
```

The crux is that the `check_sensors` task only moves the robot after stopping the `move_square` task. So this task cannot interfere with the moving away from the obstacle. Once the backup procedure is finished, it starts `move_square` again.

Even though this is a good solution for the above problem, there is a problem. When we restart `move_square`, it starts again at the beginning. This is fine for our small task, but often this is not the required behavior. We would prefer to stop the task where it is and continue it later from that point. Unfortunately this cannot be done easily.

Using semaphores

A standard technique to solve this problem is to use a variable to indicate which task is in control of the motors. The other tasks are not allowed to drive the motors until the first task indicates, using the variable, that it is ready. Such a variable is often called a semaphore. Let `sem` be such a semaphore. We assume that a value of 0 indicates that no task is steering the motors. Now, whenever a task wants to do something with the motors it executes the following commands:

```
until (sem == 0);
sem = 1;
// Do something with the motors
sem = 0;
```

So we first wait till nobody needs the motors. Then we claim the control by setting `sem` to 1. Now we can control the motors. When we are done we set `sem` back to 0. Here you find the program above, implemented using a semaphore. When the touch sensor touches something, the semaphore is set and the backup procedure is performed. During this procedure the task `move_square` must wait. At the moment the back-up is ready, the semaphore is set to 0 and `move_square` can continue.

```
int sem;

task main()
{
  sem = 0;
  start move_square;
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      until (sem == 0); sem = 1;
      OnRev(OUT_A+OUT_C); Wait(50);
      OnFwd(OUT_A); Wait(85);
      sem = 0;
    }
  }
}

task move_square()
{
  while (true)
  {
    until (sem == 0); sem = 1;
    OnFwd(OUT_A+OUT_C);
    sem = 0;
    Wait(100);
    until (sem == 0); sem = 1;
    OnRev(OUT_C);
    sem = 0;
    Wait(85);
  }
}
```

You could argue that it is not necessary in `move_square` to set the semaphore to 1 and back to 0. Still this is useful. The reason is that the `OnFwd()` command is in fact two commands (see Chapter VIII). You don't want this command sequence to be interrupted by the other task.

Semaphores are very useful and, when you are writing complicated programs with parallel tasks, they are almost always required. (There is still a slight chance they might fail. Try to figure out why.)

Summary

In this chapter we studied some of the problems that can occur when you use different tasks. Always be very careful for side effects. Much unexpected behavior is due to this. We saw two different ways of solving such problems. The first solution stops and restarts tasks to make sure that only one critical task is running at every moment. The second approach uses semaphores to control the execution of tasks. This guarantees that at every moment only the critical part of one task is executed.

XI. Communication between robots

If you own more than one RCX this chapter is for you. The robots can communicate with each other through the infra-red port. Using this you can have multiple robots collaborate (or fight with each other). Also you can build one big robot using two RCX's, such that you can have six motors and six sensors (or even more using the tricks in Chapter IX).

Communication between robots works, globally speaking, as follows. A robot can use the command `SendMessage()` to send a value (0-255) over the infra-red port. All other robots receive this message and store it. The program in a robot can ask for the value of the last message received using `Message()`. Based on this value the program can make the robot perform certain actions.

Giving orders

Often, when you have two or more robots, one is the leader. We call him the *master*. The other robots are *slaves*. The master robot sends orders to the slaves and the slaves execute these. Sometimes the slaves might send information back to the master, for example the value of a sensor. So you need to write two programs, one for the master and one for the slave(s). From now on we assume that we have just one slave. Let us start with a very simple example. Here the slave can perform three different orders: move forwards, move backwards, and stop. Its program consists of a simple loop. In this loop it sets the value of the current message to 0 using the `ClearMessage()` command. Next it waits until the message becomes unequal to 0. Based on the value of the message it executes one of the three orders. Here is the program.

```
task main()          // SLAVE
{
  while (true)
  {
    ClearMessage();
    until (Message() != 0);
    if (Message() == 1) {OnFwd(OUT_A+OUT_C);}
    if (Message() == 2) {OnRev(OUT_A+OUT_C);}
    if (Message() == 3) {Off(OUT_A+OUT_C);}
  }
}
```

The master has an even simpler program. It simply send the messages corresponding to orders and then waits a bit. In the program below it orders the slave to move forwards, then, after two seconds, backwards, and then, again after two seconds, to stop.

```
task main()          // MASTER
{
  SendMessage(1); Wait(200);
  SendMessage(2); Wait(200);
  SendMessage(3);
}
```

After you have written these two program, you need to download them to the robots. Each program must go to one of the robots. Make sure you switch the other one off in the meantime (see also the cautions below). Now switch on both robots and start the programs: first the one in the slave and then the one in the master.

If you have multiple slaves, you have to download the slave program to each of them in turn (not simultaneously; see below). Now all slaves will perform exactly the same actions.

To let the robots communicate with each other we defined, what is called, a protocol: We decided that a 1 means to move forwards, a 2 to move backwards, and a 3 to stop. It is very important to carefully define such protocols, in particular when you are dealing with lots of communications. For example, when there are more slaves, you could define a protocol in which two numbers are send (with a small sleep in between): the first number is the number of the slave, and the second is the actual order. The slave than first check the number and only perform the action if it is his number. (This requires that each slave has its own number, which can be achieved by letting each slave have a slightly different program in which e.g. one constant is different.)

Electing a leader

As we saw above, when dealing with multiple robots, each robot must have its own program. It would be much easier if we could download just one program to all robots. But then the question is: who is the master? The answer is easy: let the robots decide themselves. Let them elect a leader which the others will follow. But how do we do this? The idea is rather simple. We let each robot wait a random amount of time and then send a message. The one that sends a message first is the leader. This scheme might fail if two robots wait exactly the same amount of time but this is rather unlikely. (You can build more complicated schemes that detect this and try a second election in such a case.) Here is the program that does it:

```
task main()
{
  ClearMessage();
  Wait(200);              // make sure all robots are on
  Wait(Random(400));      // wait between 0 and 4 seconds
  if (Message() > 0)      // somebody else was first
  {
    start slave;
  }
  else
  {
    SendMessage(1);       // I am the master now
    Wait(400);            // make sure everybody else knows
    start master;
  }
}

task master()
{
  SendMessage(1); Wait(200);
  SendMessage(2); Wait(200);
  SendMessage(3);
}

task slave()
{
  while (true)
  {
    ClearMessage();
    until (Message() != 0);
    if (Message() == 1) {OnFwd(OUT_A+OUT_C);}
    if (Message() == 2) {OnRev(OUT_A+OUT_C);}
    if (Message() == 3) {Off(OUT_A+OUT_C);}
  }
}
```

Download this program to all robots (one by one, not at the same moment; see below). Start the robots at about the same moment and see what happens. One of them should take command and the other(s) should follow the orders. In rare occasions, none of them becomes the leader. As indicated above, this requires more careful protocols to solve.

Cautions

You have to be a bit careful when dealing with multiple robots. There are two problems: If two robots (or a robot and the computer) send information at the same time this might be lost. The second problem is that, when the computer sends a program to multiple robots at the same time, this causes problems.

Let us start with the second problem. When you download a program to the robot, the robot tells the computer whether it correctly receives (parts of) the program. The computer reacts on that by sending new pieces or by resending parts. When two robots are on, both will start telling the computer whether they correctly receive the program. The computer does not understand this (it does not know that there are two robots!). As a result, things go wrong and the program gets corrupted. The robots won't do the right things. *Always make sure that, while you are downloading programs, only one robot is on!*

The other problem is that only one robot can send a message at any moment. If two messages are being send at roughly the same moment, they might get lost. Also, a robot cannot send and receive messages at the same moment. This is no problem when only one robot sends messages (there is only one master) but otherwise it might be a serious problem. For example, you can imagine writing a program in which a slave sends a message when it bumps into something, such that the master can take action. But if the master sends an order at the same moment, the message will get lost. To solve this, it is important to define your communication protocol such that, in case a communication fails, this is corrected. For example, when the master sends a command, it should get an answer from the slave. If is does not get an answer soon enough, it resends the command. This would result in a piece of code that looks like this:

```
do
{
  SendMessage(1);
  ClearMessage();
  Wait(10);
}
while (Message() != 255);
```

Here 255 is used for the acknowledgement.

Sometimes, when you are dealing with multiple robots, you might want that only a robot that is very close by receives the signal. This can be achieved by adding the command `SetTxPower(TX_POWER_LO)` to the program of the master. In this case the IR signal send is very low and only a robot close by and facing the master will "hear" it. This is in particular useful when building one bigger robot out of two RCX's. Use `SetTxPower(TX_POWER_HI)` to set the robot again in long range transmission mode.

Summary

In this chapter we studied some of the basic aspects of communication between robots. Communication uses the commands to send, clear, and check messages. We saw that is is important to define a protocol for how the communication works. Such protocols play a crucial role in any form of communication between computers. We also saw that there are a number of restrictions in the communication between robots which makes it even more important to define good protocols.

XII. More commands

NQC has a number of additional commands. In this chapter we will discuss three types: the use of timers, commands to control the display, and the use of the datalog feature of the RCX.

Timers

The RCX has four built-in timers. These timers tick in increments of 1/10 of a second. The timers are numbered from 0 to 3. You can reset the value of a timer with the command `ClearTimer()` and get the current value of the timer with `Timer()`. Here is an example of the use of a timer. The following program lets the robot drive sort of random for 20 seconds.

```
task main()
{
  ClearTimer(0);
  do
  {
    OnFwd(OUT_A+OUT_C);
    Wait(Random(100));
    OnRev(OUT_C);
    Wait(Random(100));
  }
  while (Timer(0)<200);
  Off(OUT_A+OUT_C);
}
```

You might want to compare this program with the one given in Chapter IV that did exactly the same task. The one with timers is definitely simpler.

Timers are very useful as a replacement for a `Wait()` command. You can sleep for a particular amount of time by resetting a timer and then waiting till it reaches a particular value. But you can also react on other events (e.g. from sensors) while waiting. The following simple program is an example of this. It lets the robot drive until either 10 seconds are past, or the touch sensor touches something.

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  ClearTimer(3);
  OnFwd(OUT_A+OUT_C);
  until ((SENSOR_1 == 1) || (Timer(3) >100));
  Off(OUT_A+OUT_C);
}
```

Don’t forget that timers work in ticks of 1/10 of a second, while e.g. the wait command uses ticks of 1/100 of a second.

The display

It is possible to control the display of the RCX in two different ways. First of all, you can indicate what to display: the system clock, one of the sensors, or one the motors. This is equivalent to using the black view button on the RCX. To set the display type, use the command `SelectDisplay()`. The following program shows all seven possibilities, one after the other.

```
task main()
{
  SelectDisplay(DISPLAY_SENSOR_1); Wait(100);  // Input 1
  SelectDisplay(DISPLAY_SENSOR_2); Wait(100);  // Input 2
  SelectDisplay(DISPLAY_SENSOR_3); Wait(100);  // Input 3
  SelectDisplay(DISPLAY_OUT_A);    Wait(100);  // Output A
  SelectDisplay(DISPLAY_OUT_B);    Wait(100);  // Output B
  SelectDisplay(DISPLAY_OUT_C);    Wait(100);  // Output C
  SelectDisplay(DISPLAY_WATCH);    Wait(100);  // System clock
}
```

Note that you should not use `SelectDisplay(SENSOR_1)`.

The second way you can control the display is by controlling the value of the system clock. You can use this to display e.g. diagnostic information. For this use the command `SetWatch()`. Here is a tiny program that uses this:

```
task main()
{
  SetWatch(1,1); Wait(100);
  SetWatch(2,4); Wait(100);
  SetWatch(3,9); Wait(100);
  SetWatch(4,16); Wait(100);
  SetWatch(5,25); Wait(100);
}
```

Note that the arguments to `SetWatch()` must be constants.

Datalogging

The RCX can store values of variables, sensor readings, and timers, in a piece of memory called the datalog. The values in the datalog cannot be used inside the RCX, but they can be read by your computer. This is useful to e.g. check what is going on in your robot. RCX Command Center has a special window in which you can view the current contents of the datalog.

Using the datalog consists of three steps: First, the NQC program must define the size of the datalog, using the command `CreateDatalog()`. This also clears the current contents of the datalog. Next, values can be written in the datalog using the command `AddToDatalog()`. The values will be written one after the other. (If you look at the display of the RCX you will see that one after the other, four parts of a disk appear. When the disk is complete, the datalog is full.) If the end of the datalog is reached, nothing happens. New values are no longer stored. The third step is to upload the datalog to the PC. For this, choose in RCX Command Center the command **Datalog** in the **Tools** menu. Next press the button labelled **Upload Datalog**, and all the values appear. You can watch them or save them to a file to do something else with them. People have used this feature to e.g. make a scanner with the RCX.

Here is a simple example of a robot with a light sensor. The robot drives for 10 seconds, and five times a second the value of the light sensor is written into the datalog.

```
task main()
{
  SetSensor(SENSOR_2,SENSOR_LIGHT);
  OnFwd(OUT_A+OUT_C);
  CreateDatalog(50);
  repeat (50)
  {
    AddToDatalog(SENSOR_2);
    Wait(20);
  }
  Off(OUT_A+OUT_C);
}
```

XIII. NQC quick reference

Below you find a list of all statement constructions, commands, constants, etc. in NQC. Most of these have been treated in the chapters above, so only short descriptions are given.

Statements

Statement	Description
while (*cond*) *body*	Execute body zero or more times while condition is true
do *body* while (*cond*)	Execute body one or more times while condition is true
until (*cond*) *body*	Execute body zero or more times until condition is true
break	Break out from while/do/until body
continue	Skip to next iteration of while/do/until body
repeat (*expression*) *body*	Repeat body a specified number of times
if (*cond*) *stmt1* **if** (*cond*) *stmt1* **else** *stmt2*	Execute stmt1 if condition is true. Execute stmt2 (if present) if condition is false.
start *task_name*	Start the specified task
stop *task_name*	Stop the specified task
function(*args*)	Call a function using the supplied arguments
var = *expression*	Evaluate expression and assign to variable
var += *expression*	Evaluate expression and add to variable
var -= *expression*	Evaluate expression and subtract from variable
var *= *expression*	Evaluate expression and multiply into variable
var /= *expression*	Evaluate expression and divide into variable
var |= *expression*	Evaluate expression and perform bitwise OR into variable
var &= *expression*	Evaluate expression and perform bitwise AND into variable
return	Return from function to the caller
expression	Evaluate expression

Conditions

Conditions are used within control statements to make decisions. In most cases, the condition will involve a comparison between expressions.

Condition	Meaning
true	always true
false	always false
expr1 == expr2	test if expressions are equal
expr1 != expr2	test if expressions are not equal
expr1 < expr2	test if one expression is less than another
expr1 <= expr2	test if one expression is less than or equal to another
expr1 > expr2	test if one expression is greater than another
expr1 >= expr2	test if one expression is greater than or equal to another
! condition	logical negation of a condition
cond1 && cond2	logical AND of two conditions (true if and only if both conditions are true)
cond1 || cond2	logical OR of two conditions (true if and only if at least one of the conditions are true)

Expressions

There are a number of different values that can be used within expressions including constants, variables, and sensor values. Note that SENSOR_1, SENSOR_2, and SENSOR_3 are macros that expand to SensorValue(0), SensorValue(1), and SensorValue(2) respectively.

Value	Description
number	A constant value (e.g. "123")
variable	A named variable (e.g "x")
Timer(*n*)	Value of timer n, where n is between 0 and 3

`Random(n)`	Random number between 0 and n
`SensorValue(n)`	Current value of sensor n, where n is between 0 and 2
`Watch()`	Value of system watch
`Message()`	Value of last received IR message

Values may be combined using operators. Several of the operators may only be used in evaluating constant expressions, which means that their operands must either be constants, or expressions involving nothing but constants. The operators are listed here in order of precedence (highest to lowest).

Operator	Description	Associativity	Restriction	Example
`abs``()` **`sign`**`()`	Absolute value Sign of operand	n/a n/a		`abs(x)` `sign(x)`
`++` `--`	Increment Decrement	left left	variables only variables only	`x++ or ++x` `x-- or --x`
`-` `~`	Unary minus Bitwise negation (unary)	right right	constant only	`-x` `~123`

`*`
`/`
`%`
	Multiplication
	Division
	Modulo
		left
		left
		left
			constant only	`x * y`
				`x / y`
				`123 % 4`

`+` `-`	Addition Subtraction	left left		`x + y` `x - y`
`<<` `>>`	Left shift Right shift	left left	constant only constant only	`123 << 4` `123 >> 4`
`&`	Bitwise AND	left		`x & y`
`^`	Bitwise XOR	left	constant only	`123 ^ 4`
`|`	Bitwise OR	left		`x | y`
`&&`	Logical AND	left	constant only	`123 && 4`
`||`	Logical OR	left	constant only	`123 || 4`

RCX Functions

Most of the functions require all arguments to be constant expressions (number or operations involving other constant expressions). The exceptions are functions that use a sensor as an argument, and those that can use any expression. In the case of sensors, the argument should be a sensor name: SENSOR_1, SENSOR_2, or SENSOR_3. In some cases there are predefined names (e.g. SENSOR_TOUCH) for appropriate constants.

Function	Description	Example
`SetSensor(`*`sensor`*`, `*`config`*`)`	Configure a sensor.	`SetSensor(SENSOR_1, SENSOR_TOUCH)`
`SetSensorMode(`*`sensor`*`, `*`mode`*`)`	Set sensor's mode	`SetSensor(SENSOR_2, SENSOR_MODE_PERCENT)`
`SetSensorType(`*`sensor`*`, `*`type`*`)`	Set sensor's type	`SetSensor(SENSOR_2, SENSOR_TYPE_LIGHT)`
`ClearSensor(`*`sensor`*`)`	Clear a sensor's value	`ClearSensor(SENSOR_3)`
`On(`*`outputs`*`)`	Turn on one or more outputs	`On(OUT_A + OUT_B)`
`Off(`*`outputs`*`)`	Turn off one or more outputs	`Off(OUT_C)`
`Float(`*`outputs`*`)`	Let the outputs float	`Float(OUT_B)`
`Fwd(`*`outputs`*`)`	Set outputs to forward direction	`Fwd(OUT_A)`
`Rev(`*`outputs`*`)`	Set outputs to backwards direction	`Rev(OUT_B)`
`Toggle(`*`outputs`*`)`	Flip the direction of outputs	`Toggle(OUT_C)`
`OnFwd(`*`outputs`*`)`	Turn on in forward direction	`OnFwd(OUT_A)`
`OnRev(`*`outputs`*`)`	Turn on in reverse direction	`OnRev(OUT_B)`
`OnFor(`*`outputs`*`, `*`time`*`)`	Turn on for specified	`OnFor(OUT_A, 200)`

	number of 100ths of a second. Time may be an expression.	
SetOutput(*outputs*, *mode*)	Set output mode	SetOutput(OUT_A, OUT_ON)
SetDirection(*outputs*, *dir*)	Set output direction	SetDirection(OUT_A, OUT_FWD)
SetPower(*outputs*, *power*)	Set output power level (0-7). Power may be an expression.	SetPower(OUT_A, 6)
Wait(*time*)	Wait for the specified amount of time in 100ths of a second. Time may be an expression.	Wait(x)
PlaySound(*sound*)	Play the specified sound (0-5).	PlaySound(SOUND_CLICK)
PlayTone(*freq*, *duration*)	Play a tone of the specified frequency for the specified amount of time (in 10ths of a second)	PlayTone(440, 5)
ClearTimer(*timer*)	Reset timer (0-3) to value 0	ClearTimer(0)
StopAllTasks()	Stop all currently running tasks	StopAllTasks()
SelectDisplay(*mode*)	Select one of 7 display modes: 0: system watch, 1-3: sensor value, 4-6: output setting. Mode may be an expression.	SelectDisplay(1)
SendMessage(*message*)	Send an IR message (1-255). Message may be an expression.	SendMessage(x)
ClearMessage()	Clear the IR message buffer	ClearMessage()
CreateDatalog(*size*)	Create a new datalog of the given size	CreateDatalog(100)
AddToDatalog(*value*)	Add a value to the datalog. The value may be an expression.	AddToDatalog(Timer(0))
SetWatch(*hours*, *minutes*)	Set the system watch value	SetWatch(1,30)
SetTxPower(*hi_lo*)	Set the infrared transmitter power level to low or high power	SetTxPower(TX_POWER_LO)

RCX Constants

Many of the values for RCX functions have named constants that can help make code more readable. Where possible, use a named constant rather than a raw value.

Sensor configurations for SetSensor()	SENSOR_TOUCH, SENSOR_LIGHT, SENSOR_ROTATION, SENSOR_CELSIUS, SENSOR_FAHRENHEIT, SENSOR_PULSE, SENSOR_EDGE
Modes for SetSensorMode()	SENSOR_MODE_RAW, SENSOR_MODE_BOOL, SENSOR_MODE_EDGE, SENSOR_MODE_PULSE, SENSOR_MODE_PERCENT, SENSOR_MODE_CELSIUS, SENSOR_MODE_FAHRENHEIT, SENSOR_MODE_ROTATION
Types for SetSensorType()	SENSOR_TYPE_TOUCH, SENSOR_TYPE_TEMPERATURE, SENSOR_TYPE_LIGHT, SENSOR_TYPE_ROTATION
Outputs for On(), Off(), etc.	OUT_A, OUT_B, OUT_C
Modes for SetOutput()	OUT_ON, OUT_OFF, OUT_FLOAT
Directions for SetDirection()	OUT_FWD, OUT_REV, OUT_TOGGLE
Output power for SetPower()	OUT_LOW, OUT_HALF, OUT_FULL
Sounds for PlaySound()	SOUND_CLICK, SOUND_DOUBLE_BEEP, SOUND_DOWN, SOUND_UP,

	SOUND_LOW_BEEP, SOUND_FAST_UP
Modes for SelectDisplay()	DISPLAY_WATCH, DISPLAY_SENSOR_1, DISPLAY_SENSOR_2, DISPLAY_SENSOR_3, DISPLAY_OUT_A, DISPLAY_OUT_B, DISPLAY_OUT_C
Tx power level for SetTxPower()	TX_POWER_LO, TX_POWER_HI

Keywords

Keywords are those words reserved by the NQC compiler for the language itself. It is an error to use any of these as the names of functions, tasks, or variables. The following keywords exist: **__sensor, abs, asm, break, const, continue, do, else, false, if, inline, int, repeat, return, sign, start, stop, sub, task, true, void, while.**

XIV. Final remarks

If you have worked your way through this tutorial you can now consider yourself an expert in NQC. If you have not done this up to now, it is time to start experimenting yourself. With creativity in design and programming you can make Lego robots do the most wonderful things.

This tutorial did not cover all aspects of the RCX Command Center. You are recommended to read the documentation at some stage. Also NQC is still in development. Future version might incorporate additional functionality. Many programming concepts were not treated in this tutorial. In particular, we did not consider learning behavior of robots nor other aspects of artificial intelligence.

It is also possible to steer a Lego robot directly from a PC. This requires you to write a program in a language like Visual Basic, Java or Delphi. It is also possible to let such a program work together with an NQC program running in the RCX itself. Such a combination is very powerful. If you are interested in this way of programming your robot, best start with downloading the spirit technical reference from the Lego MindStorms web site.

http://www.legomindstorms.com/

The web is a perfect source for additional information. Some other important starting points are on the links page of my own site:

http://www.cs.uu.nl/people/markov/lego/

and LUGNET, the LEGO® Users Group Network (unofficial):

http://www.lugnet.com/

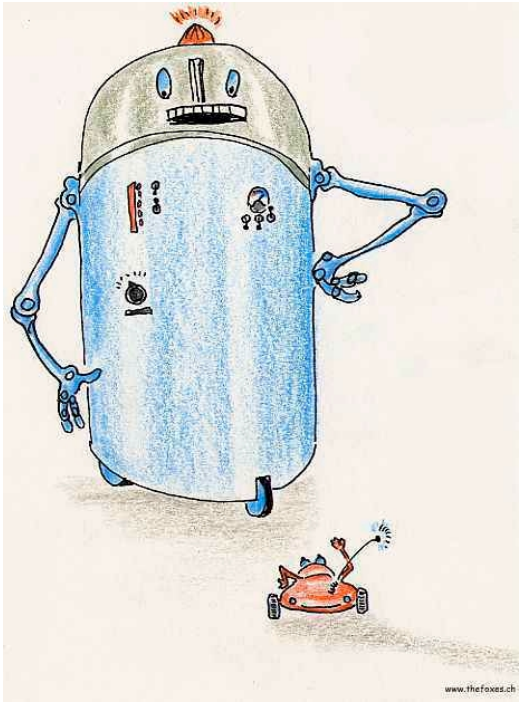
A lot of information can also be found in the newsgroup lugnet.robotics and lugnet.robotics.rcx.nqc at lugnet.com.

Berner Fachhochschule

Hochschule für
Technik und Architektur Bern

Künstliche Wesen

Nach dem Buch von Valentin Braitenberg zum Verhalten kybernetischer Vehikel.



Seminararbeit I96

Rolf Moser, Michael Dürig, Stefan Graf

Valentin Braitenberg und das Buch

Zur Person

Prof. Dr. Valentino Braitenberg arbeitet und wirkt an verschiedenen Instituten und Universitäten, wie dem Max-Planck-Institut Tübingen, der Universität Tübingen und der Universität von Trento in Rovereto in Italien.

Braitenberg beschäftigt sich hauptsächlich auf dem Gebiet der Hirnforschung. Aktuelle Forschungen betreffen z.B. die Bedeutung der Verbindungsmuster in der grauen Hirnmasse *«research on the meaning of patterns of connections within the grey substance of the brain»*, eine neurophysiologisch realistische Theorie der Sprachmechanismen *«a neurophysiologically realistic theory of language mechanisms»* oder die kognitiven Aspekte der Hirntheorie *«cognitive aspects of brain theory»*.

Zum Buch

Die amerikanische Originalausgabe erschien unter dem Titel «*Vehicles – Experiments in Synthetic Psychology»* 1984 bei MIT Press und wurde 1986 ins Deutsche übersetzt. Das Buch war ursprünglich von Braitenberg nur als scherzhaften Essay gedacht und musste vor der Veröffentlichung entsprechend überarbeitet werden.

Einleitung

Dieses Skript stellt im Wesentlichen eine Zusammenfassung des ersten Teils des Buchs «*Künstliche Wesen – Verhalten kybernetischer Vehikel»* von Valentin Braitenberg erschienen im Vieweg Verlag dar.
Im Zuge seines Buchs entwickelt Braitenberg in Gedanken vierzehn verschieden Wesen und untersucht ihr Verhalten und ihre Interaktionen mit der Umwelt und mit anderen künstlichen Wesen. Damit dies funktioniert fordert er den Leser jeweils auf die Wesen objektiv von aussen und ohne das Einbringen von Vorabwissen zu betrachten und zu beurteilen. Das heisst,

der Leser soll die Wesen mit den selben Augen betrachten wie er ein lebendiges Tier betrachten würde. Erst durch diese Betrachtungsweise entsteht das eigentlich interessante an den ansonsten relativ einfachen Vehikeln. Provokativ benutz er Begriffe wie Zuneigung, Aggression, Angst, Wissen und Denken in der Beschreibung des Verhaltens seiner Wesen, obwohl die Konstruktionsvorschriften jeweils rein technischer – wenn auch häufig hypothetischer – Natur sind. Er zeigt, dass das Verhalten von einfach zu konstruierenden Wesen von aussen betrachtet kompliziert und undurchschaubar zu sein scheint und deutet damit an, dass auch das Verhalten natürlicher Wesen unserer Umwelt durch einfachen Regeln motiviert sein könnte. Die Motivation für seine Wesen stammt aus seiner Arbeit als Hirnforscher. Die Konstruktionsvorschriften entsprechen jeweils Strukturen wie sie in ähnlicher Form grundsätzlich in biologischen Gehirnen auch vorhanden sind. Auf diese Parallelen geht er im zweiten Teil seines Buchs genauer ein.

Wir versuchen in dieser Zusammenfassung die Grundgedanken des Autors, seine Gedankenexperimente und seine Argumentationen zu erläutern. Dabei werden wir seine ersten sieben Wesen genauer beschreiben und uns eng an die Originaldarstellungen halten. Die restlichen Wesen fassen wir in einem Überblick zusammen. Gerade durch die relative Einfachheit der ersten Wesen bieten sich diese zur Darstellung des Grundgedankens an. Interessierten Lesern Empfehlen wir das äusserst interessante und unterhaltsame Buch von Valentin Braitenberg.

Streunen

Das erste Wesen besteht aus einem Sensor und einem Motor. Der Sensor nimmt eine Eigenschaft der Umgebung (z.B. die Temperatur) war und leitet diese Information an den Motor weiter. Der Motor läuft nun um so schneller, desto ausgeprägter die Eigenschaft ist auf die der Sensor reagiert. Dieses Wesen kann nur geradeaus fahren und nicht stehenbleiben. Es wird seine Geschwindigkeit in Warmen Gebieten erhöhen und in kühleren wird es verlangsamen.

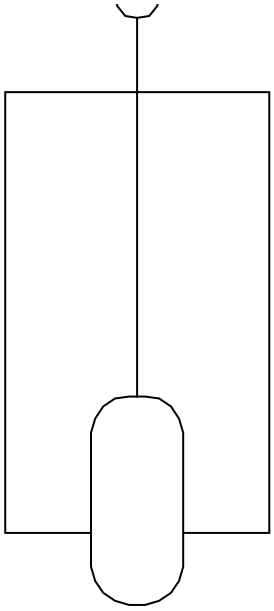


Fig. 1 Die Geschwindigkeit des Motors (abgerundeter Kasten am Hinterende) wird über einen Sensor geregelt (Halbkreis am Vorderende).

Stellt man sich nun ein solches Wesen in einem Teich herum schwimmend vor, würde man sagen, es ist ruhelos und mag kein warmes Wasser. Es ist aber ziemlich dumm, denn es kann weder zu den angenehm kalten Stellen zurückkehren noch dort stehe bleiben. In seiner Unruhe schiesst es jeweils darüber hinaus und gerät so wieder in wärmere Regionen. Auf jeden Fall würde man aber von einem solchen Wesen behaupten, dass es lebt, da ein totes Stück Materie sich kaum derart verhalten würde.

Furcht und Aggression

Das zweite Wesen ist eine Weiterentwicklung des ersten. Es besitzt statt einem Motor und einem Sensor jeweils zwei die an je einer Längsseite des Vehikels angebracht sind. Dadurch sind grundsätzlich zwei verschiedenen Wesen möglich: Eines mit gekreuzten Motor-Sensor Verbindungen und eines mit ungekreuzten.

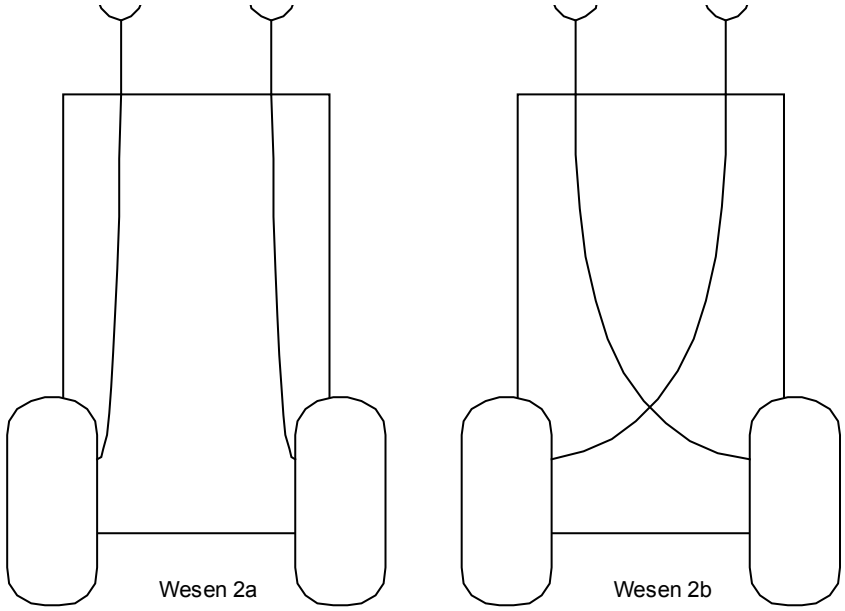


Fig. 2 Wesen 2 mit zwei Motoren und zwei Sensoren. Unterschiedliche Verknüpfung zwischen Motoren uns Sensoren möglich.

Das Verhalten der beiden Wesen ist Unterschiedlich: Wesen a fährt auf die Quelle die seine Sensoren erregt zu, wenn es sich gerade vor ihr befindet. Liegt die Quelle allerdings seitlich, wird der der Quelle zugewandte Sensor stärker erregt, der zugehörige Motor läuft schneller und das Wesen wendet sich von der Quelle ab.
Wesen b verhält sich gleich wie Wesen a wenn sich die Reizquelle direkt in Fahrrichtung liegt. Ist die nicht der Fall und liegt die Quelle etwas seitlich, Wendet sich dieses Wesen zur Quelle hin und fährt mit immer zunehmender Geschwindigkeit auf sie zu. Es gibt für dieses Wesen kein entrinnen, es wird schlussendlich mit hoher Geschwindigkeit auf die Quelle auffahren.

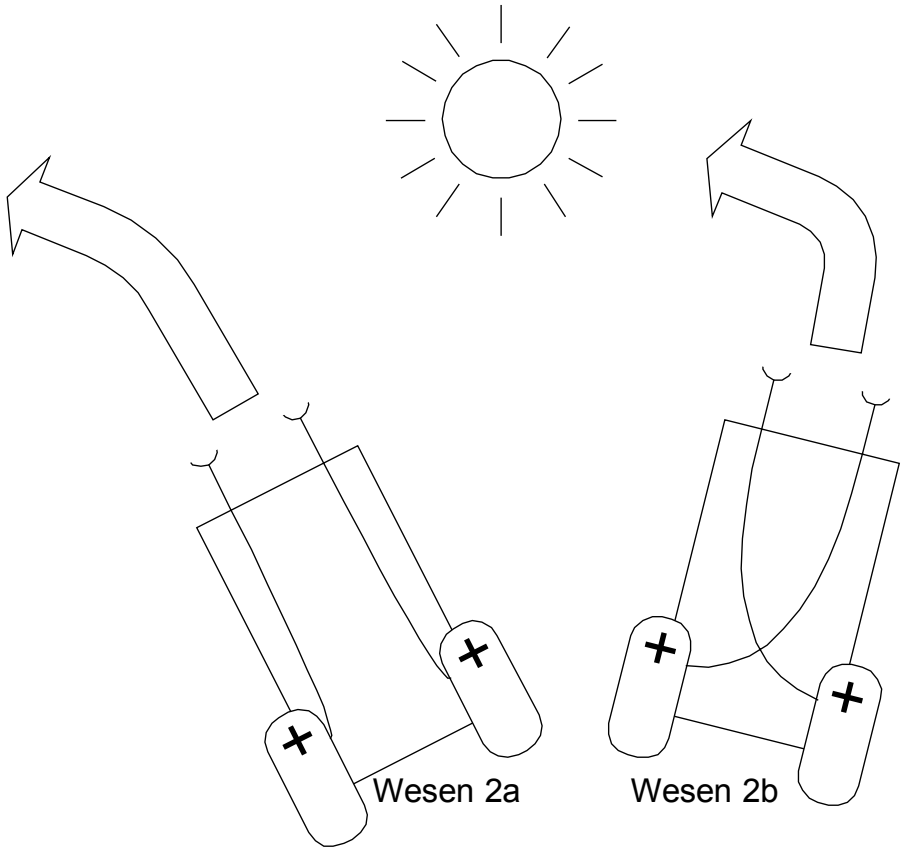


Fig. 3 Wesen 2a und 2b in der Nachbarschaft einer Reizquelle.

Beim Beobachten der Wesen in ihrer Welt entdeckt man, dass beide die Quelle nicht mögen. Allerdings verhalten sie sich nicht identisch. Wesen a fürchtet sich von ihr und wendet sich ab während Wesen b aggressiv reagiert, sich der Quelle zuwendet und sie schlussendlich heftig rammt wie wenn es sie zerstören wollte.

Liebe

Dieses Wesen ist eine Abwandlung des vorherigen: Je stärker ein Sensor reagiert, desto schwächer wird der zugehörige Motor aktiviert. Auch hier existieren die beiden Möglichkeiten gekreuzter und ungekreuzter Verbindungen.
Das Wesen mit den ungekreuzten Verbindungen wendet sich zu seitlich liegenden Quellen hin, weil der Sensor der näher bei der Reizquelle ist den Motor auf dieser Seite stärker verlangsamt als den auf der anderen Seite. Das Wesen geht so mit immer langsamerer Fahrt auf die Quelle zu und bleibt schlussendlich vor dieser stehen. Ähnlich verhält sich das Wesen mit den gekreuzten Verbindungen, allerdings bleibt dieses mit den Hinterteil der Quelle zugewandt stehen. Wo das erste Wesen vollkommen in den Bann der Quelle gezogen ist und

sich von nichts mehr von dieser lösen lässt, kann das zweite Wesen von einer weiteren – stärkeren – Quelle angezogen werden und sich so von seiner ursprünglichen Quelle lösen.

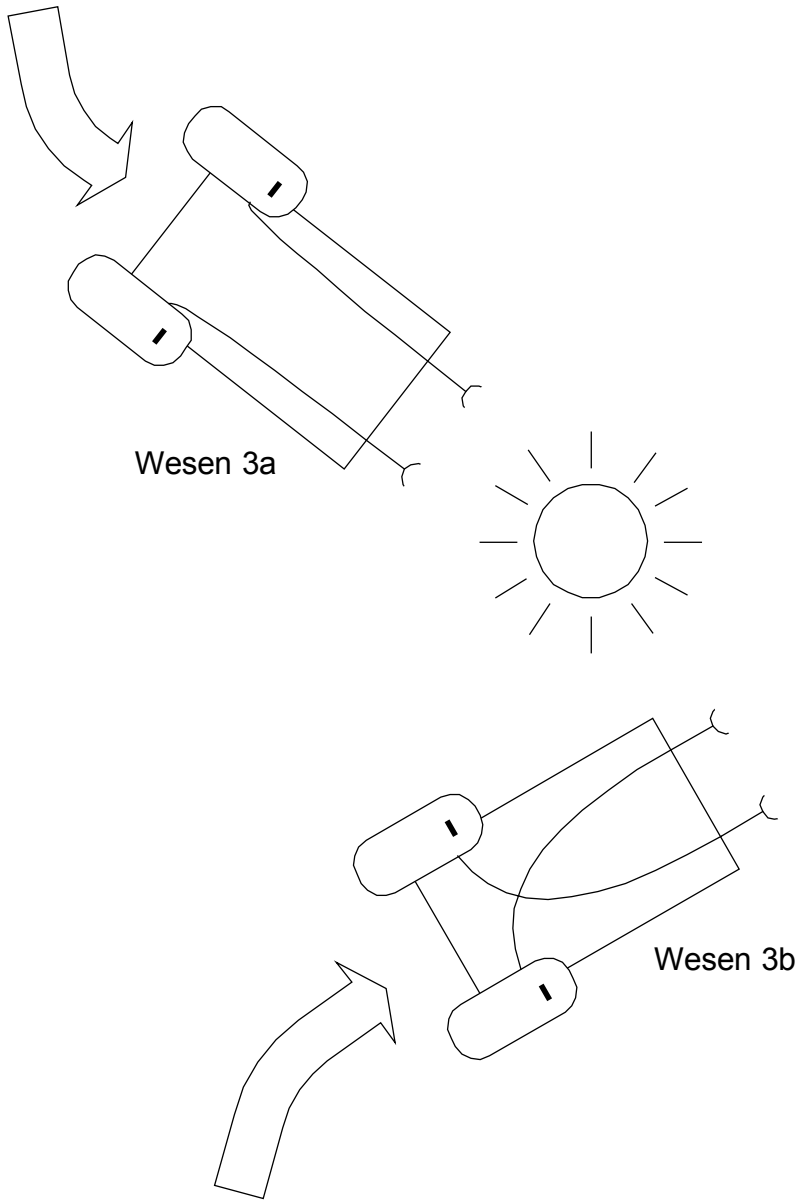


Fig. 4 Wesen 3a und 3b in der Nachbarschaft einer Reizquelle.

Diese beiden Wesen mögen die Reizquelle, allerdings in unterschiedlicher Weise. Wesen a liebt die Quelle unbedingt und beständig. Es geht auf sie zu und verharrt von da an auf alle Ewigkeit in stiller Bewunderung vor der Quelle. Wesen b hingegen liebt seine Quelle und möchte sie immer in der Nähe haben, ist jedoch auch immer für neues offen. Entdeckt es eine stärkere Quelle wird es mit dieser eine neue Liebe eingehen.
Dieses Wesen lässt sich noch erweitern. Man kann mehrere Sensoren einführen, die auf verschiedene Umgebungsreize wie Licht, Geruch, Temperatur und Konzentration von chemi-

schen Stoffen reagieren und diese Reize erregend oder hemmend auf einen oder beide Motoren weiterleitet. Wesen dieses Typs können bereits ein sehr differenziertes Verhalten zeigen. z.B. mögen sie keine heissen Stellen und wenden sich ab und sie hassen Lichtquellen, denn sie wenden sich auf die Glühbirnen zu und zerstören sie. Beobachtet man das Verhalten solcher Wesen, wird man sagen sie besitzen ein System von Werten: Sie wissen, dass Glühbirnen Wärme erzeugen wovor sie sich fürchten. Daher zerstören sie diese Bevor sie die Umgebung unnötig aufheizen können.

Wertung und Geschmack

Bei allen Bisherigen Wesen war die Abhängigkeit zwischen Sensoren und Motoren proportional oder zumindest monoton. Bei diesem Wesen fällt diese Einschränkung weg. Es sind jetzt alle möglichen Abhängigkeiten zwischen Sensoren und Motoren denkbar. So kann die Aktivierung eines Motors laufend zunehmen und ab einer gewissen Intensität der Wahrnehmung wieder abnehmen. Solche Wesen werden sich dann an den Orten mit den für sie am geeignetsten Bedingungen aufhalten bzw. sich auf komplizierten Bahnen zwischen diesen hin und her bewegen. Für einen Beobachter wird es unmöglich die Motivation für diese Bewegungen und die verzwickten Geschmäcker der Wesen durchschauen zu können die ihrem Verhalten zugrundeliegen. Man würde behaupten, diese Wesen werden von einer Vielzahl von Instinkten geleitet.
Wesen vier kann aber noch mehr wenn neue Verknüpfungen eingeführt werden. Dabei beeinflussen die Sensoren die Motoren nicht mehr kontinuierlich sondern sprunghaft. Der Reiz einer Quelle wird bis zu einer bestimmten Schwelle ignoriert, darüber wird der Motor dann aber plötzlich heftig aktiviert. Wesen die mit solchen Schwellwerten ausgerüstet sind werden einen Beobachter beeindrucken: Nähert man sich mit einem Köder einem dieser Wesen, wird es vorerst nicht reagieren, nachdem es sich dann aber entschlossen hat wird spontan rasch und entschieden handeln. Das Wesen kann also eine Entscheidung fällen. Es weiss ab wann es sich lohnt sich für etwas einzusetzen und lässt sich nicht durch jede Kleinigkeit anlocken. Man könnte sogar behaupten, dass wo Entscheidungen gefällt werden auch ein Wille sein muss.

Logik

Dieses Wesen baut auf die Entdeckung der Schwellenelemente beim letzten Wesen auf. Anstatt einfach ein Schwellenelement zwischen Sensor und Motor zu schalten, werden mehrere Schwellenelemente miteinander verschaltet. Dabei können mehrere Ausgänge auf einen Eingang oder ein Ausgang auf mehrere Eingänge wirken. Die Schwellenelemente können auch auf sich selbst zurück wirken oder auf Schwellenelemente die auf den eigenen Eingang wirken. Mit genügend Schwellenelementen lässt sich ein künstliches Gehirn (Neuronales Netz) bauen, das grundsätzlich die selben Aufgaben bewältigen kann wie ein Computer. Durch vorhandene Rückkopplungen im Netz ist sogar ein Gedächtnis vorhanden welches frühere Wahrnehmungen oder Berechnungen zwischenspeichern kann.

Die Wesen die mit solchen Gehirnen ausgestattet sind, werden sich derart kompliziert verhalten, dass ein Beobachter nicht ohne weiteres die Motivation und den Grund erkennen kann. Bei diesen Wesen zeigt sich sehr ausgeprägt, dass es viel leichter ist eine Maschine zu erfinden die bestimmte Fertigkeiten hat und diese dann zu beobachten, als ein Wesen von aussen zu betrachten und daraus zu versuchen auf seine innere Struktur zu schliessen. Die Konsequenz daraus ist, dass man bei der Beobachtung und Analyse eines Mechanismus dessen Komplexität zu überschätzten neigt.

Evolution

Die Wesen in diesem Kapitel stammen aus einer Massenproduktion: Einige Exemplare der bisherigen Wesen befinden sich auf einem Tisch und interagieren dort. Nun werden zufällig Wesen ausgewählt und kopiert und zusammen mit den Kopien wieder auf den Tisch gestellt. Natürlich schleichen sich beim Kopieren vereinzelt Fehler ein und einige Wesen werden mit der Zeit vom Tisch fallen und dabei zerstört. Die erfolgreicheren Wesen, denen es gelingt auf dem Tisch zu bleiben bewahren die guten Ideen die sich im Lauf der Zeit durch unbeabsichtigte Fehler beim kopieren einschleichen, während die Versager automatisch ausgeschieden werden.
Durch dieses darwinistische Selektionsprinzip werden mit der Zeit vollständig neue Wesen entstehen die in ihrem Verhalten perfekt an ihre Umgebung angepasst sind. Ein Analyse des Verhaltens solcher Wesen wird in den meisten Fällen misslingen. Die Verdrahtung, welches das Verhalten produziert, kann derart kompliziert sein, dass es für einen Beobachter unmöglich ist es zu durchschauen. Es scheint als ob eine geheimnisvolle schöpferische Kraft die Wesen erzeugt hat.

Begriffe und Abstraktion

Bereits mit den einfachen Schwellenelementen konnten die Wesen Entscheidungen fällen. Es schien das für die Entscheidung notwendige Wissen vorhanden zu sein. Dabei handelte es sich jedoch um den Wesen ‚angeborenes' Wissen. Keines der Wesen hatte die Möglichkeit zur Adaption an neue Bedingungen.
Die Wesen dieses Kapitels werden mit einem neuen Element ausgerüstet: Zwischen allen Schwellenelementen wird eine Verbindung hergestellt. Jede dieser Verbindungen wird jeweils dann aktiviert wenn die Schwellenelemente die sie verbindet gleichzeitig aktiviert sind. Die Aktivierung hält für einige Zeit an, auch wenn die Aktivierung der Schwellenelemente bereits wieder nachgelassen hat. Ist nun ein Schwellenelement aktiviert bevor die Aktivierung der Verbindung zu einem anderen Schwellenelement nachgelassen hat, wird dieses ebenfalls aktiviert. Es handelt sich hier um eine Assoziation die das Wesen zu seiner Lebzeit gebildet hat: Wenn aggressive Wesen üblicherweise rot sind, dann sind die Schwellenelemente für rot und aggressiv häufig gleichzeitig aktiviert und somit auch die dazwischen geschaltete Verbindung. Trifft nun ein Wesen auf ein anderes, rotes Wesen wird das Schwellenelemente für rot aktiviert und über die aktivierte Verbindung das Schwellenelemente für aggressiv. Das

rote Wesen wird dann als aggressiv eingestuft. Allgemein betrachtet hat das Wesen einen Begriff gebildet: Wann immer aggressives Wesen naht, wird es ‚Rot sehen'. D.h. es wird etwas tun das es vorher nur tat, wenn es dir Farbe rot sah.
Man kann sich nun auf den Standpunkt stellen, dass es sich hier um eine triviale Assoziation handelt, und dass die Wesen nicht fähig sind zu abstrahieren. Allerdings kann man sich auch ein Wesen vorstellen, das gelernt hat, dass graue Wesen friedlich sind und von roten jeweils Aggression erfahren hat. Dieses Wesen wird von einem Blauen ebenfalls zurückschrecken und es als aggressiv einstufen, nur aus der Tatsache heraus weil es nicht grau ist. Ein Beobachter dieses Wesens wird sagen können, es habe eine Abstraktion von den Begriffen ‚Rot' und ‚Blau' zum allgemeinen Begriff ‚Farbe' vorgenommen.

Raum und Zeit

Das Verhalten der bisherigen Wesen basierte jeweils auf einfachen Wahrnehmungen und deren Assoziation. Rüstet man die Wesen mit besseren Sensoren aus, Sensoren die es ermöglichen die räumlichen Beziehungen von Objekten wahrzunehmen erhalten die Wesen die Möglichkeit sich – mittels ihrer Fähigkeiten zur Assoziation – ein eigenes subjektives Abbild ihrer Umgebung zu schaffen. Dieses Abbild enthält beispielsweise Nachbarschaftsbeziehungen von Objekten so wie sie vom dem Wesen selbst wahrgenommen werden. Es kann auf einfache Weise zur Überprüfung der Realität von Dingen dienen, indem momentane Wahrnehmungen mit Früheren verglichen werden. Dieses ‚Weltbild' ist jedoch vollkommen statisch. Es ist nur dazu geeignet rein räumliche Ereignisse (Objekte) abzubilden, nicht jedoch zeitliche Ereignisfolgen. Das Nacheinander von Ereignissen ist wichtig zum Abbilden von Kausalbeziehungen wie Ursache und Wirkung. Um den Wesen die Möglichkeit zur Bildung eines Abbilds der dynamischen Bestandteile der Welt zu geben, werden alle Schwellenelemente miteinander durch eine neue Verbindung verknüpft: Diese leitet die Aktivierung nur in einer Richtung und mit einer gewissen Verzögerung weiter. So erhalten die Wesen einen Mechanismus um Ereignisse die jeweils zeitlich verzögert in einer bestimmten Reihenfolge stattfinden zu assoziieren.
Die so konstruierten Wesen werden deutlich ausgebildete Reaktionen bei Ereignissen zeigen die regelmässig bestimmte Folgen haben: Die Situation eines Wesen das mit hoher Geschwindigkeit auf ein Hindernis zufährt, wird nach der Beobachtung einiger Kollisionen sofort richtig mit ‚Gefahr' assoziiert.

Denken

Man kann mit recht behaupten, dass die bislang konstruierten Wesen nichts können das über gewöhnliches Lernen hinausgeht. Insbesondere können sie nicht denken, denn Denken ist ein Vorgang der sich über einen längeren Zeitraum erstrecken kann bei dem eine lange Folge verschiedenere Zustände im Gehirn aufeinander folgen. Um dies zu erreichen, müssen sich die Schwellenelemente gegenseitig und rückgekoppelt aktivieren können. Damit es dabei nicht zu unkontrollierten Explosionen der Aktivität (Epilepsie) kommt, muss man einen Kon-

trollmechanismus einbauen, der die Aktivierungsschwelle der Elemente global anheben und senken kann. Bei allgemein niedriger Aktivität wird diese Schwelle abgesenkt und bei hoher angehoben. Das Gehirn eines solchen Wesens wird nun – einmal durch eine Wahrnehmung angeregt – laufend von einem Zustand in den nächsten Wechseln, wobei der jeweilig vorausgehende Zustand und die aktuellen Wahrnehmungen den Folgezustand bestimmen. Das Wesen scheint sich Gedanken zu machen!
Das so konstruierte Gehirn entspricht der Iteration einer Funktion [$Z_{n+1} = f(Z_n)$] und seine Zustände sind im Allgemeinen nicht vorhersagbar. Obwohl der Mechanismus des Gehirns bekannt ist, lässt sich ein zukünftiger Zustand und somit das Verhalten des Wesens nicht voraussagen. Beobachtet man nun ein solches Wesen, wird man behaupten, es besitze einen freien Wille, denn es lässt sich nicht voraussagen was es im nächsten Moment tun wird und auch das Wesen selbst wird dies nicht können.

Zitat Braitenberg: *«Wer immer Tiere und Menschen erschaffen hat, wollte vielleicht nicht mehr als wir, die Schöpfer der Vehikel: den Geschöpfen etwas mitgeben, das für jeden, der mit ihnen zu tun hat, wie freier Wille aussieht. Damit wäre wenigstens der schäbigen Ausbeutung eines Individuums durch Beobachtung und Vorhersage seines Tuns ein Ende gesetzt. Und zu des Individuums Stolz und Freude ist es selbst nicht imstande, genau vorherzusagen, welcher Gedanke sich im nächsten Augenblick in seinem Gehirn einstellen wird, und es mag daraus den Schluss ziehen, dass seine Entscheidungen am Anfang, nicht am Ende von Kausalketten stehen.»*

Voraussagen und Ziel

Auch wenn die Wesen denken, so bleibt ihr Wirken weitgehend ziellos. Es scheint keinen Zweck zu geben, der ihr Verhalten bestimmt, sie scheinen keinen Weg zu einem Ziel zu folgen. Um dies zu erreichen müssen die Wesen Voraussagen machen können. Sie müssen zu erwartende Ereignisse voraussehen können, damit sie die Konsequenzen ihrer Handlungen abschätzen können. Das Gehirn der Wesen lässt sich leicht erweitern um dies zu erreichen: Man muss einzig einen Mechanismus einbauen, der es ermöglicht gespeicherte Ereignisfolgen abzuspielen. So können die Wesen verschiedene Ereignisfolgen durchspielen, nach ihrem Geschmack bewerten und schliessend ihre Handlung danach wählen. Wesen dieser Art werden sich mit einer deutlichen Bestimmung durch ihre Welt bewegen. Sie scheinen immer hinter etwas her zu sein das oft von aussen nicht erkennbar ist. Sie scheinen einem Ziel oder vielleicht auch ihrem Traum nachzujagen.