# LegOS HOWTO


## Luis Villa

### luge@users.sourceforge.net


## About LegOS

To quote the official LegOS homepage (http://www.noga.de/legOS/), maintained by the author, Markus L. Noga, LegOS is an

> independent LEGO Mindstorms OS. Offers preemptive multitasking, energy saving, dynamic memory management, POSIX semaphores, native access to display, buttons, IR communication, motors and sensors.

In contrast to the standard Lego Group software, LegOS offers serious programmers the ability to write programs for the Lego Mindstorms in C and C++, with few significant limitations. Among the benefits cited by Markus (in comparison to the standard software):

- "User tasks are executed as native code, not interpreted."
- "More memory. Forget 32 variables. Think 32k."
- "Full-featured language. Use the same gcc that compiles Linux." The standard software uses graphical building blocks instead of a text-based language.
- "Priority-based preemptive multitasking."
- "Real process synchronization with POSIX semaphores."
- "Fine-grained hardware control: complete LCD control, raw mode IR."

I'll add:

- Random number generation. Real unpredictability.
- Floating point math- use doubles and floats without problems.
- Multiple programs. You can now store up to eight programs on your RCX at once, and choose between them.

However, there are certain limitations (which is why this is a programmer's tool for the Mindstorms!):

- "System is unstable - this is beta." I'll add that 0.2.4 is the most stable release of legOS yet- some small but persistent bugs have finally been eliminated.
- "GCC and binutils required."
- "C programming knowledge required."

## About this LegOS HOWTO

This is the second major revision of the legOS HOWTO. When I started playing with LegOS, I had no experience with LegOS, and was very rusty with what rudimentary C I had. At that time, there was no documenation for legOS, other than the API docs. As a result, it took me a while to get up to speed, particularly since I was learning both LegOS and the RCX at the same time. With that experience in mind, I wrote the HOWTO as an intro to legOS and the RCX.

This is not intended to be a full technical manual. The API documentation at http://legOS.sourceforge.net/API/ fills that role. In cases where the API documentation is lacking (it is auto-generated, and sometimes it shows), the code itself is very well documented for those who are more technically inclined. Rather, the HOWTO is directed squarely at an audience with no RCX experience, and only barely above intro-level C or C++ experience- i.e., a grasp of most programming concepts and data structures, but not necessarily more advanced topics like threading and semaphores. This pretty much describes a class of intermediate level CS students who have just had an RCX plopped in their lap for a class, or someone who is interested in the RCX but doesn't have access to a Windows machine to run the Lego Group software. I am a member of the Windows free group myself, and I hope that CPS 196 was not the last class of its type. My goal for this HOWTO is to serve as a "quick start" guide for these (and similar) groups, or a companion to the source for anyone who is more advanced.

Just because I've focused on beginners for now does not mean that I don't want the HOWTO to cover advanced topics. I'm very open to this, since I hope anyone who gets a start from this will use the time I've saved them to do more than just simple projects. Unfortunately, however, what little progress I had made was basically clobbered by the switch to 0.2.x. At the moment, my most glaring area of ignorance is the legOS Network Protocol, but there are other areas as well. Anyone who has done these things and wants to contribute for the more advanced reader (as well as my own edification) is welcome to.

## About the Author

The author is a college student with not as much time on his hands to play with Lego as he would like :) He can be contacted at <luge@users.sourceforge.net>, except during college basketball season, when he can usually be found on or around ESPN.

I try to respond to all emails; however, this is not always possible. If you have a general legOS question (as opposed to one specifically about the HOWTO) then you are much better off asking the fine people at <lugnet.robotics.rcx.legos@lugnet.com>. I'm usually one of them, so you'll get me plus lots of other developers at the same time.

## Development Platforms

This document and most of my code was written on various Linux systems.As a result, I should note that when dealing with system commands in the install section, I have assumed that the reader is on a Unix system of some type, with which the reader is familiar (i.e., the reader can use tar, gunzip, etc.)

If you are on a Windows system, please read section 2.2 to install LegOS, or section 2.4 look at another option. The rest of the documentation should be completely cross-platform- i.e., it should apply to whatever OS you are using to host legOS, since it is mostly legOS specific.

There are also many places in the text where files need to be opened tolook around. Again, I assume that whatever system you are working on has some type of text or code editor, so I leave the option of how to work with files up to the reader.

## Current legOS version

The current version of the HOWTO has been written for legOS 0.2.4. However, it was originally written with legOS 0.1.7. I have done my best to remove anachronisms from the text, but I'm sure that effort is incomplete. Please e-mail me at <luge@users.sourceforge.net>if you find any.

## Disclaimer/Legal/Technical Stuff

Like LegOS itself, this HOWTO is not in any way endorsed or supported or even authorized by The Lego Group.

I have written this under the Linux Documenation Project (LDP) Copying License. In short, this document can be copied and distributed freely, but cannot be modified in any way without permission from me. The full text of the LDP License is available at

http://metalab.unc.edu/LDP/LDP-COPYRIGHT.html, and you can reach me at
`<luge@users.sourceforge.net>`.

## Document Source

This document is now maintained in the DocBook format. The DocBook source is available here. (HOWTO.sgml) If you want to do serious work on the HOWTO, I suggest sending me patches against this source. To find out more about DocBook, I suggest reading DocBook: The Definitive Guide (http://docbook.org/tdg/html/docbook.html) or (more concisely) the LDP Author's Guide (http://linuxdoc.org/LDP/LDP-Author-Guide/).

## Printable Version of This Document

A PostScript version of this document is available here. (HOWTO.ps)

## Acknowledgements


- Markus L. Noga, for writing LegOS.
- Kekoa Proudfoot, for doing much of the early internals workthat allowed LegOS to be written. (Check out his RCX Internals Page. (http://graphics.stanford.EDU/~kekoa/rcx/))
- Dave Baum. Dave gave me the opportunity to spend a summer working mainly on legOS, which gave me the time to re-write the HOWTO and assume maintainership of the program. The outcome of this opportunity was the book Extreme Mindstorms, which you can now buy from Amazon. (http://www.amazon.com/exec/obidos/ASIN/1893115844/tieguyorg) This is a pretty shameless plug, but I'm very proud of the book and it offers some serious insight into the kinds of things you can do with legOS.
- Dr. Michael Littman (formerly of Duke, now at AT&T), who allowed me to fulfill a childhood dream by getting paid to play with Lego.
- The entire community at lugnet.com, without whom there would have been only the smallest of audiences (and the most minimal of advice) on the writing of this HOWTO.
- VA Linux, who sponsor sourceforge.net, which makes it possible for all legOS work to be housed (for free!) under one roof. If you run a Free Software (http://www.gnu.org/philosophy/free-sw.html) project of your own, I strongly recommend looking into Sourceforge, especially if you want to distribute large binaries or data files and face charges for bandwidth use on your normal ISP.

## Contributors

Because so many people have contributed to early versions of this documentation, and it is difficult to recognize them individually in the text, I've added this Contributors section. This doesn't replace or supplant the traditional CONTRIBUTORS file in any way; please make sure to read and remember that as well. There are also many names scattered throughout Section 8.1, the Changelog.

These people below have helped out in ways that were once specifically mentioned in the text, but have now been obsoleted. This doesn't lessen their contribution, though, so they get a mention here:

- Phil Watt; Paolo Masetti; Gavin Smyth; Brian Stormont; for the Cygwin instructions for Windows in their various iterations.

# Setting up LegOS

Unlike the software provided by the Lego Corporation, LegOS requires a compiler for your code, compilation of the legOS kernel, and special download tools for the kernel and your programs. I will discuss each of the components in turn, including download sites and setup instructions.

## Setting Up The Compiler Under Unix.

LegOS is compiled using the egcs compiler from Cygnus (http://egcs.cygnus.com), along with the binutils package from the GNU project. While these are present on nearly every Linux system, they must be re-compiled in order to serve as a cross-compiler for the Hitachi H8300 chip in the RCX.

### A Note About C++

The compilers described and linked to below will not be able to use legOS's new C++ support. To use that, you'll have to build and compile gcc 2.95 instead of the older egcs. To see more on the use of C++ in legOS, check out the section called *C++ under legOS*.

## Linux and Other Unix-compatible Executables.

Normally, I prefer to compile my own. However, this is not a fool-proof solution, and can easily fail. If you don't want to try this, or have tried, and have had it fail, some pre-compiled binaries are available:

- Redhat RPMs (http://legOS.sourceforge.net/files/linux/redhat/) are available here. These were built against RedHat 6.2, and should install cleanly with the normal rpm tools on any version of RedHat greater than 6.0.
- Debian, compiled against glibc 2.1 (http://legOS.sourceforge.net/files/linux/debian/)These are just tarballs, that you should be able to use with Debian 2.1, 2.2, or Debian unstable. Anyone who wants to contribute .deb files is welcome to :)

At least some people have indicated success in building these tools on Solaris Unix. However, it would appear that because of differences in I/O, it may not be possible (without a great deal of work) to get the rest of the system running on Solaris. For more information, check out the following Lugnet threads:

- Post #487 (http://www.lugnet.com/robotics/rcx/legos/?n=487)
- Post #605 (http://www.lugnet.com/robotics/rcx/legos/?n=605)
- Post #495 (http://www.lugnet.com/robotics/rcx/legos/?n=495)
- Post #515 (http://www.lugnet.com/robotics/rcx/legos/?n=515)

It is important to recognize that even if you do not build your own compiler, and these pre-compiled tools work for you, you will still need the basic development tools (like make) installed on your system. Most Unix-compatible (which I will abbreviate *nix) systems should come with these. If you are using Linux, in Redhat, glint or gnome-rpm should have a directory called "C/C++ development tools", and in Debian, use dselect to get the Standard packages in the devel group.

## Building *nix Executables.

If you want to build your own, here are some step by step instructions, adapted from Randal Loomis's linuxgcc-H8300-HOWTO.

1. Get the necessary archive files: egcs and binutils. They are here for egcs (ftp://ftp.freesoftware.com/pub/sourceware/gcc/old-releases/egcs/) and here for binutils. (ftp://ftp.gnu.org/gnu/binutils/) You'll need the file egcs-1.1.2.tar.bz2 and binutils-2.9.1.tar.gz. It is likely that these instructions will not work with gcc 2.95, because it uses different options for

configuration. However, the bug that prevented the use of gcc 2.95 to build legOS has been fixed, so if you can convince 2.95 to compile, you should be able to use it to build legOS programs.

2. Create the directory /usr/local/crossgcc. If you don't have space (roughly 130 megs) in /usr/local, use a symlink (on my machine, /usr/local/crossgcc is a link to /home/luis/crossgcc). Alternately, if you can't access /usr/local, use a directory of your own choice, but remember it for future use.

3. In the directory you've downloaded binutils and egcs into, type

   ```
   tar xvzf egcs-1.1.2.tar.gz binutils-2.9.1.tar.gz
   ```

   to expand and untar the files.

4. In the directory you used in step #3, create working directories for the build process with the following commands:

   ```
   mkdir build-binutils
   mkdir build-egcs
   ```

5. Build the binutils with the following commands:

   ```
   cd build-binutils
   ../binutils-2.9.1/configure --target=h8300-hitachi-hms --
   prefix=/usr/local/crossgcc --exec-prefix=/usr/local/crossgcc/h8300-
   hitachi-hms
   make all
   make install
   ```

6. Add the new binutils to your PATH with one of these commands (which one works for you is shell-dependent):

   ```
   export PATH=/usr/local/crossgcc/h8300-hitachi-hms/bin:$PATH
   ```

   or

   ```
   setenv PATH /usr/local/crossgcc/h8300-hitachi-hms/bin:$PATH
   ```

7. Configure the cross-compiler with the following commands:

   ```
   cd ../build-egcs
   ../egcs-1.1.2/configure
   --target=h8300-hitachi-hms --prefix=/usr/local/crossgcc --exec-
   prefix=/usr/local/crossgcc/h8300-hitachi-hms --enable-target-optspace
   ```

8. At this point, you must edit the Makefile for the cross-compiler. Opening it in your editor of choice, find the line which starts with CFLAGS. After the options that are already there, add the string "-Dinhibit_libc" at the end.

9. Now, type the command:

   ```
   make cross LANGUAGES="c c++"
   ```

10. At this point, the compilation will procede for quite some time. On my Pentium 166, it takes about 20-25 minutes. The following error message should then be generated:

```
checking whether the C compiler (/home/luis/build-egcs/gcc/xgcc -
B /home/luis/build-egcs/gcc -g -Os ) works
configure: error: installation or configuration problem:
C compiler cannot create executables.
```

11. The compilation fails at this point because it cannot find an appropriate standard C library. You can safely ignore this error, since LegOS provides the necessary replacements. Proceed to step 11. If you get another message which refers to libgcc.a, it indicates that you have a bad path. Check to make sure you executed step 6 correctly, and then execute the command "make clean" and re-do steps 7 and 8.
12. Install the compiler with the command:

```
make LANGUAGES="c c++" install
```

This should create a cross-compiler in /usr/local/crossgcc, or the directory you have passed using --prefix.

If your attempt at compilation generates an error that is not covered in this section, the folks on the crossgcc mailing list (MailTo:crossgcc@sourceware.cygnus.com) have been more than helpful to me. This is a majordomo type mailing list- if you don't subscribe, please remind them to cc: you a copy of their replies.

## Setting Up The Compiler Under Windows.

Running legOS under Windows is the longest ongoing saga in the legOS community. By my count, at least six different people have given a substantial amount of time to various attempts to explain and control legOS under windows.

There are two current solutions. Each are based on the same GNU tools that are used on Linux, however, one runs within DOS while the other runs within the cygwin environment (which itself runs in DOS.) Instructions for the first (and generally simpler) solution can be found at the page of Rossz Vamos-Wentworth. His page is pointed at from this Sourceforge page: http://legOS.sourceforge.net/files/windows/djgpp/. Be aware that this doesn't run on W2K.

The second option, which is more complex to get running but probably also slightly more powerful, is based on the cygwin tools. Following in the "long" tradition of cygwin users, Paolo Masetti has provided instructions for 0.2.4 that can be found at this sourceforge page. (http://legOS.sourceforge.net/files/windows/cygwin/)

Both of these pages are kept reasonably up-to-date and should give solid advice. Unfortunately, at the moment, DJGPP only works with legOS 0.2.3, but we are working on it for 0.2.4 and cleaning up a lot of code in the process.

### Historical Windows URLs.

These are here because I haven't had the chance to test these yet, and because I feel like I ought to have them around somewhere if they are still good.

Lego page (http://www.beesknees.freeserve.co.uk/lego/) - original Cygwin page

Brian Stormont (http://ex.stormyprods.com/lego/legOS.html)- howto build compiler for Cygwin

## Setting up LegOS itself.

### Getting LegOS.

The files which comprise LegOS are available here. (http://legOS.sourceforge.net/files/common/) The current version is 0.2.4. If you'd like to get a more current version from CVS, there are instructions on how to do that later in the HOWTO, at the section called *Getting legOS Development Versions From CVS*. If you do choose to get legOS from CVS, remember that it is (as of this publication) in pretty heavy development, and so it may be unusable on any given day (though we try very hard to prevent this.)

### Makefile and Tool Configuration.

Once you have the cross-compilers compiled and installed as per section2.1 or 2.2, the file Makefile.common (in the legOS directory) may have to be modified to reflect the location of the compiler on your system. If you are running Cygwin or DJGPP, and have made the modifications to your PATH specified in those instructions, you can safely skip the next paragraph.

In the file Makefile.common, line 23 (which starts with "TOOLPREFIX=") should be set to the location you passed to [egcs-1.1.2]$./configure, plus the string "h8300-hitachi-hms/bin/h8300-hitachi-hms-." In my system (and the example used in Section 2.1.2) this would be /usr/local/crossgcc/h8300-hitachi-hms/bin/h8300-hitachi-hms-.

You are now set up to compile the kernel and associated tools. To do this, type make in your legOS directory. This should create a legOS kernel in the legOS/boot directory. To find it, look for the filename legOS.srec. The util/ directory should also have **firmdl3** and **dll**, assuming that gcc is correctly configured on your system. If those aren't built, then you'll need to fix gcc and build them before proceeding.

## Using LegOS Without Your Own Compiler.

Once upon a time, there were two web-based compilers for LegOS. You could cut and paste your code into their webpage, and they'd post a URL where you can download the .srec file. Unfortunately, they both use legOS 0.1.x, so I include them here only as possible inspiration for others who might want to do the same with legOS 0.2.x. In fact, if you have the time to write the code in php, drop me a line and we can get it set up at sourceforge (maybe ;)

Here is the remaining historic URL:

Dave Madden's web compiler (https://vheissu.mersenne.com/~dhm/compile-legOS.html)

Both of these sites were basically the same, using the 990330 snapshot of 0.1.7. The one that is gone gave you the option of getting the s-record emailed to you. You will still need to compile firmdl or find another method to download the outputted .srec file to the RCX.

## Testing your compiler by compiling a simple program

Before I go any further, I should point out that this process assumes youhave actually built a robot :-). The example code I give in the section called *Sample Code* assume a simple design, with motors (and presumably wheels or treads) attached to output ports A and C, a touch sensor on input port 1, and a light sensor on input 2. Since this is mainly a LegOS document, I have no formal advice on construction here. The Constructopedia that comes with the Mindstorm kit has several good basic robots, from which more complicated structures can be built. You may also want to check out the 6.270 homepage (http://www.mit.edu/courses/6.270/home.html) at MIT, which includes these notes (http://web.mit.edu/6.270/www/notes/chap7.ps) on building solid structures and creating drivetrains. A related course at the University of West Florida has similar instructions (http://www.coginst.uwf.edu/lego/designexamples.html) with pictures.

### Compiling a legOS program into a .lx file

Unlike version 0.1.x, legOS 0.2.x has it's own file format- .lx. Instead of downloading the entire OS everytime you want to test a new program, you just download a .lx file.

Compiling a .lx is pretty simple. Under your legOS directory, create a sub-directory. I call mine "code", but "projects" or some such would also work fine. Copy the Makefile from the legOS/demo/ directory into your new directory, and put your code into the same directory. Then type make demo.lx (where demo is your the name of your .c file) and voila- you should get demo.o and demo.lx files.

## Putting a kernel into the RCX

Once the kernel has been compiled, firmdl3 is used to download the kernel to the robot itself. Firmdl3 is pretty straightforward, and you should only need to use it occasionally, since it is only used for the kernel. To test it, run it to get the kernel into the RCX. While in your legOS directory, type util/firmdl3 boot/legOS.srec. If your tower is positioned correctly, and the RCX is turned on, you should see a series of numbers flash by on the LCD of the RCX. When done, you'll see a - on the right side of the screen. (As a pseudo-historical aside, the .srec extension indicates that the legOS kernel is stored as an s-record. (http://www.objsw.com/CrossGCC/FAQ-8.html) In legOS 0.1.x, this was the standard file type for all legOS programs.)

### Possible firmdl3 errors.

I've tried to show the exact error messages outputted by firmdl3. Since the actual errors aren't quite perfect, I won't correct them.

- *"delete firmware failed"* Because of a limitation in the Lego firmware, firmdl3 cannot delete LegOS. Instead, before downloading the new firmware, you must first stop the old LegOS program, and then hold down the Prgm button while hitting the On/Off switch. This will delete the old LegOS firmware, allowing a new download. Alternately, if you see no green light on the IR box and get this error message, it may indicate you have a dead battery in the IR tower.
- *"/dev/ttyS0: Permission denied"*, "com1: Permission denied", or a similar message. The first part of the error message is the output port that firmdl3 is trying to use. This can mean two things. Either you have specified the wrong device or you have specified the correct device but do not have permission to access it. If it is an access problem, speak to your sysadmin. If not, you can use the --tty argument to specify another device. For example, to specify the standard device under linux, try "firmdl3 --tty /dev/ttyS0". Under cygwin, you probably probably want to use "firmdl3 --tty com1". If you discover that this is the case, you may wish to recompile firmdl3 to make your setting the default.
- *"/dev/ttyS0: Input/output error"* or a similar message.You forgot to plug the tower in, most likely.
- *"unlock firmware failed"* This indicates that there was a slight corruption in the download. Clear legOS and download it again.
- *"no response from RCX"* This error should be called "you forgot to turn on your RCX, moron." If I hadn't done it about 10 million times, I'd submit the patch to change the error myself.

- If firmdl3 gives you repeated problems, you may want to try using the --slow argument, which will make your download much slower but can correct for certain errors caused by ambient light.

## Downloading the .lx to the robot using dll

First, make sure you have the legOS kernel in your RCX. If you've been following my instructions, you've already done this. If not, go back to section 2.6 and use firmdl3 to download the kernel to the RCX.

Once you've got the kernel installed, to download the .lx file you have just created, go to the util/ directory, and type ./dll ../code/demo.lx This will begin the download process, which shouldn't take too long. You'll note that the download protocol is very fault tolerant- you should be able to move the bot or temporarily block the download, and still have it recover.

dll is a flexible tool with lots of useful options. The two more important are -p and -e. -pX allows the user to specify a program slot from 0-7. In this way, you can store multiple programs in your RCX. Once downloaded, use the "PRGM" button to rotate between the various programs you have downloaded. If you don't use -p, then dll defaults to downloading the program to the first program slot (which is numbered 0, of course). The -e flag automatically runs the program as soon as it is downloaded. To see the rest of the commands available, just type "dll" without any arguments.

Under Linux, I find it useful to add a symbolic link from my working directory to dll, so that I don't have to change directories or use sets of ../../ to download code once it is compiled in my working directory.

### dll error messages

Unfortunately, there is basically only one dll error message, and it applies to most possible errors, including a disconnected IR tower, an RCX that is turned off, an RCX that doesn't have legOS, and an RCX that has legOS and is already running a program. So, check all those things when dll fails.

# Mechanical control of the RCX.

In this section, I will describe the functions used to control the basic mechanical functions of the RCX. I have attempted to list the files in which the relevant functions can be found for all the functions I have described, as well as links to the relevant pages in the API documentation at the LegOS website. If the API docs are lacking info you need, dig into the LegOS code itself. It is not very complicated stuff (for the most part) and someone with reasonable C skill should be able to figure out what is going on (not to mention figure out the tons of features I've left undocumented!)

## Demo Code

In order to show you code that uses features throughout the rest of the text, I've added the section called *Sample Code* to the text. It has two parts: the first part, the section called *Code Fragments*, which is linked to from multiple places in the main body of the HOWTO, is a set of code fragments that demonstrate functions relevant to the sections they are linked from. The second part, the section called *Demo.c* is a single program, demo.c, which can be compiled and run. Basically, it allows you to actually see the various code fragments in action. I know this isn't the prettiest way to organize this, but it works, which is a start.

## Motor control.

The motors on the RCX are controlled through functions in include/dmotor.h (http://legOS.sourceforge.net/API/dmotor.h.html). For more details, try: kernel/dmotor.c (http://legOS.sourceforge.net/API/dmotor.c.html)

- motor_X_dir(enum MotorDirection) (http://legOS.sourceforge.net/API/dmotor.h.html#a6): This function controls the direction of motor travel. X is the letter of the output to which the motor is connected(a, b, or c, lowercase only). MotorDirection is an enum- accepted values are: fwd, rev, off, and brake. Brake shorts the motor, which physically prevents the motor from spinning, whereas off allows it to spin freely. Be aware. Shorting the motor using brake bleeds current and can drain your battery quickly. If you need to use brake, use it only briefly and then make sure you switch to off after an appropriate time period.
- motor_X_speed(int speed) (http://legOS.sourceforge.net/API/dmotor.h.html#a9): This function controls the speed of the motor. As in motor_X_dir(), X is the letter of the motor (a, b, or c). Speed is an integer value between MIN_SPEED and MAX_SPEED, where each of those are defined in include/dmotor.h as 0 and 255, respectively. Once that is applied, the motor's RPMs should be reasonably linear.
- There is unfortunately no mechanism of yet to control arbitrary motors. However, it shouldn't be all that hard to write a small function to do this for you.

I have found it very useful in my programs to define a MIN_ENV_SPEED, in which I store the speed at which the motor gains traction on whatever surface I am working on. Since this is so dependent on flooring and the wheels in use, I often have to change it, which is why I define it at the beginning. I use a simple program to test the speed at which the bot gets traction, and then program the value I obtain on a particular flooring surface into my other programs. Alternately, with a rotation sensor, it should be possible to measure exactly when the robot starts moving and use that value dynamically in a program.

The code in the section called *Motor Demo Code* is pretty self-explanatory. It will slowly increment the motor speed, and then decrement one wheel, so that the robot spins in place. It will then put the engines into "brake" and "stop" so that you can attempt to move them with your fingers, and note the difference between the two.

If you've compiled the code in the section called *Demo.c*, one press of the RUN button should run the motor demo. If the motor behavior isn't as described in the previous paragraph, you may want to check two things. First, motor direction depends on the placement of the motor leads on the RCX. This means that you can reverse motor direction by removing the leads and rotating them. So, if your motors are running "backwards," or in opposite directions, you might try switching the wiring around. Second, if one motor is not functioning properly, you may want to check which outputs the motors are connected to. Remember, demo.c expects motors attached to output ports A and C. Attaching motors and sensors to the wrong ports is a common mistake, which can be hard to notice if you have buried the leads under a substantial number of Lego bricks.

## Light Sensors

The light sensor on the RCX is controlled by kernel/dsensor.c (http://legOS.sourceforge.net/API/dsensor.c.html), andinclude/dsensor.h (http://legOS.sourceforge.net/API/dsensor.h.html).

The light sensors are used to read the difference between light and dark areas. For our purposes, this can be used to detect dark objects against a light background, or follow a dark line on a bright floor. There is sufficient detail in the reading to allow your robot to choose between different colors, but be aware that you may need to strictly control light conditions and experiment with placement of the sensor to get consistent and usable values.

Remember: if you are having inexplicable problems with the light sensors (i.e., as far as you can tell not in your code) then remember to check and make sure that they are plugged in where your code expects them to be.

### Measuring Light Intensity

The constant LIGHT_X (where X=1, 2, or 3, appropriate to the number of the input on the RCX) is set by the OS roughly every 1/4 ms. The scale of the value depends on the "mode," which I describe in the next section.

## Setting the Light Sensor Modes

The light sensors have two modes, active and passive. To change modes in the code, use these function calls:

- To go into passive mode: *ds_passive(&SENSOR_X)*, where X is the sensor number, 1, 2, or 3.
- To go into active mode: *ds_active(&SENSOR_X)* as above.

What are these modes? Well, as you can see by looking at the front of the light-sensor brick, there are two components in the brick. The first is the actual light detector, and the second is a small light source. The idea is that the light is turned on if you want to find something reasonably close which will have a big difference in reflectivity from the surrounding. This will amplify the difference between the light and dark (much like shining a flashlight on something.) If you want to judge the environment (say, finding a white spot on the wall a distance away) the built in light may interfere, so we turn it off. When the light is on, the sensor is in "active" mode, and when the light is off, it is in "passive" mode. An easy way to check which mode you are in (besides looking at the code) is to see if a small red spot appears in front of the light-sensor.

In active mode, the light sensor currently returns values between roughly 50 and 300. However, this may be fixed soon, so that light scales more linearly between 0 and 100. In passive mode, the sensor itself is unpowered, and so different values (between 220 and 280, roughly speaking) will occur. They are also not as stable as when the sensor is in active mode. It is quite possible that your range will be different than this for any number of reasons: as I write, my robot seems convinced that the world never gets darker than about 60 and goes up to 300. This is probably because of my batteries running low, and is not the normal situation.

In active mode, when the sensor is particularly close to an object, LIGHT_X becomes less a measure of darkness/brightness than it does of reflectivity. Even my jeans record a value of roughly 150 when the sensor is pressed against them. Given sufficient separation (roughly 4-5 inches in my brief experimentation) the reflected light should give a reasonably consistent value.

Be aware that in either mode, the response of the light sensor is not strictly linear (i.e. the graph of actual light intensity vs. the value read from the sensor.) Michael Gasperi has a graph of light response and discussion of the light sensor internals here. (http://www.plazaearth.com/usr/gasperi/light.htm)

The light sensor demo fragment is in the section called *Light Sensor Demo Code*. Basically, the code should make the LCD read either "dark" or "light," depending on where it is pointed. Despite the fact that I implemented the example in this fairly simplistic manner, please note that it can (with experimentation and appropriate values placed in the code) be fairly nuanced, reading far more values than the simple binary that I have demonstrated here.

In the section called *Demo.c*, two presses of RUN will run the light sensor demo. Just like in the motor example, if the light is not attached to the correct port (input port 2 in this case) the behavior will be unusual.

## Touch Sensors

Like the light-sensors, control functions for the touch sensors are in kernel/dsensor.c (http://legOS.sourceforge.net/API/dsensor.c.html), and include/dsensor.h (http://legOS.sourceforge.net/API/dsensor.h.html).

To access the touch sensors, legOS 0.2.x has TOUCH_X, which should return either a 1 (pressed) or a 0 (not pressed). Just use it as a variable, and it'll contain the proper value.

Be aware that the sensors don't spring back very well after having been touched. This allows them to be more sensitive, but also means that they tend to get stuck and remain pressed after the initial pressure is removed. Therefore, if you attach a Lego mechanism to the sensor, make sure that it is weighted such that it will spring back on its own if your design requires the sensor to be pressed and then released.

The touch sensor demo code can be found in the section called *Touch Sensor Demo Code*. This demo should run the motors forward until the touch sensor is pressed. At that event, it will back up the motors briefly and then go forward again. A slightly more sophisticated version of this (for example, if the robot turned while backing up) could easily be used to implement a robot that runs around until hitting an object, and then backs up and avoids the object in the next pass.

In the section called *Demo.c*, three presses of RUN will run the touch sensor demo. It assumes that the touch sensor is connected to input port 1.

## Angle or Rotation Sensors

LegOS includes the capability for the measurement of rotations using the Lego Group's angle sensors. These sensors are designed to indicate when an axle passing through the sensor rotates 1/16 of a rotation. In order to use the rotation sensors in LegOS, three functions must be used:

- ds_active() must be called before you expect to use the sensors, just as with a light sensor.
- ds_rotation_set( number, int position) (http://legOS.sourceforge.net/API/dsensor.c.html#a9) allows you to set the value of the rotation sensor to an arbitrary position, which the OS will then increment or decrement as appropriate. If not called, the position defaults to zero. In both this and the next functions, sensor number is of the form SENSOR_X, where X is 1, 2, or 3.

- *ds_rotation_on(&sensor number)* turns on the rotation sensor processing in the OS, which is necessary before the program starts running.

To access the rotation sensors, use ROTATION_X, where X is 1, 2, or 3. The value returned should change every 1/16 of a turn. This value should start at the position set by ds_rotation_set, and then increment when turned in one direction and decrement when turned in the other.

There is currently no demo code to show the use of the rotation sensors. Perhaps in my next life :)

## The LCD

The LCD on the front of the RCX is controlled by functions in include/dlcd.h (http://legOS.sourceforge.net/API/dlcd.h.html), include/rom/lcd.h (http://legOS.sourceforge.net/API/lcd.h.html),kernel/lcd.c (http://legOS.sourceforge.net/API/lcd.c.html) include/conio.h (http://legOS.sourceforge.net/API/conio.h.html), and kernel/conio.c (http://legOS.sourceforge.net/API/conio.c.html). It is surprisingly versatile, allowing the display either of individual LCD sections or pre-defined characters.

Writing to the LCD is pretty straightforward: just call one of the functions listed below. In legOS 0.1.x, a call to refresh() was required, but that in legOS 0.2.x this is done automatically by the OS every 100 ms.

- lcd_int(x) (http://legOS.sourceforge.net/API/lcd.h.html) writes the int x to the buffer.
- lcd_show(enum lcd_segment) (http://legOS.sourceforge.net/API/lcd.h.html#a6) and lcd_hide(enum lcd_segment) show and hide, respectively, various lcd_segments to the screen. These include the man on the screen and the arrows next to each input/output. Read the API documentation for lcd_segment (http://legOS.sourceforge.net/API/lcd.h.html) for the full list.
- lcd_clear() clears the entire screen, which can be handy, since a character which has not been overwritten will stay on the screen. cls() does the same thing, except only for the characters and not for the various "process visualization" characters like the man.
- cputs(char *s) (http://legOS.sourceforge.net/API/conio.h.html#a12) displays a five character string. If the string is less than five characters long, only the given characters will be printed, and any characters already on screen will not be cleared.

There are examples of LCD usage in each of the code fragments in the section called *Sample Code*. Note that basically all of them have sleep() or msleep() calls after them, in order to prevent overwriting. Some of these aren't obvious, though- they may come after an if/else statement, for example.

## Buttons.

In 0.2.x, buttons are accessed through dbutton.h and dkey.h. Unfortunately, unlike legOS 0.1.x, button access is no longer fully arbitrary. On/Off and Program are fixed so that On/Off always turns the robot off and Program always stops the running program. This is nice in some ways and not so nice in others. Because of that linkage, only the view and run buttons are now accessible all the time.

PRESSED(button_state(), NAME) or RELEASED(button_state(),NAME) allow access to the remaining buttons. Each function returns true if the button that has been named is in the appropriate state (pressed or released) and false otherwise. NAME must be one of the following: BUTTON_RUN or BUTTON_VIEW. This function is not debounced: i.e., if you don't put msleep() statements into your code, and have several PRESSED() statements in a row, they may all be triggered by a single press! Under certain circumstances (usually while loops that repeatedly look for button input) this can actually cause the program to freeze. Alternately, you can just tell your loop to wait until the button is released before accepting any other input or attempting to run any other code. task_swapper() in the section called *Button Demo Code* makes use of this method.

There is also another way to get get button presses. The function getchar(), when called, will wait until a button is pressed, and then immediately return a value signaling the first button that has been pressed. If VIEW is pressed, it will return 4, and if RUN is pressed, it will return 8. To use this, #include dkey.h.

## The IR unit

As of 0.2.x, legOS now uses the LegOS Network Protocol (LNP) to communicate with the PC. Under Linux, the LNP Daemon can be used to communicate with the RCX from the PC, and under Windows, WinLNP allows for the use of the various Windows programming languages to communicate with a legOS robot. Unfortunately, neither of these are completely well documented as of yet. However, if you want to find the most up-to-date information, you should be able to check out the sourceforge pages for each tool: http://legOS.sourceforge.net/files/linux/LNPD/ and http://legOS.sourceforge.net/files/windows/winLNP/ Neither of these are the "official" pages, but they should be kept up to date and point you to the right place.

## Sound

LegOS now has very thorough support for sound. For details, check out include/dsound.h. There is also an example in the demo directory, demo/robots.c, which uses this driver.

The basic structure of the driver is pretty straightforward. Music is defined as a series of note_t structs. Each note_t contains a pitch and a duration. Once you've defined an array of note_t's that you want to play, pass it to dsound_play() and the robot will play the music. dsound.h contains a list of #defines that are useful as definitions of pitches and durations, as well as a few functions (beyond dsound_play()) which may be helpful but are not necessary for the playing of sounds.

# LegOS Network Protocol

## About this section

Mike Ash (mail@mikeash.com) (mailto:mail@mikeash.com) wrote and maintains this section. Mad, mad props to him for that :)

## What is LNP?

LNP stands for LegOS Network Protocol. It allows for communication between legOS-powered robots and host computers.

Although LNP can be used for communication between many devices, with any mixture of computers and RCXs, my experience only covers communication between my computer and a single RCX, and only that situation will be discussed. LNP is still very useful in this case, since it takes care of a lot of details and is generally easier to work with than raw IR communication.

## Getting Started

LNP is included in the legOS kernel, but to use it in a program on the PC, you need to download the LNP package here. (http://legos.sourceforge.net/files/) Currently it appears the page for the Windows version is down. I run Linux, so these instructions may have OS-specific issues I am not aware of, but once LNP is set up, the code to use it will be the same. Compile the package according to the directions in the README. This will create lnpd (the LNP daemon), liblnp (the library that your programs use on the PC), and example programs for both the RCX and the PC. It will also create a version of dll that uses lnpd, which is extremely useful (I'll get to that in a moment). Note that the documents in the package talk about a patch to legOS that's necessary when using LNP from multiple threads; however, that patch has since been rolled into legOS itself, and it's not necessary to apply it.

In order to use any LNP programs on the PC side, the lnpd program must be running. It does not have to run as root, but there are performance problems with a particular serial chipset, and running as root lets lnpd work around those problems. However, beware! I apparently do not have this chipset, and running lnpd as root crashes my entire machine when it tries to reconfigure the serial port. I have no problems running lnpd as a normal user.

The command-line options for running lnpd are explained in the README, but there are a couple that are particularly useful. One is the **--nolock** option. Normally, lnpd tries to lock the serial port on startup, but permissions to do that are usually denied for users. Lnpd will exit if it fails to create a lock. The **--nolock** option will keep lnpd from even trying, and so it will run successfully. Second is the **--log[=filename]** option. This option makes lnpd log messages to the specified file, or to the system log if no file is specified. The log file can be extremely useful for tracking down problems. If your IR tower isn't on the port lnpd expects, you may specify another port with **--tty=<device>**. So, putting all this together, when I start up lnpd my command line looks like this:

```
./lnpd --nolock --log=foo (My tower is on the default port.)
```

The daemon starts up, and the command prompt returns. Assuming all goes well, the green LED on your IR tower should be lit (it will go off in a few seconds), and you should be able to use LNP programs. However, any other program that uses the serial port (firmdl3, normal dll) will not work. This is why the version of dll that comes with the LNP package is so valuable, as it allows you to dowload new versions of your program without killing lnpd every time.

# Programming

## Intro

Programming for LNP is fairly simple. Anyone with knowledge in network programming will probably pick it up easily, but it shouldn't be necessary at all. For the most part, code on the RCX is identical to the equivalent code on the PC. Any places where they are different will be noted.

LNP has two messaging layers, the integrity layer and the logical layer. The integrity layer makes sure that packets get through uncorrupted, but they aren't directed anywhere in particular. The logical layer adds addressing on top of the integrity layer, so that packets can be directed to a specific port on a specific device. The integrity layer is marginally easier to use than the logical layer, but the minor additional detail is worth the ability to address specific ports and devices, so using the integrity layer will not be covered.

The logical layer provides similar functionality to the internet's UDP. Packets are guaranteed to arrive free of corruption, *if* they arrive at all. No guarantees are made as to whether your packets actually get delivered, and if they drop off into the bit bucket your program may not be notified, nor will another transmission be attempted. The rate of packet loss can be very low if the environment is clear and no network collisions happen, but it's still important to keep the possibility of lost packets in mind.

### Headers

As always, you have to include headers in order to use LNP. On the RCX, the #include line is **#include <lnp.h>** and on the PC it is **#include "liblnp.h"**.

### Initialization

On the PC, LNP must be initialized before use. LNP is always running on the RCX, but it is still a good idea to do a little setup before using it.

On the PC, the function to call is **lnp_init(tcp_hostname, tcp_port, lnp_address, lnp_mask, flags)**. For all parameters, 0 means default. PC programs connect to lnpd over TCP, so **tcp_hostname** and **tcp_port** are the hosntame and port of the lnpd daemon to connect to. (Yes, you can actually have an LNP-using program running on one computer and have the daemon and IR tower on another.) **lnp_address** is the network address for your program on the LNP network. **lnp_mask** determines which bits of the eight-bit LNP address determine the network device, and which bits determine the port number. **flags** can change the behavior of lnpd in certain situations. There is rarely a reason to change the defaults for this call. **lnp_init**() will return 0 if init was successful, non-zero if it was not successful.

```
if(lnp_init(0,0,0,0,0))
{
  perror("lnp_init");
  exit(1);
}
else
  printf(stderr,"init OK\n");
```

No initialization is necessary on the RCX, but it can be a good idea to set the IR transmitter's range at the beginning of your program. The function call is **lnp_logical_range(range)**. Pass it 0 for near range, and 1 for far range. The advantage of far is that it works over much greater distances. However it uses much more battery power. This function call is like setting the switch on the front of the IR tower. Set the range to 0 unless the RCX needs to communicate from great distances.

There is no requirement for the computer and the RCX to have the same range. If you want the computer to command the RCX but you don't need the RCX to talk back, the RCX can be set to near while the switch on your IR tower is set to far. Of course, the computer won't be able to tell if it loses contact with the RCX this way, but maybe that's not important.

It's good to have a set of #defines with your ports, the remote host, and its ports. In my RCX program, these look like this:

```
#define MY_PORT 2
#define DEST_HOST 0x8
```

```
#define DEST_PORT 0x7
#define DEST_PORT_2 0x8
#define DEST_ADDR ( DEST_HOST << 4 | DEST_PORT )
#define DEST_ADDR_2 ( DEST_HOST << 4 | DEST_PORT_2 )
```

In other words, my program's port is port 2. The remote host is at address 8, and I'll be sending messages to ports 7 and 8 on the remote host. With the default mask, the upper four bits of the address determine the host, and the lower four bits determine the port on that host. When transmitting data, the address must be passed to LNP with the host and port already stuffed into one byte, which is why I have those last two #defines.

## Packet Handlers

In order to recieve data with your program, you must set up packet handlers. A handler will be called whenever a packet arrives on that handler's port. You must set up a handler for each port you expect to recieve data on. Addressing handlers are defined as follows:

```
void addr_handler_1(const unsigned char *data, unsigned char length, un-
signed char src);
```

**data** is a pointer to the data in the packet. Length is how much data is in the packet. Src is the address of the program that sent the packet. Notice that since length is a char, it cannot hold more than 255 .With networking overhead, the limitation is actually 253 bytes per packet. Now the handler must be registered with LNP.

```
lnp_addressing_set_handler (MY_PORT_1, addr_handler_1);
```

This will install the **addr_handler_1**() function on port MY_PORT_1. Now, if any data is recieved on that port, your function will be called. Note that if your code is running under legOS (as opposed to on your PC), this function is called from the legOS interrupt routines, so avoid things like memory functions or thread control functions. Things like memcpy, or anything that doesn't cause memory allocation or task switching, are ok. It's also a bad idea to take too much time inside a handler. A good basic strategy would be to copy the data into a buffer and setting a flag, with a thread standing by to process the data. But that will be covered later.

### Sending Data

Sending data is pretty simple as well. You need to have some data to send, of course. You need to know how big the data is. Finally, you need an address to send it to.

```
result = lnp_addressing_write(data, length, DEST_ADDR, MY_PORT);
```

Where **data** is a pointer to what you actually want to send, **length** is how many bytes to send, **DEST_ADDR** is the address you're sending to, and **MY_PORT** is the port you're sending from. This function returns 0 if successful, and usually nonzero if it failed. "Usually" because you are not guaranteed to know if the transmit failed. Never rely on your packets arriving 100% of the time. The function call will block, not returning until the transmission is complete.

## What do I do with it? (and how?)

By now you probably have some great ideas as to what to do with LNP. In case you don't, or you want some more, or you just want some examples about getting things done with LNP, read on! That's what this section will cover.

### Debugging

One of the things that can be frustrating with legOS is the difficulty in debugging programs. Debugging using the LCD and sound is just not very easy. However, debugging can be made simple using LNP and a basic program on the host PC. The idea is to use the old method of "debug by printf". That is, print status messages and variable values to figure out what's going on in the program.

I want the ability to send strings and numbers. Since I can send multiple types of data, I make it so that the first byte of any packet is a code that identifies the type of data, and the remaining bytes will be the data itself. Strings will include the terminating \0 for convenience, though it's not strictly necessary (the length parameter in the handler could be used instead). This method will also allow the easy addition of other data types. Note that I am not taking a printf route, with a format string and optional arguments. Not only would it be harder to write, but I feel the overhead would not be justified. You, of course, are free to work differently.

```
void printString(char *s)
{
  int len, result;
  char *buffer;

  len = strlen(s);
```

```
  buffer = malloc(len + 2);
  buffer[0] = 's';
  memcpy(buffer + 1, s, len + 1);
  result = lnp_addressing_write(buffer,len + 2, DEST_ADDR,  MY_PORT);
  free(buffer);
}
```

This function takes a string as input, and sends it out with a type identifier of 's' (for string) in front of it. First we need to know how long the string is, then a temporary buffer is allocated. All of the data to be sent has to be in one buffer, so some scratch space is needed to build the packet in. That scratch space is allocated, the identifier is placed into the first byte, and the string (including terminating null) is copied in using memcpy. Finally, the whole thing is sent out with **lnp_addressing_write()**, and the buffer is deallocated.

Note that an array for a buffer would be faster than allocating and deallocating a buffer for each function call. There are disadvantages, however. If the array is a normal stack variable, then it will take up a large amount of stack space, which could cause a problem with overflows. If the array is declared static, then you must ensure that only one thread at a time calls the function, either by design or with semaphores.

```
void printInt(int i)
{
  char buf[3];
  int result;

  num = (char *)(&i);
  buf[0] = 'i';
  memcpy(buf + 1, &i, 2)
  result = lnp_addressing_write(buf,3, DEST_ADDR, MY_PORT);
}
```

This function takes a single int and sends it over the network. Here the buffer is just a local array, since it only needs to be three bytes long. The first byte of the buffer is set to the type identifier ('i', for int), and the following two are set to the two bytes composing the int. Finally, again, the buffer is sent over the network with **lnp_addressing_write**.

Now we have the LNP handler on the PC side:

```
void addr_handler(const unsigned char* data,unsigned char length, un-
signed char src)
{
  short temp;
  char *ptr;

  switch(data[0])
```

```
  {
  case ’s’:
    puts(data + 1);
    break;
  case ’i’:
    ptr = (char *)∧temp;
    #ifdef BIG_ENDIAN
      ptr[0] = data[1];
      ptr[1] = data[2];
    #else
      ptr[0] = data[2];
      ptr[1] = data[1];
    #endif
    printf("%d", temp);
    break;
  }
}
```

The basic idea here is to examine the type identifier in the packet and then either print the string that follows or extract the int and print it. Some trouble comes when considering so-called endian issues. (If you know what endian is and how to deal with it, skip to the paragraph after next.) The short version is, there are two main ways to represent numbers that take up multiple bytes in memory. "Big-endian" processors store the most-significant byte at the lowest address, and the least-significant byte at the highest address. If the number 0x1234 is stored at location 100, then the byte at 100 contains 0x12 and the byte at 101 contains 0x34. Little-endian processors store things in the opposite direction, so that byte 100 would contain 0x34 and byte 101 contains 0x12.

The good news is, you almost never have to deal with this. As long as your processor is consistent (and if it wasn’t, it wouldn’t work) and you don’t do anything evil like examining the individual bytes of a number, you’ll never notice it. However, there is a problem when it comes to transferring data from one computer to another. If the two computers have different byte orders, any numbers transferred will appear scrambled. To fix this, the bytes must be swapped, and that’s why the indices into **data** are backwards in the **#else** clause in the function. The RCX’s processor is big-endian, so if the host PC is also big-endian then the two processors can use each others data without modification. Of course, the **BIG_ENDIAN** identifier is fictional: you have to figure out what your computer is for yourself. It’s not hard. If you’re running an Intel-based PC, your processor is little-endian. If you’re running a PowerPC, it’s big-endian. If you’re running something else, chances are it’s big-endian, but you can also use the program below to check:

```
int main(void)
{
  int x = 1;
  char *foo = (char *)&x;
```

```
  if(foo[0] == 1)
    printf("little-endian\n");
  else
    printf("big-endian\n");
  return 0;
}
```

Now you have a much better debugging system! These function calls can be called pretty much anywhere, anytime, with the exception of LNP data handlers. There are a couple of things to watch out for, though. One problem is that these functions take some time to run, since they don't return until the full packet has been transmitted. Keep that in mind when debugging time-critical routines. Also, these strings are actually stored on the RCX, and sent over the network every time they are printed. This takes up limited memory and time. In my project, turning off debugging strings reduces my program size by over 1k, a significant portion of the program. If memory is a problem, it may be worth the trouble to send string ID numbers from the RCX which the PC would then look up in a table to get the actual string.

Once you understand the principles above, it should be pretty easy to move past simple debugging and into real data exchange. Keep in mind endian issues when transmitting any numbers bigger than one byte if your machine is little-endian.

## Remote Control

Sending data from the RCX to the PC is pretty simple, since anything goes in an LNP packet handler on the PC. On the RCX side, however, an LNP packet handler is limited in what it can do. Things like memory allocation and thread controls are off-limits. Heavy data processing is highly discouraged. In general, it's best to simply copy the data into a buffer, set a flag, and let a thread take care of the processing. An example setup of this follows:

```
int gNewData = 0;
int gMessagingData[256];
int gDataLength;

void packet_handler(const unsigned char* data,unsigned char length, un-
signed char src)
{
  if(gNewData == 0)
  {
    memcpy(gMessagingData, data, length);
    gDataLength = length;
    gNewData = 1;
  }
}
```

```
int PacketWatcher(int argc, char **argv)
{
  wakeup_t WaitForData(wakeup_t data)
  {
    return gNewData;
  }

  while(1)
  {
    wait_event(WaitForData, 0);
    switch(gMessagingData[0])
    {
      ...
    }
    gNewData = 0;
  }
}
```

All the packet handler does is copy the data to **gMessagingData** and then set **gNewData**, but only if **gNewData** is clear. That flag is the signal for the packet watcher thread to look at the new data. If any packets comes in before the packet watcher can take care of the data, those packets will be discarded. If that is a problem, it may be possible to use the pool more efficiently, or use some other scheme. Since the legOS scheduler does not support explicitly waking threads, the packet watcher has to wait for its turn to be touched before it can go into action. Again, the first byte of the packet is used as an identifier to figure out what to do with the rest of the data. Fill in the blank in the switch statement to do whatever you need. Finally the flag is reset, signalling that the packet watcher is ready to process more data.

On the PC side, send data just like in the RCX programs. Keep in mind byte-order issues if your machine is little-endian. Again, since your PC has the processing power and the memory, I recommend doing the byte swapping in the PC program, rather than on the RCX.

# Program Flow and Control in LegOS

*Note: at the moment, this is pretty rudimentary. The original (and still primary) aim of this document is to allow students with a basic knowledge of C to get up to speed in LegOS quickly. For the most part, the functions and discussion I include here are designed for that purpose.*

Probably the best way to learn code is to read other people's code. There is some in the demo/ directory in your LegOS directory. rover.c is a good place to get a feel for what is going on in simple movement and bumper input. Alternately, as already mentioned, tm-and-ir.c gives you a look at the internals of the IR system. There are also a few examples on the LegOS homepage. Demo.c, which I have already referenced, ties together some of the concepts in this section.

I've tried to do this listing by general concepts, so that people can find within them the functions they need for a particular task. If you are looking for documentation on specific snippets of code, look at the API again first, then come here. I've tried to include some general ideas for using the code, so that others can benefit from new ways of looking at the same old function. Hopefully, others will contribute to this as well.

## Basic thread and process control

To do many things in LegOS you need at least a basic grasp of threading. Why? Because robotics, unlike most basic programming tasks, is not a linear task. Robots usually have to perform a certain set of uninterrupted tasks repeatedly, while at the same time waiting for specific inputs from the outside world. This is difficult at best with similar linear programming, since you have to run a loop, check for something, and then fork, while still maintaining the original behavior in the new fork. With threads, you can maintain one behavior in one thread while using other threads to maintain or modify other behaviors.

In a nutshell (in case you are really inexperienced) threading allows multiple processes to run side-by-side on the same chip. The OS executes part of one thread, part of the next, and so on and so forth. In the section called *Demo.c*, one thread waits for button input and controls which other thread is running, while the others execute the various examples as they are called. This process is controlled internally by tm.h (http://legOS.sourceforge.net/API/tm.h.html) in LegOS. The interface for the LegOS programmer is unistd.h (http://legOS.sourceforge.net/API/unistd.h.html#a1), a reduced version of the standard C library for Unix. For starters, execi() (http://legOS.sourceforge.net/API/unistd.h.html#a1) is the most important function in unistd.h.

*execi(&function_name, int argc, char argv, int prior, DEFAULT_STACK_SIZE)* calls the function function_name and assigns it the priority int prior. execi() immediately returns a pid_t, which you don't need to store, but you may wish to if you intend to manipulate it later. execi() immediately returns, and then you can start other threads by calling execi() from within main (or within other functions), without waiting having to wait for the original function_name to run to completion.

Several things should be kept in mind when starting a thread with execi:

- The function function_name must take an int and a **char, even if they are not used. They will get passed the values of argc and argv, which are described below.
- Selecting priorities by using int prior in execi can be important. Basically, they can't be above PRIO_HIGHEST (defined as 20 by default)- otherwise, the scheduler will fail to kill them.

- While not always important, the argc and argv values can be very useful for programming, particularly if you want to start a separate thread from within another thread. Under most circumstances, threads are started when a program is first called, and in that case, you'll have no data to pass to a thread. In this case, pass 0 for argc, and NULL for argv. However, if you want to start a thread from within another thread, after the program has been started, you may want to pass data culled from the first string to the new string. You can pass the data by making argc equal the number of items you want to pass, and putting those arguments into the char argv. These values will be passed to the function.

*kill(pid_t)* allows you to pass the process id returned by execi() to kill a thread.

Unlike 0.1.x, in 0.2.x, when your program is killed by the OS, all motors, sensors, etc., are turned off by the OS, so you no longer need to worry about that in threaded programs. However, you still have to turn them off after individual threads are killed.

You can see all of this (both execi() and kill()) in action in task_swapper() in the section called *Button Demo Code*. Basically, each button press kills the previous thread (using it's pid_t), does some cleanup (making sure the motors are stopped, for example) and then uses execi to start the next thread.

## Timing and Event Control

Three basic functions are used in LegOS to control time-related issues: sleep() (http://legOS.sourceforge.net/API/unistd.h.html#a6), msleep() (http://legOS.sourceforge.net/API/unistd.h.html#a7), and wait_event() (http://legOS.sourceforge.net/API/tm.c.html#a11).

sleep(X) and msleep(X) do exactly what their names imply- they put a thread to sleep for an integral number of seconds or milliseconds X, repectively. Because these functions are limited to exact amounts of time and cannot be tied to specific events, their uses are limited. I find them most useful in debugging loops, where I output to the LCD and then pause the bot briefly so that I can read a string or value. If you've been looking at the code in the section called *Sample Code*, then you've already seen this. If not, check out the section called *Motor Demo Code*, where msleep() is used to control timing of the LCD and sleep() is used to time the behavior of the motors for extended periods (well, 3 seconds).

For more sophisticated waiting, there is wait_event(function_name, data) (http://legOS.sourceforge.net/API/tm.c.html#a11). wait_event() allows you to create a function of type wakeup_t in your code, which will return zero under normal conditions (say, the button has not been pressed) but then will return a non-zero value when whatever condition you want to wait for has occurred (say, the button has been pressed.) In order to test your function, pass it to wait event (along with a string or other necessary data). The OS will activate the function every time it touches the thread until the function returns a non-zero value, at which point the non-zero value is returned by wait_event() and can be used or ignored as necessary. Obviously, this is one of the most important functions in the OS: you will want to use it (for example) to monitor bumpers, or check for button presses, or to activate the robot when light values change suddenly. the section called *Demo.c* uses this to wait for a button press, so that it knows to change functions. You can see this in the section called *Button Demo Code*.

## Semaphores

Semaphores are sort of complex, and I'm no expert in them (yet) so I won't discuss them in much detail here. However, they are there- and POSIX 1003.b compliant (just like Linux.) So, take a look at include/semaphore.h to see the interface, or, if you want more detail and run Linux, use man sem_init to see a more full description of what the semaphores are supposed to do. Since both implementations (Linux and legOS) attempt to be POSIX compliant, the documentation for the one is good documentation for the other.

# C++ under legOS

If you've gotten this far, you've probably noticed that the examples in the previous section are all in C. Don't despair- Pat Welch has added excellent C++ support to legOS. Unfortunately, at the moment, this is only available in CVS, but it will be available more easily in version 0.2.5 (whenever that is released.) To see how to use CVS, read the section called *Getting legOS Development Versions From CVS*.

To see how to use the C++, Pat has written a large set of demo code, available in the demo/c++/ directory.

# Debugging

Unfortunately, debugging the RCX is difficult. The key to debugging is to dump information to the LCD, much like you would use a printf or cout statement when programming normally. However, there is a trick to it- remember you aren't in front of an xterm where you can scroll through a list of couts (Unfortunate, isn't it?). As I've already pointed out, there are physical limits to the refresh rate of the LCD. As a result, if you get caught in a loop, your couts will get piped to the LCD faster than the screen can deal with it and it will appear as if the screen is covered with junk. To deal with this problem, you'll need to insert delay loops in the code, effectively pausing the 'bot and the codeuntil the screen can catch up. To insert a delay in the code, use msleep(X) or sleep(X). Alternately, you can insert a wait_event() command, allowing you to look at and contemplate an outputted value until you press a button on the RCX. Depending on the complexity of the code, this might demand a great deal of repetitive button pushing.

I find that a simple function that accepts a string and an int, where the string is an identifier for the int about to be outputted, serves well. This allows you to output the identifier, pause (so you can read it!) and then output the appropriate value to be checked against what you expected. I have also found it useful to make many of my variables universal, so that I can write a function that will cycle through them based on button presses I call while a program is being run. It's bad programming form, but you can always get rid of that when you are satisfied that it works. Similarly, though I have not yet attempted to make this work, it should be possible to make a program pause and continue based on button inputs, while cycling through variables during each pause. This would, of course, require a bit of patience- you would have to check a flag after every function call that you'd wish to check variables after.

Now that LNP is better documented, it is also possible to use it to dump debugging information to your PC. For more details, read the section called *Debugging*.

## Emulators

This section is included for a variety of reasons: historical interest, nod to the work that has been done, etc... Unfortunately, neither of these work with 0.2.x. May be worth trying, though, if you have a lot of time to fix things up for 0.2.x. The code for both are very well organized, so you may want to check them out if you really, really need an emulator.

### EmulegOS

For some code problems, running your code in emulation on a PC may help with debugging. Mario Ferrari and Marco Beri have written a LegOS emulator for Windows PCs called emuLegOS. The program is available at http://www.geocities.com/~marioferrari/emulegos.html. A tk based port is also available, which should run on most Unix machines. There are three main issues to consider when looking at emulegOS:

1. Instead of LegOS's native threading, the emulator uses the pthreads library, which may hide threading problems in your code or in LegOS.
2. Certain other cross-platform issues may be obscured by the code port, for example, your platform is 32-bit, while LegOS is not, and emuLegOS (or any other emulator, for that matter) cannot catch that type of error on your part.
3. Depending on how complex and thorough you want the emulation to be, you may have to write emulation for the mechanical aspects of the robot. For the most part, inputting raw values to the sensors should be sufficient. If, however, you would like the program to mimic the full mechanical behavior of your robot (complicated switches and the like) then you may have to write some simple code to mimic those behaviors.

Clearly, none of these should be show-stoppers. I merely point them out to make users aware that certain issues may not be debuggable with an emulator.

### legOSIM

There is a program called legOSSim that I can't find the link to right now :)

# Hacking on legOS

Some people have more of an itch than the rest of us- they really, really want to add features and such to legOS. The next two sections deal with that.

## Kernel Internals

Stig Neilsson has written an *awesome* guide to the legOS kernel, as of version 0.2.4. It covers just about everything you could ever possibly want to know about the kernel, and is a great place to start (other than reading the code ;) if you want to learn about the kernel. You can find it at http://legOS.sourceforge.net/docs/kerneldoc.ps (http://legos.sourceforge.net/docs/kerneldoc.ps).

## Getting legOS Development Versions From CVS

### Instructions

A number of folks have asked how to get legOS out of CVS, so here are the basic instructions. These are each separate commands, to be executed at the command prompt.

```
export CVS_RSH=ssh
export CVSROOT=:pserver:anonymous@cvs.legOS.sourceforge.net:/cvsroot/legOS
cvs login
cvs -z3 checkout legOS
```

Because CVS works poorly with symlinks, you'll have to add one link, using the following commands (adjust as necessary for your file system):

```
cd legOS/util/dll-src/
ln -s ../../kernel/lnp.c .
```

### More Details on CVS at Sourceforge

If you want to know more about how to set up CVS with Sourceforge, you'll probably want to look at http://sfdocs.sourceforge.net/sfdocs/, which (about halfway down the page) provides a whole list of links about CVS, including basics, more detailed stuff, and CVS with Windows.

### Getting CVS Write Access

Getting CVS write access really isn't that hard. The first thing to do is have a patch to add something useful to legOS. Preferably, the patch should apply cleanly against the current CVS version of legOS, and should be well commented. Markus did an excellent job of commenting his code, and new contributors should strive to do the same. Once you've done that, write to the legOS list at lugnet.com (mailto:lugnet.robotics.rcx.legos@lugnet.com), describe your patch, and state that this is only the first in a long and woderful series of patches that will make legOS more stable and ensure world peace :) Additionally, you'll have to register as a developer at sourceforge. Once you've done that, we are normally pretty generous about giving write permission.

# Advanced Topics and Random Thoughts

These are just oddities that don't really fit anywhere else at the moment. Please feel free to contribute if necessary.

## Floating Point Emulation

Floating point emulation is now done automatically in the build process. Just use floats and doubles as you would normally, and support for them will be compiled into your .lx files. Be aware that floating point math will take about 50% longer than integer math that has the same results.

## Random Number Generation

random() and srandom(int x) are now available in legOS. To use them, just call srandom(int x) at some point during your program startup. x is a "seed", which allows you to get the same sequence of numbers if you so desire (by passing the same seed) or to generate more truly "random" numbers by feeding in, for example, LIGHT_1. Be aware: unlike the "standard" implementation of these two functions, there is no "default" seed, so if you don't call srandom() at least once, random() will return the same number over and over again.

## Math

ints and such may be slightly smaller than you are used to in "real" programming. In specific, an int is only 16 bits long, so it will overflow at around 32K if signed and around 64K if unsigned.

## Available Memory

Not that this means much until you've actually compiled code, but there is 32K available for the entire system. The standard OS configuration should consume about 18K of that, and judicious use of config.h can reduce this to 14K or so.

## *config.h* and Memory Optimization

If you need to ensure that LegOS has a particularly small memory footprint, you can edit the #defines in the boot/config.h file to prevent the compilation of certain features. The features that can be turned on and off include:

- if you aren't using threads: task management, semaphores, and memory management
- IR communication with the computer
- sound
- motors
- all sensors
- rotation sensors only

# Sample Code

This section is a listing of sample code, demonstrating most of the main features of legOS. They are probably overly complicated, but attempt to give a thorough demonstration of the relevant points. Roughly speaking, these are in the order given in the rest of the text.

# Code Fragments

These are just fragments, for the most part drawn directly from the section called *Demo.c*.

## Motor Demo Code

There are four parts to this example. The first while loop does a simple linear speed increase. The second while loop puts the bot into a spin by reversing one motor. The last two statements demonstrate the difference between brake and off. You should use your fingers to test the difference between the two if you run it.

```
/*This needs to be retabbed :(*/
void motor_driver()
{
  int rover_speed = 0;

  /*speeds up the bot in linear fashion- watch out!*/
  while(rover_speed<MAX_SPEED)
{

  motor_a_speed(rover_speed);
  motor_c_speed(rover_speed);

  lcd_int(rover_speed);

  motor_a_dir(fwd);
  motor_c_dir(fwd);

  msleep(500);  /*adjust here if you need a finer differentiation*/

  rover_speed+=20;
}

      /*slows down the bot in linear fashion*/
      while(rover_speed>=0)
{
  motor_a_speed(rover_speed);
  motor_c_speed(rover_speed);

  lcd_int(rover_speed);

  motor_a_dir(fwd);
  motor_c_dir(rev);
```

```
  msleep(500);  /*adjust here if you need a finer differentiation*/

  rover_speed-=10;
}

      /*test the brake state with your fingers*/
      cputs("brake");
      motor_a_speed(brake);
      motor_c_speed(brake);
      sleep(3);

      /*test the off state with your fingers*/
      cputs("off");
      motor_a_speed(off);
      motor_c_speed(off);
      sleep(3);
    }
}
```

## Light Sensor Demo Code

This assumes that input 2 is a light sensor. If pointed at a relatively bright area, it will display "light" on the LCD, and if pointed at a relatively dark area, it will display "dark."

```
void light_sensor()
{
  cputs("light");  /*indicate which function we are in*/
  msleep(500);

  ds_active(&SENSOR_2);    /* turn on light sensor LED (active mode)*/
  while(1)
    {
      lcd_int(LIGHT_2);
      msleep(250);

      if (LIGHT_2 < 150)
{
  cputs("dark");
}
      else
{
  cputs("light");
}
```

```
      msleep(250);
    }
}
```

## Touch Sensor Demo Code

```
/*
  the touch sensor example:
  assumes that the sensor is attached at the first input port
  when sensor is not pressed, runs motors forward
  when sensor is pressed, runs motors backward briefly
  also outputs appropriate sensor value
*/

void touch_sensor()
{
  cputs("touch");  /*indicate which function we are in*/
  msleep(500);

  motor_a_speed(100);
  motor_c_speed(100);

  while(1)
    {
      motor_a_dir(fwd);
      motor_c_dir(fwd);

      cputs(TOUCH_1); /*output sensor value*/

      if(TOUCH_1!=0)    /*if sensor has been touched*/
{
  motor_a_dir(rev);
  motor_c_dir(rev);
  sleep(1);
}
    }
}
```

### Button Demo Code

This code is a set of wakeup functions that are used elsewhere in the code. Wakeup functions are covered in detail in the section called *Program Flow and Control in LegOS*.

```
/*
   button press functions
*/

wakeup_t button_press_wakeup(wakeup_t data) {
return PRESSED(button_state(),data);
}

wakeup_t button_release_wakeup(wakeup_t data) {
return RELEASED(button_state(),data);
}
```

This code uses the last fragment for debouncing. execi() is explained in the section called *Program Flow and Control in LegOS*.

```
int  task_swapper()
{

  wait_event(&button_press_wakeup,BUTTON_RUN); /*wait for PROGRAM button*/
  wait_event(&button_release_wakeup,BUTTON_RUN);/*wait for release*/

  pid2=execi(&motor_driver, 0, NULL, 2, DEFAULT_STACK_SIZE);/*start mo-
tor example*/
  wait_event(&button_press_wakeup,BUTTON_RUN); /*wait for PROGRAM button*/
  wait_event(&button_release_wakeup,BUTTON_RUN); /*wait for release*/
  kill(t2);                                        /*kill motor example*/
  motor_a_speed(off);
  motor_c_speed(off);

  pid3=execi(&light_sensor, O, NULL, 3, DE-
FAULT_STACK_SIZE);/*start light example*/
  wait_event(&button_press_wakeup,BUTTON_RUN); /*wait for PROGRAM button*/
  wait_event(&button_release_wakeup,BUTTON_RUN);/*wait for release*/
  kill(t3);                                        /*kill light example*/
  ds_passive(&SENSOR_2);   /* turn off light sensor LED (pas-
sive mode again) */


  pid4=execi(&touch_sensor, 0, NULL, 4, DE-
FAULT_STACK_SIZE);/*start touch example*/
```

```
  wait_event(&button_press_wakeup,BUTTON_RUN); /*wait for PROGRAM button*/
  wait_event(&button_release_wakeup,BUTTON_RUN);/*wait for release*/
  kill(t4);                                        /*kill touch example*/

  lcd_clear();
  sleep(1);

  cputs("reset");
  sleep(3);

  return 0;
}
```

## Demo.c

This is (or rather, should be ;) a fully compilable program that, when run, demonstrates all the main functionality of legOS. Just cut and paste it into a file called `demo.c` and then compile and download it as described in the section called *Compiling a legOS program into a .lx file*.

```
/*
  demo.c: a simple program designed to work with examples from the Le-
gOS HOWTO
  http://legOS.sourceforge.net/HOWTO/
  author: Luis Villa
*/

#include <conio.h>
#include <unistd.h>
#include <dbutton.h>
#include <dsensor.h>
#include <dmotor.h>
#include <sys/tm.h>
#include <rom/system.h>


pid_t pid1, pid2, pid3, pid4;

/*
   button press functions
*/
```

```
wakeup_t button_press_wakeup(wakeup_t data) {
return PRESSED(button_state(),data);
}

wakeup_t button_release_wakeup(wakeup_t data) {
return RELEASED(button_state(),data);
}

/*
  the touch sensor example:
  assumes that the sensor is attached at the first input port
  when sensor is not pressed, runs motors forward
  when sensor is pressed, runs motors backward briefly
  also outputs appropriate sensor value
*/

void touch_sensor()
{
  cputs("touch");  /*indicate which function we are in*/
  msleep(500);

  motor_a_speed(100);
  motor_c_speed(100);

  while(1)
    {
      motor_a_dir(fwd);
      motor_c_dir(fwd);

      cputs(TOUCH_1); /*output sensor value*/

      if(TOUCH_1!=0)    /*if sensor has been touched*/
{
  motor_a_dir(rev);
  motor_c_dir(rev);
  sleep(1);
}
    }
}


/*
  the light sensor example:
  assumes that the sensor is attached at the second input port
  if pointed at a relatively bright area, will display "light"
```

```
  if pointed at a relatively dark area, will display "dark"
  Based on a suggestion from Richard Franks
*/

void light_sensor()
{
  cputs("light");  /*indicate which function we are in*/
  msleep(500);

  ds_active(&SENSOR_2);    /* turn on light sensor LED (active mode)*/
  while(1)
    {
      lcd_int(LIGHT_2);
      msleep(250);

      if (LIGHT_2 < 150)
{
  cputs("dark");
}
      else
{
  cputs("light");
}

      msleep(250);
    }
}

/*
   the motor example:
   the first while loop does a simple linear speed increase
   the second while loop puts the bot into a spin
   the final two statements demonstrate brake and off-
       use your fingers to test the difference between the two.
   the whole thing loops until PROG is pushed (see task_swapper() for con-
trol of this)
*/

void motor_driver()
{
  int rover_speed;
  while(1)
    {
      rover_speed=0; /*realign here, otherwise you start at a nega-
tive value*/
```

```
      /*speeds up the bot in linear fashion- watch out!*/
      while(rover_speed<MAX_SPEED)
{

  motor_a_speed(rover_speed);
  motor_c_speed(rover_speed);

  lcd_int(rover_speed);

  motor_a_dir(fwd);
  motor_c_dir(fwd);

  msleep(500);  /*adjust here if you need a finer differentiation*/

  rover_speed+=20;
}

      /*slows down the bot in linear fashion*/
      while(rover_speed>=0)
{
  motor_a_speed(rover_speed);
  motor_c_speed(rover_speed);

  lcd_int(rover_speed);

  motor_a_dir(fwd);
  motor_c_dir(rev);

  msleep(500);  /*adjust here if you need a finer differentiation*/

  rover_speed-=10;
}

      /*test the brake state with your fingers*/
      cputs("brake");
      motor_a_speed(brake);
      motor_c_speed(brake);
      sleep(3);

      /*test the off state with your fingers*/
      cputs("off");

      motor_a_speed(off);
      motor_c_speed(off);
```

```
      sleep(3);
    }
}

int  task_swapper()
{

  wait_event(&button_press_wakeup,BUTTON_RUN); /*wait for PROGRAM button*/
  wait_event(&button_release_wakeup,BUTTON_RUN);/*wait for release*/

  pid2=execi(&motor_driver, 0, NULL, 2, DEFAULT_STACK_SIZE);/*start mo-
tor example*/
  wait_event(&button_press_wakeup,BUTTON_RUN); /*wait for PROGRAM button*/
  wait_event(&button_release_wakeup,BUTTON_RUN); /*wait for release*/
  kill(t2);                                       /*kill motor example*/
  motor_a_speed(off);
  motor_c_speed(off);

  pid3=execi(&light_sensor, O, NULL, 3, DE-
FAULT_STACK_SIZE);/*start light example*/
  wait_event(&button_press_wakeup,BUTTON_RUN); /*wait for PROGRAM button*/
  wait_event(&button_release_wakeup,BUTTON_RUN);/*wait for release*/
  kill(t3);                                       /*kill light example*/
  ds_passive(&SENSOR_2);   /* turn off light sensor LED (pas-
sive mode again) */


  pid4=execi(&touch_sensor, 0, NULL, 4, DE-
FAULT_STACK_SIZE);/*start touch example*/
  wait_event(&button_press_wakeup,BUTTON_RUN); /*wait for PROGRAM button*/
  wait_event(&button_release_wakeup,BUTTON_RUN);/*wait for release*/
  kill(t4);                                       /*kill touch example*/

  lcd_clear();
  sleep(1);

  cputs("reset");
  sleep(3);

  return 0;
}

int main()
{
  pid1=execi(&task_swapper, 0, NULL, 1, DEFAULT_STACK_SIZE);
```

```
  return 0;
}
```

# Links

## Official and Organization-level pages

- The official Lego Mindstorms Site. (http://www.legomindstorms.com/)
- The LegOS homepage. (http://www.noga.de/legOS/)
- The *other*legOS homepage (http://legOS.sourceforge.net). This "official" page is used mainly for CVS and file download, since sourceforge provides 100M of free storage and unlimited downloads. It will (eventually) be the "official" site for all things legOS, including development and documentation.
- LUGNET (http://www.lugnet.com)- the Internet's home of all things Lego. See particularly the robotics mailing list (http://www.lugnet.com/news/display.cgi?lugnet.robotics) and the LegOS mailing list (http://www.lugnet.com/news/display.cgi?lugnet.robotics.rcx.legos).
- Lego Mindstorms Internals (http://www.crynwr.com/lego-robotics/.): This page has tons of links, focusing mainly on OS-type programs and communications protocols.
- MIT Constructopedia (http://el.www.media.mit.edu/groups/el/projects/constructopedia/): A work in progress at the MIT Media Lab.
- O'Reilly Link Page (http://www.oreilly.com/catalog/lmstorms/resources/index.html): Jonathan Knudsen, writing for O'Reilly Publishing, has created a Mindstorms book. This page has all the links from the book.

## LegOS related programs

- EmuLegOS (http://www.geocities.com/~marioferrari/emulegos.html): A LegOS emulator which should be very useful for debugging LegOS code.

- OnScreen (http://alumni.cse.ucsc.edu/~davevw/onscreen/index.html): David Van Wagner's LegOS program, which allows for simple programming on the fly.
- PyLNP (http://www.hare.demon.co.uk/lego/pylnp.html) is a Python library that allows Python programs running on the PC to control an RCX via LNP.

## Personal pages for Mindstorms- legOS and non-legOS

- Mario Ferrari's Lego page (http://www.geocities.com/CapeCanaveral/Galaxy/9449/lego_mindstorm.html). Some of these are legOS powered, and there are lots of neats things legOS and non-legOS.
- Eric Brok's Lego on my Mind (http://homepages.svc.fcj.hvu.nl/brok/legomind/)
- Bert van Dam's page (http://www.bvandam.net): heavy AI stuff, including a neural net and lots of links.
- Simen Skrogsrud's page (http://www.mop.no/~simen/lego.htm): He built a scanner out of Lego. What more do you want?
- Alex's page (http://www.phred.org/~alex/lego/): some good designs based on the "differtrans," a transmission which uses the differential unit included with the Mindstorms kit. A good place to start if you are very concerned about traveling in a straight line.
- Machina Speculatrix (http://www.plazaearth.com/usr/gasperi/walter.htm): A page discussing a very early "learning" robot, with an attempt to replicate it in the RCX. Very interesting. Part of the Gasperi web complex :)
- Clint Rutkas' page (http://members.xoom.com/_XOOM/rutkas/index2.html): A page with some good designs, including a large, functional claw/arm combination. Worth checking out.
- Rich Thompson's page (http://www.marsrobot.com/lego.htm): you know you are serious when you get your own URL :)
- Joe's Mindstorm's Gallery (http://member.nifty.ne.jp/mindstorms/): unfortunately, this page is in Japanese, but if you look, you'll find some gems- including a Lego-based Turing Machine.
- An aluminum can collector. (http://www.geocities.com/EnchantedForest/Glade/1380/mind.htm)
- The Mill Page (http://www.geocities.com/Heartland/Bluffs/7900/): Picture a Lego mill cutting designs into burnt toast. An interesting experiment in Lego control.
- Dave's Page (http://www.astolfo.com/lego/home.asp): if nothing else, check out the Johnny 5 replica.

- Jin Sato's page (http://www.mi-ra-i.com/JinSato/MindStorms/index-e.html): Another Japanese page. Excellent pictures of an arm and a robot which rotates in place.

## Other Related Pages

- Fred Martin's MIT page (http://fredm.www.media.mit.edu/people/fredm/projects/6270/): check out in particular the Robot Builder's Guide. The same author (who was a co-founder of the MIT class) has also written "The Art of Lego Design," available in PDF (ftp://cherupakha.media.mit.edu/pub/people/fredm/artoflego.pdf) and compressed PostScript (ftp://cherupakha.media.mit.edu/pub/people/fredm/artoflego.ps.Z). This is an expanded and better illustrated version of chapter 7 of the Robot Builder's Guide, which I mentioned in Section 2.4 of this document.
- RCX Arena Combat (http://www.azimuthmedia.com/RobotArena/mainframe.html): some ideas for interactive Lego competitions.
- A Robotic Vacuum Cleaner (http://www.eureka.com/whatsnew/robotvac.htm): Should be very easy to do with LegOS, but Eureka wants to charge $1,000 dollars a piece for it.

## Other Programming Environments

This list has now evolved into the Lego + Linux mini-HOWTO (http://linuxdoc.org/HOWTO/mini/Lego-HOWTO/), available separately from the Linux Documentation Project (http://linuxdoc.org/). I urge Linux users to look there first, since it is (most likely) more up to date and definitely better organized and more informative. Windows users may also want to take a look, since most of the projects described there have Windows ports.

- NQC (http://www.enteract.com/~dbaum/nqc/index.html): this is a C-style language that uses the standard Lego firmware. This makes it easier for beginners, but much less powerful for experienced coders, since there can be only 32 variables, and there are limitations on threading and concurrency.
- pbFORTH (http://www.hempeldesigngroup.com/lego/pbFORTH/index.html) FORTH is a reasonably common scripting language, for which Ralph Hempel has written an interpreter which fits on the RCX. This means no compiler on your machine. Furthermore, the latest version allows cooperative multi-tasking and interactive debugging. You can also link sophisticated scripts in order to allow for a

basic form of libraries. This has many of the strengths of LegOS, since it has unlimited variables and threading. Only substantial drawback, as far as I can tell, is learning FORTH :) Ralph tells me there are good tutorials linked to from his site, but I have not explored them.

- Smalltalk (http://www.object-arts.com/Bower/Bot-Kit/): Don't know much about this at all. Let me know!
- TinyVM (http://www.crazyrobots.com/tinyvmsite/index.html): An actual Java VM for the RCX. That's right: write Java, compile on the PC (or a Sun box) and download to the RCX to run. Not yet as sophisticated as legOS, but catching up fast.
- Lego::RCX.pm (http://members.home.com/quillan/lego/rcx.pm.html): If Java or C just aren't your style, TMTOWTDI. That's right, write in Perl to control the RCX. Unlike TinyVM or legOS, this is just a way to send commands to the robot from your computer. However, that can still be very useful; in fact, some would argue more useful. Whatever :)
- Jini (http://pandonia.canberra.edu.au/java/jini/tutorial/MindStorms.xml): Program the RCX in Jini, Sun's Java-based distributed systems architechture. There was a similar document for J++ 6.0 at Aviv Eyal's site. (http://www.dromology.com/mindstorms/spirit.htm), but the entire site is 403: Forbidden at this time.

# Changes and Notes(as if anyone cares :-)

## Change Log

### 0.1.1- 4/19/99

- Added the web compilers.
- Corrected an error in the description of how the motors brake.
- Clarified the compilation of firmdl.c. *(All of the above suggested by Richard Franks.)*

### 0.1.2- 4/20/99

- Standardized most examples on demo.c.
- Remembered the existence of buttons on the RCX.
- Grammar mistakes corrected.
- Deprecated the use of one of the motor patches, on Lou Sortman's advice.

### 0.1.3- 4/21/99

- Fixed problem in touch sensor section (forgot about the <0x2000!) *(error found by Mario Ferrari.)*
- Noted the existence of rom_reset(). *(Thanks to Lou Sortman.)*

### 0.1.4- 5/2/99

- Wrote 4.6 (buttons).
- Started the links page. Still needs to be reorganized at least once, with more emphasis on construction.
- Small cleanups, including moving most notes to self out of text and into Section 9.2.
- Added the mention of the LNP.

### 0.1.5- 5/13/99

- Many grammar and spelling corrections (*corrected by Michael Littman.*)
- Clarified some examples and discussions.
- Added a post-script version to the webpage.
- Added list of pros and cons to Section 1.1.

- Officially changed license to LDP (Section 1.2).
- Completely re-wrote Section 2.1.2 by including the step-by-step instructions that had previously been linked, instead of included.

### 0.1.6- 6/22/99

- Many, many small changes (*suggested by many people.*)

### 0.1.7- 7/9/99

- Incorporated new argc and argv passing into description of execi() (*per Markus L. Noga's request.*)
- First details on rotation sensors and send-raw.

### 0.2.0-pre1- 5/26/00

- First hack at 0.2.4 compatibility.
- Still lots and lots of broken stuff, particularly windows install.

### 0.2.0-pre2- 6/12/00

- Smaller changes than 0.2.0-pre1
- improve clarity of dll and .lx sections.
- add more benefits of legOS to introduction
- removed lots of antiquated stuff.

### 0.2.0-pre3- 7/08/00

- Started converting all URLs to legOS.sourceforge.net URLs. Still not perfect by any means, particularly when it comes to the API docs. At least the server is guaranteed to exist, which is a step up.
- Begun process of correcting file locations- direct-motor.h is now include/dmotor.h, for example.
- Added the Contributors section (1.8) to recognize past contributors whose contributions are no longer directly included in the text.
- Small fixes of lots of sections, plus reorganizations.
- quickly approaching "correctness"- only three non-link re-writes necessary.

### 0.2.0-pre4 7/16/00

- Added the links for TinyVM and RCX.pm.
- described the situation with dll error messages.
- fixed button description.
- Converted to DocBook.

### 0.2.1 8/16/00

- Began converting demo.c to legOS 0.2.4 and Docbook. The new demo.c is still untested, though, and some functions (like rotation sensors) are still lacking demo code. Also, there are still severe formatting problems in the demo code, because of limitations in programlisting.
- Some small changes in the Links section, including mention of the Lego + Linux HOWTO.

### 0.2.3 9/21/00

- Mention new C++ support
- New section on CVS usage
- Mention of Extreme Mindstorms
- Mention of the Kernel Internals guide

### 0.2.4 10/22/00

- Mike Ash contributed a great LNP section, and I converted it to SGML.

## Notes/TODO

- Test the new demo.c. Add rotation sensor demo code.
- 2.4- URL for new (sourceforge based?) web-based compiler, delete old ones
- mention all the last minute changes for 0.2.4, like persistence, and new 0.2.5isms, like velocity.
- SGML: figure out a better formatting for the xref statements.
- Add better C++ description.
- 3- Split into input and output sections? Motor, LCD, and IR can be in output, and light, touch, rotation, and buttons in input. "Mechanical control" is becoming more and more outdated as LNP etc. get added.
- More detail in the dsound section.
- Set up everything for SGML on sourceforge? Makefile might be particularly nice.