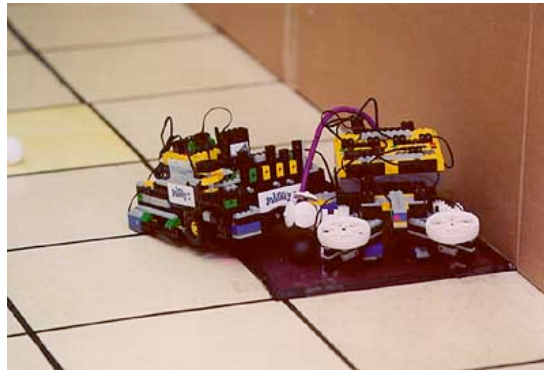


RCXLisp User Manual

Version 1.3

22 October 2003

© MMII,MMIII, MMIV, Frank Klassner



Dedicated to my AI students, who, since the Fall of 1999, have bravely participated in my quest to make Lego MindStorms a viable platform for collegiate computing curricula.

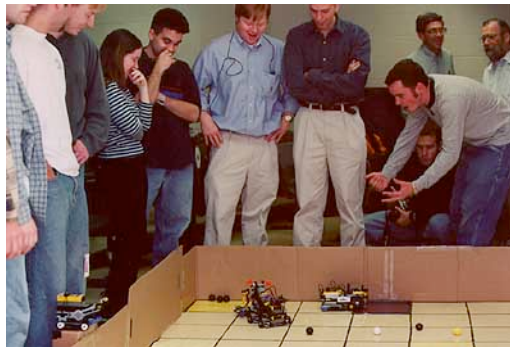
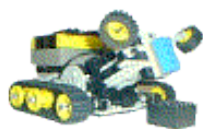


TABLE OF CONTENTS

| | |
|--|-----------|
| INTRODUCTION | 1 |
| INSTALLATION | 3 |
| PC INSTALLATION | 3 |
| MAC INSTALLATION | 4 |
| REMOTE RCXLISP | 6 |
| A LIMITED COMMON LISP INTRODUCTION | 6 |
| <i>Optional Arguments</i> | 6 |
| <i>Keyword Arguments</i> | 7 |
| <i>Using LispWorks</i> | 8 |
| THE “REMOTE RCXLISP” LANGUAGE..... | 10 |
| Common Lisp I/O Streams & “Remote RCXLisp” | 10 |
| “Remote RCXLisp” Stream Argument Convention..... | 11 |
| Return Values of “Remote RCXLisp” Functions..... | 11 |
| RCX:: <i>*standard-rcx-io*</i> | 15 |
| alive..... | 16 |
| battery-power..... | 17 |
| change-rcx-id..... | 18 |
| clock..... | 19 |
| clock-hour..... | 20 |
| clock-minute..... | 21 |
| current-program..... | 22 |
| download-executable..... | 23 |
| download-firmware..... | 25 |
| effector..... | 26 |
| firmware..... | 27 |
| message..... | 28 |
| play-system-sound..... | 29 |
| play-tone..... | 30 |
| rcx-compile-and-download..... | 31 |
| rcx-compile-file..... | 32 |
| rcx-compile-formlist..... | 33 |
| read-rcx-executable..... | 34 |
| send-message..... | 35 |
| sensor..... | 37 |
| set-clock..... | 38 |
| set-clock-fields..... | 39 |
| set-clock-hour..... | 40 |
| set-clock-minute..... | 41 |
| set-current-program..... | 42 |
| set-effector-state..... | 43 |
| set-sensor-state..... | 44 |
| set-transmit-range..... | 47 |
| set-var..... | 48 |
| shutdown..... | 49 |
| start-rcx-program..... | 50 |
| start-rcx-thread..... | 51 |
| start-timer..... | 52 |
| stop-rcx-thread..... | 53 |
| timer..... | 54 |
| using-rcx..... | 55 |
| var..... | 56 |
| with-open-com-port..... | 57 |
| with-open-rcx-stream..... | 58 |
| RCXLISP | 59 |
| FEATURE SUMMARY..... | 59 |
| Control Expressions..... | 59 |

| | |
|---|------------|
| Arithmetic & Logical Operators | 60 |
| Variables & Constants..... | 60 |
| Similarities & Differences with “Remote RCXLisp” | 60 |
| THE “RCXLISP” LANGUAGE | 62 |
| accept-only | 63 |
| change-rcx-id | 64 |
| clear-message..... | 65 |
| clock | 66 |
| clock-hour | 67 |
| clock-minute | 68 |
| CONTROL EXPRESSIONS | 69 |
| current-program | 70 |
| defconstant | 71 |
| defmacro..... | 72 |
| defregister..... | 74 |
| defthread..... | 76 |
| IMPORTANT NOTE ABOUT DEFTHREAD’S SYNTAX!!! | 77 |
| defvar | 78 |
| effector | 80 |
| LOGICAL OPERATORS..... | 81 |
| let, let* | 82 |
| MATH OPERATORS..... | 83 |
| INTEGER CONSTANTS: ALTERNATIVE BASE REPRESENTATION | 83 |
| message | 84 |
| play-system-sound | 85 |
| play-tone..... | 86 |
| random..... | 87 |
| send-message | 88 |
| sensor..... | 89 |
| set-clock-fields..... | 90 |
| set-current-program | 91 |
| set-effector-state | 92 |
| set-sensor-state | 93 |
| set-transmit-range | 96 |
| shutdown | 97 |
| sleep | 98 |
| start-rcx-thread..... | 99 |
| start-timer | 100 |
| stop-rcx-thread | 101 |
| timer..... | 102 |
| APPENDIX A: “REMOTE RCXLISP” PROGRAMMING EXAMPLES | 103 |
| APPENDIX B: “RCXLISP” PROGRAMMING EXAMPLES..... | 109 |



Introduction

Welcome to the RCXLisp programming libraries! This software package allows one to work with the RCX unit from the LEGO MindStorms® kit using Common Lisp. Specifically, RCXLisp lets one

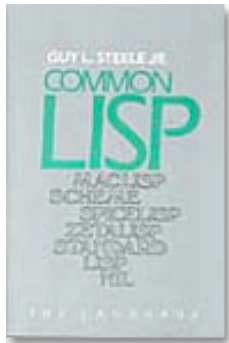
1. remotely control the RCX from a Common Lisp program running on a desktop computer,
2. write RCXLisp programs to run on the RCX,
3. create and compile RCXLisp programs for downloading to RCXs “on the fly,”
4. simultaneously control more than one RCX from a single MindStorms® infrared transceiver tower
5. set up a network of RCX units that can communicate with each other in a targeted manner (as opposed to the “broadcast manner” supported by LEGO’s kit).

The term ‘RCXLisp’ actually refers to two related languages. The first is “Remote RCXLisp,” which is a collection of macros, variables, and functions for remotely controlling RCX units from a desktop. The second language is “RCXLisp” proper, which is a subset of Common Lisp that can be used to write programs for controlling RCXs (with LEGO’s firmware, or with Mnet extended firmware that supports wireless networking and most of the opcodes from LEGO’s firmware 1.0) from onboard the units.

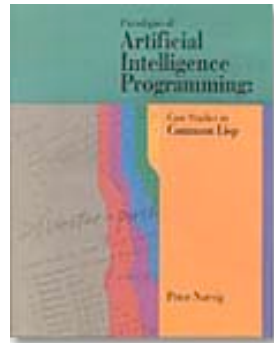
You’ll need one of the following equipment sets to use the RCXLisp environment:

1. A PC with a serial port or USB port, Xanalys’ Common Lisp programming tool LispWorks, and, of course, a LEGO MindStorms RCX unit and either a serial port or USB port IR transceiver tower.
2. A Mac with a serial port, and Virtual PC. You can then run LispWorks on VPC. An RCX unit and a serial IR transceiver tower are also required. RCXLisp has been successfully used on a PowerBook G3 Series and a 466 Dual-Processor G4 Tower under Virtual PC.
3. A “native-Mac” solution exists! In addition to a serial port, RCX unit, and IR transceiver tower, you’ll need Digitool.com’s MCL Common Lisp environment This is only for OS 8 through OS 9.2, however.

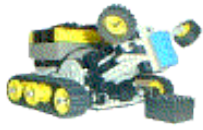
This manual assumes basic familiarity with the Common Lisp language. If you are new to the language, Guy Steele's book, "Common Lisp: The Language," 2nd edition, is a comprehensive treatment of Common Lisp's features and built-in functions. Peter Norvig's "Paradigms of Artificial Intelligence Programming" is an excellent introduction to writing nontrivial programs in Common Lisp.



"Common Lisp:
The Language"



"Paradigms of Artificial
Intelligence Programming"



Installation

PC Installation

The RCXLisp libraries require the presence of Xanalys LispWorks on your PC. A demo version (time-limited to 5 hours for each session) can be found at

http://www.xanalys.com/software_tools/downloads/lw-personal-edition.html

I strongly recommend that you download the version that includes the documentation, even though it will consume an extra 30 MB – you’ll find the documentation very useful for looking up Lisp functions on the fly, and it will be moderately-to-somewhat useful in understanding how to do multithreading in Lispworks (you’ll need to check the MP package).

Once LispWorks is installed, you’ll need to open the “Lispworks Personal” installation folder (available at <http://robotics.csc.villanova.edu>, or on the toplevel of the installation CD), and copy the “RCX.dll” file into the same folder as the LispWorks executable is kept. This folder should be

`C:/Program Files/Xanalys/LispWorks Personal.`

You then need to copy the RCX folder (and all its subfolders) to a folder named

`C:/Program Files/Xanalys/LispWorks Personal/RCX`

That’s it as far as file copying goes. Now, all you have to do is start up LispWorks, and, within the window titled “LispWorks Listener 1,” type

`(LOAD “C:/Program Files/Xanalys/LispWorks Personal/RCX/setup-rcx.lisp”)`

You’ll see a lot of compiling messages fly by. Once the system is compiled, it will automatically be loaded into the Lisp environment. After this step, you’ll have complete access to all the features of the RCXLisp system. Note that every time you quit LispWorks, you’ll have to repeat that last (LOAD ...) step, unless you place that (LOAD ...) function call into a file called “.lispworks” [note the initial period in the name!] and put it in the folder

`C:/Program Files/Xanalys/LispWorks Personal.`

If you prefer to put the RCX library directories somewhere else on your hard drive, then you'll have to edit the "setup-rcx.lisp" file so that the (DEFVAR *RCXLISP-PATH* ...) expression holds the correct pathname string.

The Mnet firmware, designed to give IR networking support, can be found in the "LispWorks Personal/RCX" subdirectory on the installation CD (or on the website) in the file named "Mnet.lgo". You should check out **download-firmware** for details on how to load this or any version of the standard Lego firmware into an RCX unit.

IF you are planning to use Lego's USB IR Tower for communicating with RCXs, you must install Lego's IR Tower driver. To do this without having to install the whole Lego MindStorms Robotics Invention System software package, you must first put the MindStorms installation CD into your CD-ROM reader. Then, quit out of the installer that automatically starts up. Then, manually open up the CD via "My Computer." On the CD you'll find a program labelled "ReinstallUsbTower." Run that and follow directions to install just the USB driver that RCXLisp will depend upon.

Mac Installation

There are two ways to get RCXLisp to run on a Mac. The first involves using Virtual PC and the directions in the previous section. Until the start of April 2002, this was the only way I developed the RCXLisp libraries on my PowerBook G3 Series.

The second way is the "Mac-native" way. For this, the RCXLisp libraries require the presence of Digitool's MCL (Macintosh Common Lisp) on your Mac (OS 8 or later, an OS X version is coming "real soon now"TM). A demo version (time-limited for 1 month, the package is \$95) can be found at

<http://digitool.com/download.html>

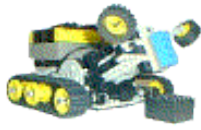
Installation of MCL consists of unzipping the binhex'ed file that you get from the above web page. Inside the MCL folder you'll find the MCL application that you can double-click to start up your Common Lisp experience.

“Installation” of the RCXLisp libraries consists of copying the RCX folder (and all its subfolders) to wherever you want them located. Since there is a lot of freedom as to where you might want to place the RCXLisp library, I’ve defined a variable named *RCXLISP-PATH* in the setup-rcx.lisp file that’s in the RCX folder. You’ll need to edit this file once to set the string to the correct directory pathname for the RCX folder. Once MCL is installed and your setup-rcx.lisp file’s *RCXLISP* path variable is correctly set up, you need to do the following EACH TIME you want to load in the RCXLisp libraries:

STEP ONE, go to the **Tools** pulldown menu, select the **Extensions** submenu, and finally select the **Defsystem** option (5th line from the top of the submenu). MCL will load this extension, which is needed so that the RCXLisp library setup file “knows” in what order to load its own components.

STEP TWO, press **⌘Y** to get the Load File Dialog to come up. Find the setup-rcx.lisp file in the RCX folder, select it for loading, and it will load in the libraries.

STEP THREE, (there’s no step three).



Remote RCXLisp

A LIMITED COMMON LISP INTRODUCTION

The “Remote RCXLisp” language is a collection of macros, functions, and variables that allow one to write a Common Lisp program that can remotely control a Mindstorms® RCX unit running either standard Lego firmware or Mnet firmware. The language is provided via two Common Lisp packages called “RCX” and “RCXLC” (the RCXLisp cross-compiler). With the exception of global package variables, all user features (functions and macros) are exported from these packages.

The design goal of this language was to adhere as closely as possible to the Common Lisp standard in Guy Steele’s text, “Common Lisp: The Language, “ 2nd edition (aka “CLTL2”). This manual assumes some basic familiarity with Common Lisp. However, because it relies heavily on the use of optional and keyword arguments, and because these features of Common Lisp are not generally explored in undergraduate programming courses, the following summary of these features is provided. Note that in “Remote RCXLisp” (and “RCXLisp” as well), no function or macro has both optional and keyword arguments.

Optional Arguments

Any argument declared in a function’s (or macro’s) argument list after the parameter-list keyword “&optional” is an optional argument; invocations of functions with such arguments do not require values to be given for these arguments. If no value is supplied, the default value is used. Order of optional arguments’ declaration is critical and must be observed when providing only some of the optional arguments’ values. For example, assume a function defined with the signature (via DEFUN – see “CLTL2”):

```
(DEFUN fn (a b &optional (c 1) (d 2))
  ...).
```

Any valid invocation of **fn** must have values for the *a* and *b* arguments. So, for example, the invocation **(fn 17 20)** is valid – the Lisp Listener has values for the required arguments *a* and *b*, and uses the default values 1 and 2 for *c* and *d*, respectively. If we want to specify a value of 0 for the *c* argument (and, say, 10 for *a* and 11 for *b*), we would type “**(fn 10 11 0)**” in the Lisp Listener window. If we wanted to specify a value of, say, 35 for the *d* argument and 0 for the *c* argument, we would type “**(fn 10 11 0 35)**” in the Listener. However, there is no way to invoke **fn** with just a given value for *d* and no given value for *c* (i.e. use the default for *c*, but not for *d*). This is because the Listener relies on the order of optional arguments to determine which value goes with which argument.

Keyword Arguments

Any argument declared in a function’s (or macro’s) argument list after the parameter-list keyword “&key” is a keyword argument; invocations of functions/macros with such arguments do not require values to be given for these arguments. If no value is supplied, the default value is used. When a keyword argument’s value is supplied in a function invocation, the value being supplied must be immediately preceded by the “keywordized” name of the argument (i.e. the argument’s name prefixed with a colon). For example, assume a function with the signature

```
(DEFUN fn (a b &key (c 1) (d 2))
  ...).
```

As with the optional-argument example, any valid invocation of **fn** must have values for the *a* and *b* arguments. . If we want to specify a value of 0 for the *c* argument (and, say, 10 for *a* and 11 for *b*), we would type “**(fn 10 11 :c 0)**” in the Lisp Listener window. If we wanted to specify a value of 35 for the *d* argument and a value of 0 for the *c* argument, we could type the invocation “**(fn 10 11 :c 0 :d 35)**” in the Listener. Because keyword arguments have to have a name supplied along with their value, order of keyword arguments’ declaration need not be observed when providing the keyword arguments’ values. In our example, both “**(fn 10 11 :c 0 :d 35)**” and “**(fn 10 11 :d 35 :c 0)**” would be valid invocations of **fn** in the Lisp Listener, and would do the same thing.

Unlike the case with optional arguments, it is possible to provide a value for one keyword argument while letting all other keyword arguments declared before it simply use their defaults. If, in our example, we only wanted to specify a value of 35 for *d* (and use *c*'s default), we could type “(fn 10 11 :d 35)” in the Listener as a valid invocation.

Using LispWorks

Using the LispWorks programming environment is relatively straightforward, even if you've never programmed in Lisp, as long as you've made sure to download the documentation installer from Xanalys. By way of a simple orientation, note that as you use RCXLisp, you'll be using three windows/tools in LispWorks, for the most part: the LispWorks main window (from which you can activate the next windows – under the Tools menu) the Listener window, and the Editor window. The Listener is used when you'd like to evaluate Lisp code interactively – just type or cut-and-paste code into it after the “CL-USER # >” prompt, hit enter, and watch it run in interpreted mode (assuming all your parentheses are balanced! ☺)

Occasionally, your Lisp code will experience a bug, and activate the command-line debugger in the Listener. When that happens, you'll see an error message telling you why the debugger was called, followed by a short menu of keyword options (e.g. :a, :c). The prompt also will have two numbers in it instead of just the command count number (i.e. “CL-USER # #>”). Until you've have more time to learn about the debugger (which is beyond the scope of this humble tome), I recommend always typing “:a”, followed by the return key, to just quit out of the debugger.

The Listener window really doesn't have extensive editing features, nor does it let you save Lisp files for later loading (see the LOAD function's documentation in LispWorks). The Editor window, however, is a full-blown emacs editor that does make life very easy for the Common Lisp programmer. If you want to save a file from the editor, you'll hold down the “control-x control-s” key sequence. To save under a new filename, the “control-x control-w” sequence is used. To open a file in the directory, use “control-x control-f”.

In the Editor you can also press “ESC-x”, and when the bottom status line shows “Extended Command:” you can type “Compile Buffer” or, if you're just compiling one region of code

“Compile Region”. Another useful extended compilation command is “Compile Defun”, which automagically finds the extents of the current function definition your cursor is in the middle of, and compiles just that definition.

Like any other programming language that can run in either interpreted or compiled form, Lisp will execute much more quickly if you compile it. You’ll find the speedup is 10-15x faster – which is very useful when you’re using “Remote RCXLisp” functions on the desktop to control an RCX in situations where you need a fast response rate between desktop code and the RCX.

THE “REMOTE RCXLISP” LANGUAGE

The “Remote RCXLisp” language is a collection of functions, macros, and variables that can be used in the Xanallys and MCL environments with Common Lisp to remotely control RCX units. These units can run either the standard LEGO firmware or the Mnet firmware. Mnet firmware provides the capability to control individual RCXs by assigning them network IDs (8 bits long). This section of the user manual describes the functions, macros, and variables available under “Remote RCXLisp.” The macro and function descriptions use the following conventions. RCXLisp function names are lowercase **boldface**, argument names are *italicized*, and optional and keyword arguments are also *italicized*, but are paired with their default values (shown in plainface text). When basic functions from Common Lisp are used in example code, they will be CAPITALIZED.

Common Lisp I/O Streams & “Remote RCXLisp”

As Guy Steele defines them in “CLTL2,” a stream is an object that is a source or sink of data. Character streams, for example, produce or absorb characters – we can send strings to or read strings from file streams that are attached to files on a hard drive or other file-like device. Carrying the analogy further, in “Remote RCXLisp,” we can create RCX stream objects that are attached to particular RCX units. These streams transfer responses from and carry desktop requests to RCX units running standard LEGO firmware 1.0, standard LEGO firmware 2.0, or Mnet firmware. It is the job of functions in the “Remote RCXLisp” library to construct the correct character strings that need to be sent over the IR transceiver tower to make an RCX perform the functions’ requested actions. It is also the job of these library functions to parse the response from these RCXs, and to make sure that the right RCX gets the right request. In “Remote RCXLisp” we specify RCX streams using the **with-open-rcx-stream** macro. To understand the details behind how to specify these RCX streams, please consult that macro’s entry in this manual.

Once a stream to an RCX unit is defined, that stream must be passed as a final argument to every “Remote RCXLisp” function in order to have the function’s effects applied to that particular

RCX. **Using-rcx** and **RCX::**standard-rcx-io**** can be used to define a default rcx stream. See their documentation for more detail.

“Remote RCXLisp” Stream Argument Convention

In the following documentation, note that if NIL is supplied as a value for the *stream* argument of any “Remote RCXLisp” function, the function will return an ASCII hex string showing the codes that would have been sent to an RCX unit (unless the function is used in a SETF macro, or is any of the functions that start with **set-**, in which case you’ll get an error when the stream argument is nil). Additionally, all *stream* arguments of “Remote RCXLisp” functions take their default value from the **RCX::**standard-rcx-io**** variable’s value.

Return Values of “Remote RCXLisp” Functions

Whenever a “Remote RCXLisp” function is invoked, it will always return **two** values. That is, you’ll see two results (one on one line, the other on the next line) when you invoke a function such as (**set-sensor-state** 1 *:type* *:light* *:stream* str). Let’s assume that the stream here is non-NIL for now. In that case, the first result will be what the RCX unit returns (assuming a reply is received), and the second result (the status) will be either a “t” or a negative number. When the second result is a negative number, that means that an error in transmission occurred, at least as far as the desktop unit is concerned. That is, a negative status number means that an improper (or no) reply was received, and the desktop has no way of “knowing” whether the RCX actually got the IR request. It doesn’t always mean that the RCX did not in fact receive the command request. It just means that the desktop’s IR Tower did not get a valid acknowledgment from the RCX unit. These negative numbers are really only useful for debugging the underlying RCX.dll library file that LispWorks (or other Lisp environments) uses to access the serial and USB ports, so I won’t explain what each value means here. Lisp users should just treat any negative second value as indicating a transmission error situation. Generally, the first result is NIL whenever the status number is negative. Note that some “Remote RCXLisp” functions are predicates that return either “t” or “nil” [see, for example, (**alivep**)]. To determine if a returned first value of NIL is really a response from the RCX rather than an empty placeholder due to a faulty transmission, it is wise to check whether the second returned value is “t” (indicating success) or a negative number. See Common Lisp’s definition of the functions MULTIPLE-VALUE-SETQ,

NTH-VALUE, and VALUE-LIST for information on how to reliably capture multiple values into variables for later checking.

In the case where the stream is NIL, then all “Remote RCXLisp” functions still return two values. Now, though, the first value is a string containing the 2-digit hexadecimal values that would correspond to the what would be sent to an RCX to get it to perform the function’s requested command. The second value is just the stringlength of the first value.

RCX::*local-rcx-id*****

[Remote RCXLisp Variable]

Argument Spec: **RCX::***local-rcx-id*****

Default value: 0.

If you are using the Mnet firmware, you'll need to be aware of this RCX system variable and its use. If you are just using the standard LEGO firmware, you can safely ignore it and leave it at its default setting.

Both the standard LEGO firmware and Mnet firmware provide the RCX with a “message register” in which 8-bit messages can be received from other RCXs or desktops (see the **message** command). These messages are different from the commands sent by the desktop to immediately make the RCX perform actions like turn on motors. They're really only useful for sending 8-bit numbers to the “message register” of programs running locally on the RCXs. Under standard LEGO firmware, RCXs and desktops can broadcast these messages to all other RCXs – they all receive it, whether they ought to or not. Programs running on these RCXs have no way of telling who sent the 8-bit number, and whether they need to act on it.

With Mnet firmware and RCXLisp, however, one can write a Lisp program that accepts 8-bit numbers from a particular “source.” That is, it is possible to write an RCXLisp program to run on an RCX unit and have it only act on messages received from unit # X, where “X” is a number between 0 and 255, inclusive.

(continued)

RCX::*local-rcx-id*****

(continued)

RCX::*local-rcx-id***** contains the Mnet ID of the desktop computer from which you are running “Remote RCXLisp.” It defaults to 0, which is the “broadcast source.” That is, all RCXs running Mnet firmware will accept 8-bit messages (or act on commands) from desktops with an ID of 0 and store them into their “message register” for further use.

Now let’s discuss how to use **RCX::***local-rcx-id***** with different values. Let’s say there are two or more desktop machines in the same room, all broadcasting requests and 8-bit messages to many RCXs (all using Mnet firmware) in the same room. If each desktop (1, 2, 3,...) were to have its own **RCX::***local-rcx-id***** variable set to a different ID (ID_1 , ID_2 , ID_3 , ...), and if each RCX associated with desktop 1 were programmed to look for 8-bit messages with the source number ID_1 (see **accept-only** command in RCXLisp section), then desktop 1 would have “exclusive” 8-bit message communication capabilities with all of those RCXs.

Again, remember that this variable is only useful for controlling the passing of 8-bit messages (or commands) from particular desktops to RCXs. If your application does not depend on such capabilities, you can probably leave this variable set to its default. If **RCX::***local-rcx-id***** is set to a value greater than 255, only the low-order 8 bits are used.

This unexported variable is in the RCX package, which means that you have to prefix references to it with the RCX:: package name. If you forget, you’ll get an error message from the LispWorks or MCL debugger that says ***local-rcx-id*** is undefined or unbound.

RCX::*standard-rcx-io***** **[Remote RCXLisp Variable]**

Argument Spec: **RCX::***standard-rcx-io*****

Default value: nil.

This variable provides the default stream that all “Remote RCXLisp” functions use to determine where to send the functions’ requests. While it is possible to use the Common Lisp SETF or SETQ to change the variable’s value, it is recommended that the **using-rcx** macro be used to locally specify a new default stream. One could also use a Common Lisp LET binding to do the same job.

This unexported variable is in the RCX package, which means that you have to prefix references to it with the “RCX::” package name. If you forget, you’ll get an error message from the LispWorks or MCL debugger that says ***standard-rcx-io*** is undefined or unbound.

alivep **[Remote RCXLisp Function]**

Argument Spec: (**alivep** &optional (*stream* RCX::**standard-rcx-io**))

Returns non-nil if the RCX unit referenced in *stream* is alive and within transmission range of the IR transceiver tower. The actual value returned in this case is the hex string containing the RCX unit's reply to the **alivep** query.

Returns nil if the query times out. This can be caused by, among other possibilities, the RCX unit moving out of range, shutting down, or being blocked by an IR-shielding obstacle.

Alivep is not just a way to ping a particular RCX. Particularly with older serial port IR Towers, users reported that preceding actual requests to RCXs with one or more gratuitous **alivep** requests had the effect of “warming up” the tower so that it (the tower) had a higher success rate of receiving replies from the RCX.

battery-power **[Remote RCXLisp Function]**

Argument Spec: (**battery-power** &optional (*stream* RCX::**standard-rcx-io**))

Battery-power returns the current voltage level (in millivolts) of the batteries in the RCX unit referenced in *stream*. A set of fresh alkaline (non rechargeable) batteries should supply about 8800 – 9400 mV, while rechargeable AA batteries should start out around 7200 – 7500 mV. Below 6500 mV, RCX units start losing their ability to retain firmware while they're shut off.

If there are any errors encountered during transmission of the request, **battery-power** will return nil.

This function is only available in the desktop “Remote RCXLisp” language; **battery-power** is not available in the “RCXLisp” language that runs on RCX units (but will be in a future Mnet release).

change-rcx-id [Remote RCXLisp Function]

Argument Spec: (**change-rcx-id** *id* &optional (*stream* RCX::*standard-rcx-io*))

This function instructs the RCX referenced in *stream* to change its network number to *id*. The *id* argument can be any number between 0 and 255, inclusive. As a side effect, if the request's transmission is successful, the function also changes the rcx-unit id in STREAM to reflect the new *id*. This means you can keep using the same rcx stream after invoking this function – you won't have to create a new one. Returns non-nil if successful, nil otherwise. Note that if an RCX has been set to only listen to commands from a particular RCX or desktop, this request will be ignored unless it comes from that particular RCX or desktop.

There is no built-in mechanism in “Remote RCXLisp” for asking an RCX what its network number is. It is recommended that RCX network numbers be written on a small piece of paper and taped to the RCX unit. Another possibility is to devise a scheme that relates the RCX serial number inscribed on the case to the RCX network number. If you find yourself with an RCX unit that you no longer remember its network number, you can always reset its number by sending the **change-rcx-id** command on an rcx-stream with an :rcx-unit argument of the number 0 (see **with-open-rcx-stream**). Mnet firmware forces all RCX units, regardless of their network number, to follow commands addressed to 0. Be careful that no other RCX units are within 10 feet of the IR Tower if you try this, however, or else you'll change their numbers as well! If all else fails (i.e. the RCX unit is in a very inconsistent state and is not responding to any requests from the desktop, even those addressed to 0), you'll have to pull out the RCX unit's batteries, causing it to lose the firmware. Then just follow the instructions for **download-firmware**.

clock **[Remote RCXLisp Function]**

Argument Spec: (**clock** &optional (*stream* RCX::*standard-rcx-io*))

The onboard RCX clock tracks the time elapsed since the RCX unit was turned on, unless the time was reset by **set-clock**, **set-clock-fields**, **set-clock-hour**, or **set-clock-minute**. The **clock** function returns the value of this clock register, in total minutes, for the RCX unit referenced in *stream*. The returned value will be an integer between 0 and 1439, inclusive.

Clock will return nil if any errors are encountered during transmission of the request.

Clock is setf-able. This means one can use, for example,

```
(SETF (clock stream) 128)
```

to set the clock register of the RCX to hold 128 minutes (“02:08” in military time). See also the **set-clock** command if you prefer not to use the SETF macro.

clock-hour **[Remote RCXLisp Function]**

Argument Spec: (**clock-hour** &optional (*stream* RCX::**standard-rcx-io**))

The onboard RCX clock tracks the time elapsed since the RCX unit was turned on, unless the time was reset by **set-clock**, **set-clock-fields**, **set-clock-hour**, or **set-clock-minute**. The **clock-hour** function returns the hour field of this clock register (military time conventions), for the RCX unit referenced in *stream*. The returned value is an integer between 0 and 23, inclusive.

Clock-hour will return nil if any errors are encountered during transmission of the request.

Clock-hour is setf-able. This means one can use, for example,

```
(SETF (clock-hour stream) 14)
```

to set the hour field of the clock register of the RCX to hold 14 (“14:xx” in military time). See also the **set-clock-hour** command if you prefer not to use the SETF macro.

clock-minute **[Remote RCXLisp Function]**

Argument Spec: (**clock-minute** &optional (*stream* RCX::**standard-rcx-io**))

The onboard RCX clock tracks the time elapsed since the RCX unit was turned on, unless the time was reset by **set-clock**, **set-clock-fields**, **set-clock-hour**, or **set-clock-minute**. The **clock-minute** function returns the minute field of this clock register (military time conventions), for the RCX unit referenced in *stream*. The returned value is an integer between 0 and 59, inclusive.

Clock-minute will return nil if any errors are encountered during transmission of the request.

Clock-minute is setf-able. This means one can use, for example,

```
(SETF (clock-minute stream) 35)
```

to set the minute field of the clock register of the RCX to hold 35 (“xx:35” in military time). See also the **set-clock-minute** command if you prefer not to use the SETF macro.

current-program **[Remote RCXLisp Function]**

Argument Spec: (**current-program** &optional (*stream* RCX::*standard-rx-io*))

Under standard LEGO firmware and Mnet firmware, RCXs can have five different programs stored onboard, numbered 1 through 5. The one that is “current” is the one that is displayed on the RCX LCD to the right of the stick figure. The “current” program is the one that is executed when the RUN button is pressed on the RCX console. The common method for making a program be the “current” one is to repeatedly press the “Prgm” button on the RCX console until the number of that program appears to the right of the stick figure.

Current-program returns the number of the “current program” of the RCX unit referenced in *stream*, or nil if a transmission error occurs. **Current-program** is setf-able. This means one can use, for example,

```
(SETF (current-program stream) 3)
```

to set the RCX’s current program to be 3. See also the **set-current-program** command if you prefer not to use the SETF macro. Setting a program to be “current” will not cause that program to start executing; the main thread (usually the one numbered 0) must be started using **start-rx-thread**.

download-executable [Remote RCXLisp Function]

Argument Spec: (**download-executable** *stream executable* &key (*program* 1)
(*verbose* nil) (*block-size* 100))

This function sends the compiled code in to the RCX unit referenced in the rcx stream in *stream*. *Executable* can be either a string (or pathname – see Guy Steele’s CLTL reference) that names a file containing compiled RCXLisp code, or an executable image created by **rcx-compile-file** or **rcx-compile-formlist**. While the executable is being sent from the IR transceiver to the RCX, you should see a series of dots advance left-to-right on the top left of the RCX unit’s LCD.

Download-executable sends the compiled code to the RCX in chunks. The *block-size* argument defines the maximum number of bytes in these chunks. The maximum feasible value for *block-size* is 500; larger values will make it take so long for a chunk to transfer that the stream will time out. It is better to send several smaller-sized chunks, especially if the RCX is distant from the IR transceiver tower.

Both standard Lego firmware and Mnet firmware support the storage of up to five (labeled 1 through 5) separate programs on the RCX. **Set-current-program** or the “Prgm” button on the RCX can be used to select which one should be current – that is, which program will run when the “Run” button on the RCX is pressed, or when **start-rcx-thread** is invoked on the desktop. The *program* argument to **download-executable** can be used to indicate what the program number of the downloaded code should be.

(continued)

download-executable

(continued)

The port of the *stream* over which code is being transferred should be opened with an *interval-timeout* limit of at least 80. Experiment with larger values depending on ambient light interference and battery power.

The *verbose* argument, when set to non-nil, will cause **download-executable** to print status messages for each chunk of code sent to the RCX unit.

If the code is successfully downloaded, **download-executable** returns t. If any errors are encountered during transmission of the code, **download-executable** will terminate with an error condition.

See **rcx-compile-and-download** for a utility function that combines stream maintenance, compilation, and downloading.

Example Use with **rcx-compile-file**

```
(with-open-com-port (prt 1 :interval-timeout 80)
  (with-open-rcx-stream (str prt :rcx-unit 172)
    (download-executable str
      (rcx-compile-file "C:/program files/xanalys/lispworks personal/rcx/test.lisp")
      :program 1)
    (start-rcx-program 1 str)))
```

;;The above invocation compiles the "test.lisp" file and produces an executable data structure,
 ;;which is then passed to load-executable. The compiled data is sent to RCX unit #172 over the
 ;;indicated stream and serial port. The code is loaded as program #1 on the RCX, and then started.

download-firmware **[Remote RCXLisp Function]**

Argument Spec: (**download-firmware** *port file* &key (*verbose nil*) (*block-size 200*))

This function translates and sends the firmware file referenced in the string *file* through the com port stream referenced in *port*. The file must be in “srec format” – that is, a LEGO firmware file or the Mnet firmware file. While the file is being sent from the IR transceiver attached to the *port*, you should see a quickly-incrementing count appear to the left of the stick figure on the RCX LCD. Make sure that the RCX is within 4 inches of the transceiver. If the firmware is successfully transferred, you will hear a fast-rising chirp from the RCX upon completion of the function’s execution. Both Mnet and LEGO firmware 1.0 will show a count over 1600 on the LCD if loading is successful.

download-firmware sends the firmware file to the RCX in several chunks. The *block-size* argument defines the maximum number of bytes in these chunks. The maximum feasible value for *block-size* is 250; larger values will overflow the port buffer, possibly causing a system crash, or at the least, causing the loss of data to be sent to the RCX. The *port* stream over which firmware is being transferred should be opened with an *interval-timeout* limit of at least 200. Experiment with larger values depending on ambient light interference and battery power.

Example

(with-open-com-port (p 1 :interval-timeout 220)

(download-firmware p “C:/Program Files/Xanalys/LispWorks Personal/RCX/Mnet.lgo”))

effector**[Remote RCXLisp Function]**

Argument Spec: (**effector** *effector* *feature* &optional (*stream* RCX::**standard-rcx-io**))

Returns the current value of the feature named in *feature* of the effector port named in *effector* on the RCX unit referenced in *stream*. The *effector* argument can be :A, :B, or :C. Note that it is not possible to specify a list for this argument (as opposed to **set-effector-state**'s definition). See the entry in this manual for **set-effector-state** for an explanation of the possible features and their values.

message **[Remote RCXLisp Function]**

Argument Spec: (**message** &optional (*stream* RCX::**standard-rcx-io**))

Returns the current integer value of the message register of the RCX unit referenced in *stream*, or nil if an error is encountered during command transmission. The RCX message register is 8 bits long; hence any integer value returned will be between 0 and 255 inclusive.

Use the **send-message** command if you want to place a message into an RCX's message register. Note that there is NO "Remote RCXLisp" command for remotely telling an RCX to transmit the contents of its message register to yet another RCX. Only a local RCXLisp program running on board the RXC can do that.

play-system-sound **[Remote RCXLisp Function]**

Argument Spec: (**play-system-sound** *sound* &optional (*stream* RCX::**standard-rcx-io**))

Causes the RCX referenced in *stream* to play a system sound. *Sound* must be an integer between 0 and 5, inclusive. The following system sounds are available:

| <u>Sound Number</u> | <u>System Sound</u> |
|---------------------|----------------------|
| 0 | “Blip” |
| 1 | Double beep |
| 2 | Descending chirp |
| 3 | Ascending chirp |
| 4 | “Buzz” (Error sound) |
| 5 | Fast Ascending chirp |

Play-system-sound returns the value in *sound* or nil if an error was encountered during command transmission.

play-tone **[Remote RCXLisp Function]**

Argument Spec: (**play-tone** *frequency duration* &optional (*stream* RCX::*standard-rcx-io*))

Causes the RCX referenced in *stream* to play a tone at *frequency* Hertz for *duration* hundredths of a second. Both required arguments must be integers. *Frequency* must be an integer between 0 and 32767, inclusive; *duration* must be an integer between 0 and 255, inclusive. Practically speaking, most people can only hear the RCX's tones up to 15000 Hz. The longest duration for a tone is 2.55 seconds.

If transmission is successful, **play-tone** returns the value in *frequency*, or nil if an error was encountered during command transmission. Note that the RCX buffers at most 5 **play-tone** requests. If you find that when you send a series of tones, some are dropped even though they were successfully transmitted, it is probably because the tone buffer overflowed. In such cases you might need to insert delays between consecutive **play-tone** requests.

rcx-compile-and-download **[Remote RCXLisp Function]**

Argument Spec: (**rcx-compile-and-download** *source* &key
 (*timeout-constant* 300) (*interval-timeout* 40) (*rcx-unit* nil)
 (*portnumber* 1) (*program-number* 1))

This function is a utility that combines the stream maintenance, file compilation, and downloading operations that ordinarily would be performed using **rcx-compile-file**, **rcx-compile-formlist**, **download-executable**, **with-open-com-port**, and **with-open-rcx-stream**.

Rcx-compile-and-download compiles the RCXLisp program in *source*, and, as long as no compilation errors were encountered, downloads the resulting executable to the RCX unit whose network number appears in *rcx-unit*. The *source* argument can be a filename string, a Common Lisp pathname, or a list of forms as for **rcx-compile-formlist**. *Portnumber* can be the numbers 1,2,3, or 4 to indicate which serial port IR tower to use, or the keyword :LEGO-USB-TOWER to dictate use of the Lego USB IR Tower, assuming that the Lego MindStorms USB IR Tower drivers have been installed from the Robotics Invention System 2.0 or later. *Program-number* indicates which slot the compiled program should be downloaded into on the RCX: 1, 2, 3, 4 or 5. *Interval-timeout* and *timeout-constant* serve the same roles as their counterparts in the **with-open-com-port** macro.

This function returns T if no compilation or download errors were encountered, nil if compilation errors were encountered.

rcx-compile-file [Remote RCXLisp Function]

Argument Spec: (**rcx-compile-file** *sourcefile* &key

execname (FORMAT nil "~A" (GENSYM "EXEC")) (*targetfile* nil))

Rcx-compile-file, its related function **rcx-compile-formlist**, and **read-rcx-executable** are the only functions within the "Remote RCXLisp" library that do not send an IR signal to an RCX unit. **Rcx-compile-file** reads the file named in *sourcefile* (either a string or a pathname – see Common Lisp, The Language), compiles the "RCXLisp" sourcecode, and returns an rcx-executable data structure. If no warnings were generated in the course of compilation, the rcx-executable can be passed to **load-executable** to be downloaded into an RCX unit for execution. If *targetfile* is a string or pathname, as a side-effect the rcx-executable is saved to *targetfile*.

Rcx-compile-file will return an rcx-executable if no compilation warnings are generated, nil otherwise. The *execname* argument specifies a name by which the rcx-executable can be distinguished from other rcx-executables. The name of the rcx-executable can be accessed with the accessor **rcx-executable-name**. This naming feature is provided as a means for a desktop program such as a planner to select from a list of preloaded, precompiled programs for on-the-fly downloading to an RCX unit. For example, a planner might use the Common Lisp code:

```
(load-executable rcxstream (FIND-IF #'(lambda (x) (STRING= (rcx-executable-name x)
                                                         "FIND BLACK SQUARE" )
                                     *action-list*))
```

to find and load a program in the global action list for finding black squares on a floor.

rcx-compile-formlist [Remote RCXLisp Function]

Argument Spec: (**rcx-compile-formlist** *forms* &key

(*execname* (FORMAT nil "~A" (GENSYM "EXEC"))) (*targetfile* nil))

Rcx-compile-formlist, its related function **rcx-compile-file**, and **read-rcx-executable** are the only functions within the “Remote RCXLisp” library that do not send an IR signal to an RCX unit. **Rcx-compile-formlist** parses the list of “RCXLisp” forms in *forms*, compiles the forms, and returns an rcx-executable data structure. If no warnings were generated in compilation, the rcx-executable can be passed to **download-executable** to be downloaded into an RCX unit for execution. If *targetfile* is a string or pathname, as a side-effect the rcx-executable is saved to *targetfile*. **Rcx-compile-formlist** returns an rcx-executable if no warnings are generated, nil otherwise. See **rcx-compile-file** for an explanation of the function’s keyword arguments.

This function is provided as a convenient interface for compiling RCXLisp expressions created by, say, a planner. For example, the code:

```
(SETF *PROGRAM* '((defthread (main :primary t) () (dotimes (i 10) (play-tone 300 100) (sleep 20))))
;;;*PROGRAM* just happens to contain a list of symbols that correspond to a program that plays a 300Hz tone 10 times
(with-open-com-port (p 1)
  (with-open-rcx-stream (str p :rcx-unit 66)
    (load-executable str (rcx-compile-formlist *PROGRAM*) :program 2)))
```

shows how an “RCXLisp” program can be created as a list of symbols, then downloaded to an RCX unit. Note well that **rcx-compile-formlist**’s *forms* argument can contain any expression that can appear in an “RCXLisp” file, but that they must be in a list. In other words, a list takes the place of a sourcecode file for this function.

read-rcx-executable **[Remote RCXLisp Function]**

Argument Spec: (**read-rcx-executable** *filename*)

Read-rcx-executable, and the two related functions **rcx-compile-file** and **rcx-compile-formlist**, are the only functions within the “Remote RCXLisp” library that do not send an IR signal to an RCX unit. Thus, they do not have a “stream” argument.

Read-rcx-executable is intended for use with files that contain compiled “RCXLisp” code. The function opens the file or pathname in *filename*, reads in the compiled RCXLisp code, and returns an rcx-executable data structure representing the compiled program. If the file does not contain compiled “RCXLisp” code, nil is returned.

This function’s primary use is for preloading rcx-executables from files and storing the RCX programs until a controlling desktop Lisp program decides that they are needed for execution onboard an RCX unit. For one-shot compiling and downloading, see the utility function **rcx-compile-and-download**.

send-message [Remote RCXLisp Function]

Argument Specs: (**send-message** *msg* &optional (*stream* RCX::*standard-rcx-io*))

This function stores the 8-bit number in *msg* into the message register of the RCX referenced in *stream*. If you wish to find out what is currently stored in the message register of an RCX, use the **message** function. If *msg* is successfully sent, **send-message** returns the value *msg*. If there is a failure in transmission, nil is returned.

Due to limitations of the standard LEGO firmware and the Mnet firmware, **send-message** is implemented to first send the 8-bit message to the target RCX, and then verify that the message was received essentially by calling the **message** function and comparing the returned result with the value in *msg*. This verification protocol suffers from three problems:

1. it cannot determine if the same *msg* was successfully sent twice in a row.
2. **send-message** cannot reliably determine if a message was successfully sent if it was possible for another RCX or another desktop in the same room to have broadcast the same message to the target RCX.
3. it is the responsibility of the target RCX to save message register values; if no local program running on the target RCX is designed to copy the message register value into a “permanent” location, the message register can be overwritten by another (different value) message.

When initially installed on an RCX, the Mnet firmware assumes that the RCX will accept messages from any entity (other RCX or desktop) without regard to what the Mnet network ID of the entity is. Mnet firmware (but not standard LEGO firmware) supports the RCXLisp command **accept-only** (see next section of the manual), which allows programs running on the RCX to limit the IDs from which they will accept messages. Thus, an RCX will not receive a message via **send-message** from a desktop if the RCX’s onboard program has invoked **accept-only** and (continued)

send-message

(continued)the desktop's **RCX::*local-rcx-id*** value does not match the RCX's **accept-only** value. **Accept-only** can therefore be used on a targeted RCX in conjunction with **send-message** on a desktop to help address the last two problems raised in the last paragraph.

A lower-tech solution to the aforementioned last two protocol problems in a networked RCX environment is to simply assign ranges of message values to each RCX. For example, make each RCX's onboard program only send 8-bit messages whose high-order N bits contain the Mnet ID of the RCX. In this way, every RCX is guaranteed to be sending messages distinct from each others' messages.

sensor [Remote RCXLisp Function]

Argument Spec: (**sensor** *sensor* &optional (*mode* :default) (*stream* RCX::*standard-rcx-io*))

For the RCX unit referenced in *stream*, returns the current integer value measured by the sensor port whose number is in the *sensor* argument. Sensor ports are numbered from 1 to 3, inclusive. The range for the returned value depends on how the sensor port was configured by the most recent **set-sensor** invocation, or on the *mode* optional argument. Note that the *mode* argument is recommended for use in temporarily overriding the sensor's base configuration. See **set-sensor** for more information about configuring sensors. The *mode* argument in **sensor** does not permanently reconfigure a sensor. If an error is encountered in transmission of the request, **sensor** returns nil.

The following keywords are recognized for the *mode* argument:

| <u>Keyword</u> | <u>Comments</u> |
|----------------|--|
| :boolean | Reports sensor's value as 0 or 1, depending on which extremum (0 or 1023) the current value is closer to. |
| :raw | Returns sensor's value as an untranslated value between 0 and 1023, inclusive. Due to the manufacturing process, not all sensors may be capable of generating the extrema. |
| :default | Returns sensor's value according to sensor's type & mode set by set-sensor . |

A common programming error in using this function is to forget that if one wants to supply an explicit value for the optional *stream* argument, one must then also provide a value for the *mode* argument.

set-clock **[Remote RCXLisp Function]**

Argument Spec: (**set-clock** *minutes* &optional (*stream* RCX::**standard-rcx-io**))

Sets the clock onboard the RCX unit referenced in *stream* to indicate that an amount of time equal to *minutes* has passed since the unit was turned on. The value of *minutes* can be 0 through 1439, inclusive. The RCX unit's LCD clock display will show this as military time. Note that setting the clock ahead may cause the RCX unit to shut down if the "elapsed time" exceeds any shutdown delay specified by **shutdown**.

If no errors are encountered in sending the request, **set-clock** returns the value passed in *minutes*, otherwise it returns nil.

set-clock-fields **[Remote RCXLisp Function]**

Argument Spec: (**set-clock-fields** *hour minute* &optional (*stream RCX::*standard-rx-io**))

Sets the clock onboard the RCX unit referenced in *stream* to the military time specified in *hour* and *minute*. The value of *minute* can be an integer 0 through 59, inclusive, and the value of *hour* can be an integer 0 through 23, inclusive. The RCX unit's LCD clock display will show this new military time. Note that setting the clock ahead may cause the RCX unit to shut down if the "elapsed time" exceeds any shutdown delay specified by **shutdown**.

If no errors are encountered in sending the request, **set-clock-fields** returns the clock's new value in minutes (e.g. $hour*60 + minute$), otherwise it returns nil. The *hour* and *minute* arguments are both required. If it is desired to change either the hour field or the minute field of the clock independently of the other field, use the **set-clock-hour** or **set-clock-minute** functions.

set-clock-hour **[Remote RCXLisp Function]**

Argument Spec: (**set-clock-hour** *hour* &optional (*stream* RCX::*standard-rex-io*))

Sets the hour field of the clock onboard the RCX unit referenced in *stream* to the integer value specified in *hour* . The value of *hour* can be an integer 0 through 23, inclusive. The RCX unit's LCD clock display will show this new military time. Note that setting the clock ahead may cause the RCX unit to shut down if the "elapsed time" exceeds any shutdown delay specified by **shutdown**.

If no errors are encountered in sending the request, **set-clock-hour** returns the clock's new value in minutes (e.g. *hour**60 + <value-of-minute-field>), otherwise it returns nil.

set-clock-minute **[Remote RCXLisp Function]**

Argument Spec: (**set-clock-minute** *minute* &optional (*stream* RCX::**standard-rx-io**))

Sets the minute field of the clock onboard the RCX unit referenced in *stream* to the integer value specified in *minute*. The value of *minute* can be an integer 0 through 59, inclusive. The RCX unit's LCD clock display will show this new military time. Note that setting the clock ahead may cause the RCX unit to shut down if the "elapsed time" exceeds any shutdown delay specified by **shutdown**.

If no errors are encountered in sending the request, **set-clock-minute** returns the clock's new value in minutes (e.g. <value-of-hour-field>*60 + *minute*), otherwise it returns nil.

set-current-program **[Remote RCXLisp Function]**

Argument Spec: (**set-current-program** *prog-num* &optional (*stream* RCX::**standard-rcx-io**))

This function instructs the RCX unit referenced in *STREAM* to set its current program to be that named in *prog-num* (a number between 1 and 5, inclusive). Note that if an RCX unit is currently running a program, then this function will cause the RCX to stop execution and set its current program to the new program. To make the RCX run the new program, however, the “Run” button on the RCX unit must be pressed, or the primary thread of the new program must be told to start via **start-rcx-thread**. Generally, the primary thread of the current program is assumed to be thread number 0.

Returns nil if there was a failure in transmitting the request, otherwise, it returns the value of *prog-num*.

Since it is very common to first set an RCX unit’s program and then start its primary thread in the course of starting a new program on an RCX unit, the “Remote RCXLisp” function **start-rcx-program** was defined as a convenient encapsulation of this idiom.

set-effector-state [Remote RCXLisp Function]

Argument Spec: (**set-effector-state** *effector feature value*
&optional (*stream RCX::*standard-rcx-io**))

This function configures an effector port named in *effector*. The *effector* argument can be :A, :B, :C, or a quoted list of one or more of these – order is irrelevant. If a list is given, that means that the specified feature/value setting is applied simultaneously to all the effectors in the list.

The features that can be set for an effector (i.e. the allowable keyword values that can be passed in the *feature* argument) are :SPEED, :POWER, and :DIRECTION. The valid values for these features are summarized below:

| | |
|------------|--|
| :SPEED | An integer between 0 (lowest setting) and 7 (highest setting) |
| :POWER | :ON, :OFF (which applies braking action), :FLOAT (which lets effectors “coast” to a stop) |
| :DIRECTION | :FORWARD :BACKWARD :TOGGLE (which reverses the effector’s direction from whatever it currently is) |

An effector port retains its configuration even after a program stops running.

The function returns NIL if an error in transmitting the command is encountered. If the command is successfully transmitted, it returns 0.

set-sensor-state [Remote RCXLisp Function]

Argument Spec: (**set-sensor-state** *sensor* &key (*type* :raw) (*mode* :default) (*slope* 0)
(*clear* nil) (*stream* RCX::*standard-rcx-io*))

This function configures the sensor port in *sensor* (one of 1, 2, or 3) on the RCX unit referenced in *stream*. Configuring a sensor port requires that one or more of the keyword arguments (not counting *stream*) be supplied. Note that **set-sensor-state**'s *stream* argument is a keyword argument, unlike most other "Remote RCXLisp" functions (whose *stream* argument is an optional one).

The *type* of a sensor port describes what kind of sensor is attached to the port. This argument's value can be one of the following keywords - :raw, :touch, :ht-compass, :ht-ultrasound, :light, :rotation, :temperature. While the majority of these keywords' meanings may seem obvious, note that setting a sensor port to be of type :raw means that the sensor's reported values will be untranslated from the sensor's register. That is, a :raw sensor can return a number between 0 and 1023, inclusive. It is possible to "stack" two or more sensors on the same port. If any of these are a light sensor, the sensor port's *type* must be :light, otherwise the RCX will not supply power to the port to make this active sensor work properly.

The :ht-* types refer to sensors manufactured by HiTechnic (www.hitechnic.com) for the MindStorms platform. Currently RCXLisp supports the magnetic compass sensor and the ultrasonic distance sensor. The compass sensor returns a value between 0 and 99 – 0 indicating magnetic North, 25 indicating East, 50 indicating magnetic South, and 75 indicating West. The sensor updates its value 6 times per second. The ultrasonic sensor also returns a value in the 0 – 100 range. This value, however, must be translated according to the following formula to get the correct distance between the sensor and any obstacle whose face is parallel to it:

$$\text{Distance-in-inches} = (\text{value}/2) + 6$$

From the formula, it is clear that the sensor cannot accurately measure distances shorter than 6 inches.

Both ht-* sensors should only use the :percent mode. The next paragraph explains what the mode of a sensor is.

The *mode* of a sensor port describes how the sensor's values will be reported by calls to the **sensor** function. This argument's value can be one of the following keywords - :raw, :boolean, :percent, :celsius, :fahrenheit, :angle, :pulse, :edge, :default. The :fahrenheit and :celsius modes are only useful for the temperature sensor. The :percent mode reports sensor values scaled to the [0, 100] integer range; note that 50 is NOT necessarily the same as the halfway point between the raw value range of [0,1023]. The :boolean mode reports sensor values as 0 or 1, depending on which extremum the raw value is closer to. The :angle mode is only useful for the rotation sensor, and reports the number of "clicks" the sensor has counted since it was last cleared (see below) – there are 16 clicks per rotation. The :edge mode causes a sensor port to report the running number of boolean transitions it detected since it was last cleared (see below). That is, each time a sensor's value flips from 0 to 1 or from 1 to 0, the count is incremented. So, depressing and releasing a touch sensor attached to such a port would add 2 to the count. The :pulse mode causes a sensor to report the running number of 0-1-0 transitions. So, depressing and releasing a touch sensor attached to such a port would only add 1 to the count.

(continued)

set-sensor-state

(continued)

The *slope* of a sensor port should be set to 0, unless you are interested in recalibrating the sensor's sensitivity. According to Kekoa Proudfoot's "RCX Internals" website:

The slope value controls 0/1 detection for the three boolean modes [:boolean, :pulse, and :edge]. A slope of 0 causes raw sensor values greater than 562 to cause a transition to 0 and raw sensor values less than 460 to cause a transition to 1. The hysteresis prevents bouncing between 0 and 1 near the transition point. A slope value in 1..31, inclusive, causes a transition to 0 or to 1 whenever the difference between consecutive raw sensor values exceeds the slope value. Increments larger than the slope result in 0-transitions, while decrements larger than the slope result in 1-transitions. Note the inversions: high raw values correspond to a boolean 0, while low raw values correspond to a boolean 1.

[from http://graphics.stanford.edu/~kekoa/rcx/opcodes.html#set_sensor_type]

The *clear* argument of **set-sensor-state** is used to reset the sensor's running count, if it is in a counting mode (:pulse, :edge, or :angle). If *clear* is given a non-nil value, the port in *sensor* will have its count set to 0. If *clear* is given a nil value (the default for this keyword argument), then no resetting is performed as part of the **set-sensor-state** invocation.

Set-sensor-state returns 0 if the configuration request is successfully transmitted, nil otherwise. This function actually may send up to three separate bytecode instructions to an RCX – if any one of those three fails to execute, **set-sensor-state** will terminate and return nil. However, in such cases, **set-sensor-state** should be re-invoked with original arguments since there is no way to determine how far into the configuration process the function successfully progressed.

set-transmit-range **[Remote RCXLisp Function]**

Argument Spec: (**set-transmit-range** *range* &optional (*stream* RCX::**standard-rcx-io**))

This function tells the RCX referenced in *stream* to set the transmission range of its infrared (IR) port to the setting in *range*. *Range* can be one of two possible values – the keyword :LONG or the keyword :SHORT. :SHORT indicates short range (less than 2 feet), :LONG indicates long range (up to 12 feet). A :SHORT range will require less of a battery power drain to the IR port than a :LONG range will.

Set-transmit-range returns a 0 if the change request was successfully transmitted, nil otherwise. There is no support in either “Remote RCXLisp” or “RCXLisp” for querying what the current setting of the IR port’s range is.

set-var [Remote RCXLisp Function]

Argument Spec: (**set-var** *variable value* &optional (*stream* RCX::**standard-rcx-io**))

Sets the global variable register (numbers 0 through 31, inclusive, are valid arguments) in *variable* on the RCX unit referenced in *stream* to hold the value in *value*. RCX variable registers hold 16-bit signed values. Returns the value in *value* if request was successfully transmitted and acted upon; nil otherwise. In LEGO standard firmware 2.0, there are actually 32 global variable registers and 16 thread-local variables (i.e. 16 locals for each thread), but the 16 thread-local variables cannot be accessed with this function.

Set-var and **var** are only supported as part of the “Remote RCXLisp” library; these functions are not available in the “RCXLisp” language. This language design choice was made to encourage RCX unit programmers to use symbolic variable names rather than work with potentially confusing register numbers. **Defregister** is the only “RCXLisp” function that gives RCX programmers access to variable registers, but only in order to bind symbolic names to particular variable registers.

shutdown

[Remote RCXLisp Function]

Argument Spec: (**shutdown** &optional (*delay* 0) (*stream* RCX::**standard-rcx-io**))

Instructs the RCX unit referenced in *stream* to shut itself down after *delay* minutes of inactivity. *Delay* can be either an integer between 0 and 255, inclusive, or the keyword :INFINITY. If *delay* is :INFINITY, the RCX will not shut down until its battery power runs down (or it is turned off manually) and subsequent **shutdown** invocations will be ignored. A delay of 0 means immediate shutdown.

Time is measured from when the last command was sent to the RCX. This means that if a “Remote RCXLisp” program issues a (**shutdown** 2) request, it must not send any more commands for the next two minutes if it wants the RCX to shut down as soon as possible. Otherwise, the RCX’s countdown to shutdown is reset each time a new command from the “Remote RCXLisp” program is received.

Note that the delay is initialized to 5 minutes when the LEGO firmware is loaded (also true for Mnet firmware).

start-rcx-program **[Remote RCXLisp Function]**

Argument Spec: (**start-rcx-program** *program* &optional (*stream* RCX::*standard-rcx-io*))

This function is actually a shortcut function that does two commonly-combined actions associated with starting programs on an RCX unit. **Start-rcx-program** first attempts to set the current program of the RCX unit referenced in *stream* to the number in *program* (1 through 5, inclusive). If this request is successful, the function then issues a request to the RCX to start thread 0 – the thread assumed by standard Lego firmware and Mnet firmware to be the primary thread of a program. If the second request is successful, the value in *program* is returned. If any errors are encountered during transmission of these requests, nil is returned.

Essentially, this function is equivalent to the Common Lisp expression:

```
(using-rcx stream
  (WHEN (NUMBERP (set-current-program program))
    (start-rcx-thread 0)))
```

There is no “stop-rcx-program” library function, since the stopping of a multi-threaded program can lead to erratic RCX behavior if the threads are not stopped in the correct (implementation-dependent) order. Generally, it is assumed that another call to **set-current-program** or **start-rcx-program** will be adequate for stopping the current program. In the uncommon situation where an RCX program must be stopped and no new program needs to run immediately afterward, it is recommended that **stop-rcx-thread** be called repeatedly on the appropriate threads in the correct sequence to stop a program.

start-rcx-thread [Remote RCXLisp Function]

Argument Spec: (**start-rcx-thread** *thread-index* &optional (*stream* RCX::**standard-rcx-io**))

Starts the thread numbered in *thread-index* (an number between 0 and 9, inclusive, or the :ALL keyword) in the current program in the RCX unit referenced in *stream*. Starting a non-existent thread in an RCX's current program produces no effect. Returns nil if an error in transmitting the request occurred, non-nil otherwise.

If the :ALL keyword is supplied as the value of *thread-index*, all threads in the current program of the RCX are started in sequence, from thread 0 to thread 9.

Some conventions about RCX threads should be noted here. All RCX programs are assumed by Lego firmware and by Mnet firmware to have thread 0 as their primary (i.e. controlling) thread. This assumption means that only when thread 0 is started will the stick figure on the LCD display begin walking – even if other RCX threads have already been started! Since it is possible for an RCXLisp program (see next section) to specify a non-0 thread as the primary thread, one must be careful not to rely on the walking-figure display to indicate whether an RCXLisp program is actually running on an RCX unit.

See the function **start-rcx-program** for a convenient idiom for changing the current program on an RCX and immediately starting its 0 thread. Also see the “RCXLisp” special form **defthread** for information on how to link symbolic thread names in an RCXLisp program to the 0-9 thread number references that “Remote RCXLisp” supports.

start-timer **[Remote RCXLisp Function]**

Argument Spec: (**start-timer** *timer* &optional (*stream* RCX::**standard-rcx-io**))

This function instructs the RCX unit referenced in *stream* to initialize to 0 the timer whose number is in *timer*, and start the timer counting 1/100's of a second. RCX timers are 16-bit registers, which means that you can use an RCX timer to measure up to 655.35 seconds of time. After that point the timer wraps around to 0. RCXs have four timer registers, numbered 0 through 3.

If the request is successfully transmitted to the RCX, **start-timer** returns 0, otherwise it returns nil.

stop-rcx-thread **[Remote RCXLisp Function]**

Argument Spec: (**stop-rcx-thread** *thread-index* &optional (*stream* RCX::**standard-rcx-io**))

Stops the thread numbered in *thread-index* (an number between 0 and 9, inclusive, or the :ALL keyword) in the current program in the RCX unit referenced in *stream*. Returns nil if an error in transmitting the request occurred, non-nil otherwise.

If the :ALL keyword is supplied as the value of *thread-index*, all threads in the current program of the RCX are stopped, simultaneously, which is different from the definition of **start-rcx-thread**.

timer**[Remote RCXLisp Function]**

Argument Spec: (**timer** *timer* &optional ((*stream* RCX::standard-rcx-io*))

RCX timers are 16-bit registers that measure the passage of time in units of 1/100's of a second. Thus, one can use an RCX timer to measure up to 655.35 seconds of time. After that point the timer wraps around to 0. RCXs have four timer registers, numbered 0 through 3.

If the request and response are successfully transmitted, the **timer** function will return the current value of the timer register numbered in *timer* on the RCX unit referenced in *stream* . If any transmission errors are encountered, nil is returned.

using-rcx

[Remote RCXLisp Macro]

Argument Spec: (**using-rcx** *rcx-stream* &rest *body*)

Executes function calls in *body* , assuming *rcx-stream* as the default stream over which to send RCX commands. Effectively, this establishes a new environment with a new binding for RCX::**standard-rcx-io**. **Using-rcx** can be nested, but keep in mind that only the stream in the innermost environment is available as the default; streams of outer environments must be referred to explicitly. The value returned from a **using-rcx** environment is the value returned by the last evaluated expression in the *body*. Multiple values returned from this environment using the Common Lisp function (VALUES ...) will be preserved.

Examples:

```
(using-rcx rcx0
```

```
  (alivep)) ; checks if the RCX unit in rcx0 is alive.
```

```
(using-rcx rcx0
```

```
  (set-effector :A :power :on) ; activates effector :A of rcx0
```

```
  (using-rcx rcx1
```

```
    (WHEN (alivep) ; tests RCX unit of rcx1
```

```
      (set-effector :A :speed 3) ; sets speed of rcx1's :A effector
```

```
      (set-effector :A :direction :forward rcx0) ; sets rcx0's effector :A's direction
```

```
      (set-effector :A :power :on))) ; activates effector :A of rcx1
```

```
  (set-effector :A :power :off)) ; turns off effector :A of rcx0
```

var **[Remote RCXLisp Function]**

Argument Spec: (**var** *number* &optional (*stream* RCX::*standard-rcx-io*))

RCXs have 32 global variable registers, numbered 0 through 31. This function will return the value stored in variable register *number* at the time the request was received by the RCX. If an error is encountered during request transmission, nil is returned.

By using the **var** and **sensor** functions, a desktop computer can repeatedly “poll” an RCX to obtain periodic samples of sensor and variables. This ability is akin to standard LEGO firmware’s support for datalogging – the sampling and storing of sensor data onboard an RCX. Because of “Remote RCXLisp’s” ability to have a desktop “poll” an RCX, “RCXLisp” (running on an RCX) does not support the datalogging features of the standard LEGO firmware.

Var can be used in conjunction with **defregister** in “RCXLisp” (see next section of manual) to write a “Remote RCXLisp” Lisp program that lets the desktop “peek” at the values of variables as they change while an onboard Lisp program is running on the RCX. It is also possible to use **set-var**, and **var** in “Remote RCXLisp” and **defregister** in “RCXLisp” to implement a message-passing protocol superior to that supported by **send-message**.

The **var** function is setf-able. That is, typing (SETF (var 4 *streamvar*) 35) in the Lisp Listener will cause the value 35 to be stored in the RCX global variable register numbered 4. If you prefer not to use the SETF macro, you can use the **set-var** function instead.

with-open-rcx-stream [Remote RCXLisp Macro]

Argument Spec: (**with-open-rcx-stream** (*stream connection*

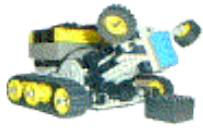
&key (rcx-unit nil) (retries 2) (flip nil))

&rest body)

This macro sets up an environment within which an rcx stream that accesses the port stream in *connection* is created and stored in the variable named in *stream*. The stream is automatically closed on exit from the **with-open-rcx-stream** form. The rcx stream stored in *stream* should be regarded as having dynamic extents and NOT lexical extents (see Examples 4 and 5 in Appendix A for preventing problems with this). The keyword arguments have the following meanings:

| | |
|-----------|---|
| :rcx-unit | Specifies the 8-bit ID of the RCX unit that <i>stream</i> is supposed to target. Use nil if you want to use “Remote RCXLisp” with standard LEGO firmware. |
| :retries | Specifies the maximum number of times that an attempt to send an RCX request to or read an RCX acknowledgement from the stream will be performed, before NIL is returned indicating failure. This limit can override the “retries” limit of the rcx stream’s underlying serial port stream (i.e. the port stream in <i>connection</i>). |
| :flip | If t, witch the state of the stream’s opcode table. This is useful when you’re repeatedly invoking the same “Remote RCXLisp” function in a “with-open-rcx-stream” environment in the Listener. The RCX only acts on a command if its opcode differs from that of the previously sent one. To handle the case of the same command being requested twice in a row, LEGO firmware (ad Mnet) support two versions of each opcode – one with its 4 th bit set to 0, and the other with its 4 th bit set to 1. The <i>flip</i> argument flips the opcode version that the stream will send. See example 3 in Appendix A for an example of when to use :flip and example 6 for when NOT to use :flip |

The value returned from a **with-open-rcx-stream** environment is the value returned by the last evaluated expression in the *body*. Multiple values returned from this environment using the Common Lisp function (VALUES ...) will be preserved. See Appendix A for examples of use, particularly examples 3 and 6, since those show how :flip works. Sometimes you might think that the RCX is not receiving a command, but it is often because you’re unintentionally sending the same command multiple times in a row, and the RCX is not recognizing them as “new” commands to be executed.



RCXLisp



FEATURE SUMMARY

“RCXLisp” is the name of the Common Lisp-like language for programming Mindstorms® RCX units running the standard Lego firmware or the networkable Mnet firmware developed at Villanova University. In summarizing the “RCXLisp” language, this section of the manual assumes basic familiarity with the Common Lisp language. If you are new to the language, Guy Steele’s book, “Common Lisp: The Language,” 2nd edition (CLTL2), is a comprehensive treatment of Common Lisp’s features and built-in functions. Peter Norvig’s “Paradigms of Artificial Intelligence Programming” is an excellent introduction to writing nontrivial programs in Common Lisp.

Control Expressions

“RCXLisp” supports analogs to the following subset of Common Lisp control expressions, along with their standard semantics defined in CLTL2: DOTIMES, IF, LOOP, PROGN, RETURN, and WHEN.

Arithmetic & Logical Operators

The “RCXLisp” language supports 16-bit signed integer arithmetic with the following operators: +, -, *, and / (integer division). “RCXLisp” provides two-argument versions of the >, >=, <, <=, =, /=, and EQUAL Common Lisp comparison operators. It also provides a limited version of the Common Lisp RANDOM function.

“RCXLisp” does not support floating point arithmetic, but it does support the boolean data type (i.e. T and NIL) and certain keyword constants (new keywords cannot be defined). “RCXLisp” also supports the **not**, **and**, and **or** boolean operators, along with their CLTL2 semantics (including the “boolean short-circuit”). Note that just as Common Lisp allows one to use comparison invocations such as “(< 2 x 6)” to test for when the value of x is between 2 and 6, “RCXLisp” also allows this.

Variables & Constants

Constants can be declared in “RCXLisp” programs with **defconstant**, which follows the semantics of the Common Lisp DEFCONSTANT form, and global variables can be declared with **defvar**, whose semantics are only partially the same as those of the Common Lisp DEFVAR form. Values (signed 16-bit integers and T and NIL) can be stored into variables using **setq**, which is similar to the Common Lisp SETQ form. Note that currently there is no analog in “RCXLisp” to the SETF macro! Global variables can also be declared using the **defregister** form.

Similarities & Differences with “Remote RCXLisp”

The “RCXLisp” language supports all of the “Remote RCXLisp” functions (without the *stream* argument), except for the following: **download-firmware**, **download-executable**, **read-rcx-executable**, **rcx-compile-and-download**, **rcx-compile-file**, **rcx-compile-formlist**, **start-rcx-program**, **battery-power**, **var**, and **set-var**. “RCXLisp” also does not support the **using-rcx**, **with-open-com-port**, and **with-open-rcx-stream** macros from the “Remote RCXLisp” language.

The language also provides two special-purpose forms that are neither in the “Remote RCXLisp” language nor the Common Lisp language: **defregister**, which is used to bind symbolic variable names to particular RCX variable registers, and **defthread**, which is used to define RCX threads to run on an RCX unit. “RCXLisp” also includes two functions not found in the “Remote RCXLisp” language: **sleep**, which is used to make an RCX thread “hibernate,” or pause, for a period of time, and **accept-only**, which is used to restrict what other RCX network numbers an RCX will accept messages from.

THE “RCXLISP” LANGUAGE

The version of “RCXLisp” described below is 1.1, and it is assumed that the RCX firmware on which the “RCXLisp” programs are running is either standard Lego firmware 1.0, Lego firmware 2.0, (both without any use of networking functions) or Mnet networking firmware.

The remainder of this section describes the functions and special forms available under the “RCXLisp” language. The descriptions use the following conventions. RCXLisp function names are lowercase **boldface**, argument names are *italicized*, and optional and keyword arguments are also *italicized*, but are paired with their default values (shown in plainface text). When basic functions from Common Lisp are referenced or used in example code, they will be CAPITALIZED. For an introduction to the intricacies of using optional and keyword arguments in Common Lisp and “RCXLisp,” see the “Remote RCXLisp” section of this manual.

Due to limitations in the Lego and Mnet firmware, many of the functions in “RCXLisp” cannot accept variables as arguments – only constant values can be passed in. This limitation is indicated in the argument specification by placing a pound sign (#) at the front of the argument name.

Note well that all “Remote RCXLisp” functions that are included in “RCXLisp” do not have stream arguments. Attempts at specifying stream arguments will cause compilation errors!

Some “RCXLisp” and Common Lisp functions and forms share the same name, but not necessarily the same complete semantics, given the hardware and firmware restrictions of the RCX. These differences were kept to a minimum, and are pointed out in the functions’ and forms’ descriptions below.

accept-only **[RCXLisp Function]**

Argument Spec: (**accept-only** *#id*)

This function is only available with Mnet firmware. It allows an RCX to limit itself to listening for IR messages from sources (desktops or other RCXs) with a network number of *id*. The only exception to this limitation is when a source (desktop or RCX) with a network number of 0 sends an IR message; **accept-only** will not cause an RCX to ignore messages from sources with a 0 network number.

If *id* is 0, then this function will put the RCX into a mode in which it will accept all IR messages without regard for the network number of the sender.

The function returns the value passed in through *id*.

change-rcx-id **[RCXLisp Function]**

Argument Spec: (**change-rcx-id** *#id*)

This function instructs the RCX to change its network number to *id*. See the documentation of the “Remote RCXLisp” version of this function and the documentation of **with-open-rcx-stream** for more details on how to use network numbers. The *id* argument can be any number between 0 and 255, inclusive, but it must be a constant.

This function should generally only be invoked in an “RCXLisp” program if there is no need to communicate with a desktop “Remote RCXLisp” program. If an RCX unit changes its network number independently of a “Remote RCXLisp” program, unless the unit’s onboard “RCXLisp” program is designed to alert the desktop program, there is no way for the desktop program to query the RCX for its new network number.

clear-message **[RCXLisp Function]**

Argument Spec: (**clear-message**)

The **clear-message** function sets the 8-bit message register on an RCX to 0. The function always returns 0.

This function can be used along with **accept-only**, **message**, and **send-message** to support message-passing protocols between two or more RCX units or (less easily so) between an RCX and a desktop “Remote RCXLisp” program. These protocols could eliminate the bugaboo of detecting whether a message has been received twice in a row from the same RCX, or two same separate messages have been received from different RCXs, assuming that no 8-bit message can be 0. Here is a sample setup. After a message has been read from the RCX unit’s message buffer with **message**, **clear-message** can be used to set the message buffer to 0. Now, even if the same message is sent again, the message buffer’s value will observably change from 0 to the new message value. **Accept-only** could be used in this setup to allow reception of new messages only from one particular RCX, eliminating the possibility that the same message could be received from more than one RCX.

clock **[RCXLisp Function]**

Argument Spec: (**clock**)

The onboard RCX clock tracks the time elapsed since the RCX unit was turned on, unless the time was reset by **set-clock**, **set-clock-fields**, **set-clock-hour**, or **set-clock-minute**. The **clock** function returns the value of this clock register, in total minutes.

See also the **set-clock** command.

clock-hour **[RCXLisp Function]**

Argument Spec: (**clock-hour**)

The onboard RCX clock tracks the time elapsed since the RCX unit was turned on, unless the time was reset by **set-clock**, **set-clock-fields**, **set-clock-hour**, or **set-clock-minute**. The **clock-hour** function returns the hour field of this clock register (military time conventions), for the RCX unit. The returned value is an integer between 0 and 23, inclusive.

See also the **set-clock-hour** command.

clock-minute **[RCXLisp Function]**

Argument Spec: (**clock-minute**)

The onboard RCX clock tracks the time elapsed since the RCX unit was turned on, unless the time was reset by **set-clock**, **set-clock-fields**, **set-clock-hour**, or **set-clock-minute**. The **clock-minute** function returns the minute field of this clock register (military time conventions), for the RCX unit. The returned value is an integer between 0 and 59, inclusive.

See also the **set-clock-minute** command.

CONTROL EXPRESSIONS

[RCXLisp Forms]

(**when** *testexpr* &body *body*) (**if** *testexpr* *thenclause* &optional *elseclause*)

(**cond** (*testexpr1* &rest *branchbody1*) ... (*testexprN* &rest *branchbodyN*))

(**loop** &body *body*)

(**return** &optional (*expr* *nil*))

(**progn** &body *body*)

(**dotimes** (*var* *countform* &optional *resultform*) &body *body*)

These control forms follow the syntax and semantics of the same-named Common Lisp forms; see CLTL2 or Xanalys LispWorks' Common Lisp Reference for details on these forms. **Loop** is the same as the basic LOOP form in Common Lisp (i.e. "infinite loop"). It is not the same as the extended loop facility that allows expressions such as (LOOP for x from 1 to 35 doing (PRINT x)). The only way to exit from a **loop** is to use the **return** form, which always exits from the innermost loop. **Return** returns nil, unless an optional return expression is supplied. The *thenclause* and *elseclause* of **if** forms must be single Lisp expressions; if you have to make them contain a sequence of expressions, wrap the sequence in a **progn**.

(**while** *expr* &body *body*)

(**until** *expr* &body *body*)

The **while** and **until** forms are not standard Common Lisp forms, but were included because they are useful shorthand for more cumbersome constructs. The **while** form will repeatedly execute the forms in its *body* as long as *expr* evaluates to non-nil, or if an explicit **return** clause forces an exit. The **until** form will repeatedly execute the forms in its *body* until *expr* evaluates to non-nil. These forms are logically equivalent to the following **loop/when/return** constructs:

;;this is equivalent to (while expr body)

```
(loop
  (when (not expr) (return))
  body )
```

;;this is equivalent to (until expr body)

```
(loop
  (when expr (return))
  body )
```

current-program **[RCXLisp Function]**

Argument Spec: (**current-program**)

Under standard LEGO firmware and Mnet firmware, RCXs can have five different programs stored onboard, numbered 1 through 5. The one that is “current” is the one that is displayed on the RCX LCD to the right of the stick figure. The “current” program is the one that is executed when the RUN button is pressed on the RCX console. The common method for making a program be the “current” one is to repeatedly press the “Prgm” button on the RCX console until the number of that program appears to the right of the stick figure.

Current-program returns the number of the “current program” of the RCX unit.

defconstant **[RCXLisp Special Form]**

Argument Spec: (**defconstant** *name value* &optional (*doc-string* “”))

Defconstant is used to declare a constant named *name* that is global to all threads declared in the same “RCXLisp” program. It follows the semantics of Common Lisp’s DEFCONSTANT form, so the reader is referred to CLTL2 for general information on its usage.

The *value* of a constant can be any expression that evaluates at compile time to an integer, T (the boolean True constant), nil (the boolean False constant), or any keyword constant that could be returned by **sensor-state** or **effector-state**. Be careful to observe that although the *value* expression can include variables, other constants, and function calls, all of the components must have a value at compile time; otherwise, a compilation error is raised. Thus, declarations such as

```
(defconstant threshold (+ 3 (sensor 1 :raw)))
```

are illegal because at compile time there is no way to measure the value of a sensor port.

Defconstant can only appear as a top-level form in a program; it cannot be used within a thread to declare a “local constant.” Usages of **defconstant** within a thread declaration will raise a compilation error.

defmacro **[RCXLisp Special Form]**

Argument Spec: (**defmacro** *name arglist macro-body-generator*)

Defmacro is used to define macros for use in a given “RCXLisp” program. Its semantics are identical to DEFMACRO in Common Lisp, with one exception: macros defined in an RCXLisp file being compiled by **rcx-compile-file** are only available for use within the file’s program, and do not remain available in the desktop environment after compilation is completed. Note that the *macro-body-generator* expression can incorporate any Common Lisp constructs and functions, since the expression is evaluated within the Common Lisp environment on the desktop, not on the RCX itself. However, the code fragment generated by the generator expression must itself only contain “RCXLisp” constructs.

Here is an example of how to declare a useful macro for an “RCXLisp” program. The HiTechnic ultrasonic sensor returns a value between 0 and 100 to indicate distance; to interpret the value as distance in inches, one must divide the sensor value by 2 and add 6. The following macro definition has the effect of adding code to do the conversion:

```
(defmacro ultrasound-distance (sensor-num)
  `( + 6 (/ (sensor ,sensor-num) 2)))

(defthread (tester) ()
  (set-sensor-state 1 :type :ht-ultrasound :mode :percent)
  (set-effector-state '(:A :C) :power :on)
  (while (> (ultrasound-distance 1) 0)
    (set-effector-state '(:A :C) :direction :forward)
    (set-effector-state '(:A :C) :power :off))
```

(continued)

defmacro

(continued)

The effect of using the macro “ultrasound-distance” in the thread is to actually replace the “call” to “ultrasound-distance” with the parenthesized expression “(+ 6 (/ (sensor 1) 2))”, and then compile the expanded thread code for the RCX. Clearly, overuse of macros in “RCXLisp” programs can significantly enlarge programs prior to compilation.

Note the backquote form in the macro-body-generator. A common mistake made by programmers is to forget to use the backquote to indicate that the expression is a template into which the text of the arguments of the macro is to be copied. The expanded template is what is inserted into the thread’s code. The next most common error is to forget to place commas in front of the places where the macro’s arguments (the text, not the values) need to be “copied into place”.

The Common Lisp macro definition mechanism is very powerful. To discuss it fully is beyond the scope of this manual. If you’re interested in getting the most out of macros, I highly recommend Peter Norvig’s “Paradigms of Artificial Intelligence Programming” for a very good introduction to the issues behind writing robust Common Lisp macros.

defregister [RCXLisp Special Form]

Argument Spec: (**defregister** *register-number* *variable-name*
&optional (*init-value* 0) (*doc-string* “”))

Defregister is similar to **defvar** in that it is also used to declare global variables for use by RCX threads. The difference is that it provides a way to instruct the “RCXLisp” compiler to “tie” the symbolic variable *variable-name* to the particular variable register in *register-number* (numbered 0 through 31) so that a “Remote RCXLisp” program on a desktop using **var** could be guaranteed to be accessing a particular “shared” variable. We’ll call these variables “register variables” when they have to be distinguished from those variables defined by **defvar**.

Thus, if an “RCXLisp” program contained the forms

```
(defregister 5 count 0)
(defvar temp)

(defthread (main :primary t) ()
  (set-sensor-state 1 :type :touch :mode :pulse)
  (loop
    (setq temp (sensor 1))
    (when (> temp 200)
      (setq count (- 200 temp))))))
```

then a “Remote RCXLisp” program running on a desktop computer could use (for example)

```
(with-open-com-port (p 1)
  (with-open-rcx-stream (str p :rcx-unit 200 :retries 6)
    (LOOP
      (SETF x (var 5 str))
      (WHEN (AND (NUMBERP x) ;; stop nil from timeout from causing an error
                 (> x 10))
        (RETURN))
      (FORMAT nil “Count exceeded by 10!”)))
```

to determine when and by how much the count variable in the RCX unit exceeded 200.

(continued)

defregister

(continued)

It is important to note that although the temp variable is also allocated to some variable register, there is no way for a “Remote RCXLisp” programmer to know precisely which of the thirty-two possible registers (registers 0 through 31) it is.

Any variable declared by **defregister** is global to all RCX threads in the RCX program in which it appears. If two separate RCX programs are loaded onto an RCX unit (see **load-executable** in the “Remote RCXLisp” section), it is possible to use **defregister** to set up a primitive form of message-passing between the two programs, since RCX variable registers keep their values between program switches. Two RCXLisp programs that agree to store a message in variables tied to the same register could communicate state to each other, assuming that there was a “Remote RCXLisp” program that was switching control of the RCX between them.

The *init-value* of a register variable defaults to 0, just as for variables defined by **defvar**. The *init-value* can be any “RCXLisp” expression that returns an integer, T, nil, or a keyword result from **effector** or **sensor-state**. Its value is not determined until runtime; thus, it is possible to initialize a register variable in “RCXLisp” with the declaration

```
(defregister 5 threshold (sensor 1 :raw))
```

since the expression (sensor 1 :raw) will not be evaluated until the time the program first starts.

Defregister is one of the four forms that are allowed as toplevel forms in an “RCXLisp” program. The other toplevel forms are **defconstant**, **defthread**, and **defvar**. These forms cannot appear inside (i.e. in lower levels) other expressions; a compile error will be raised in such cases.

defthread

[RCXLisp Special Form]

Argument Spec: (**defthread** (*name* &key (*ID* 0) (*primary* nil)) ()
&rest *body*)

Defthread is one of the five forms that are allowed as toplevel forms in an “RCXLisp” program. The other toplevel forms are **defconstant**, **defmacro**, **defvar**, and **defregister**. These forms cannot appear inside (i.e. in lower levels) other expressions; a compile error will be raised in such cases. **Defthread** is used to declare threads that will run concurrently on an RCX unit. Each thread has a symbol assigned as its *name*. Up to 10 threads can be defined for a single program.

Mnet firmware and standard Lego firmware assume that every RCX program has a primary thread that is started first whenever a program is started. In “RCXLisp” a program’s primary thread is indicated by setting the thread declaration’s *primary* keyword argument to T.

It is conceivable that some applications in which an “RCXLisp” and a “Remote RCXLisp” are coordinating execution may need to have the “Remote RCXLisp” program periodically halt and later restart threads on the RCX. The “tying” mechanism for this is to declare threads that might be remotely influenced in the “RCXLisp” program with an *ID* keyword argument. *ID* can be any integer between 0 and 9, inclusive. When a thread is given an explicit ID, say 3, a “Remote RCXLisp” program can use the function calls (**start-rcx-thread** 3 ...) or (**stop-rcx-thread** 3 ...) to influence the execution of that particular thread. The “...” in the function calls is to remind the reader that those functions, since they are being called in a “Remote RCXLisp” program, will require rcxstream arguments. Internally, the 0 thread of any RCX program is assumed to be the
(continued)

defthread

(continued)

primary thread; thus, an alternative (though discouraged) means of declaring a thread to be the primary thread of a program is to give it an *ID* of 0. Unless a thread is intended to be manipulated by a “Remote RCXLisp” program, however, it is strongly advised that threads not be given explicit *IDs* as this makes the resulting “RCXLisp” program too dependent on particular RCX firmware versions.

IMPORTANT NOTE ABOUT DEFTHREAD’S SYNTAX!!!

Be careful to follow the complete argument spec of **defthread**! There are in fact two elements that appear between the symbol “**defthread**” and the body of the thread: the first element is a list that declares the thread’s name, whether it is the primary thread, and possibly what thread number it should have. The second element is a simple empty list. In later versions of “RCXLisp” the ability to pass parameters to a thread will be added. For now, RCXLisp version 1.x requires a placeholder empty parameter list for **defthread** to ensure forward compatibility.

defvar **[RCXLisp Special Form]**

Argument Spec: (**defvar** *variable-name* &optional (*init-value* 0) (*doc-string* ""))

Defvar is used to declare a variable named *name* (initialized to *init-value*) that is global to all threads declared in the same “RCXLisp” program. Uninitialized variables’ values default to 0 (not nil, as one might expect). The *doc-string* string is currently only useful for internal documentation; later versions of “RCXLisp” will make it available in help and debugger systems. The *init-value* argument can be any “RCXLisp” expression that returns a signed 16-bit integer, T, nil, or a keyword that could be returned by **effector-state** or **sensor-state**. Its value is not determined until runtime; thus, it is possible to initialize a variable in “RCXLisp” with the declaration

```
(defvar threshold (sensor 1 :raw))
```

since the expression (**sensor** 1 :raw) will not be evaluated until the time the program first starts.

Note that the action of assigning variables (including those defined by **defregister**) their initial values is performed as an implicit first part of the primary thread of an “RCXLisp” program. That is, until the primary thread of an RCXLisp program starts, global variables’ contents are not guaranteed to contain the correct initial values. For this reason it is strongly recommended that nonprimary threads not be started by “Remote RCXLisp” programs (using **start-rcx-thread**) independent of a primary thread if the nonprimary threads depend on global variables being already initialized.

(continued)

defvar

(continued)

Experienced Common Lisp programmers should note that its semantics differ from those of DEFVAR in CLTL2 in two respects. First, uninitialized variables' values default to 0 at runtime; second, assignment of the initial value is unconditional. In CLTL2, it is stated that DEFVAR will only assign a value to *name* at runtime/loadtime if the variable has never been defined before – otherwise the old value from the earlier declaration is used. For traditional Common Lisp programs, this means that multiple loadings of the same program file with DEFVAR declarations will not override the current values of those declarations' variables in the Lisp environment. Since RCXLisp does not maintain a Listener environment onboard the RCX, it was decided that this part of the DEFVAR semantics could safely be dropped from **defvar**'s semantics.

The initial value of a variable can be any expression that evaluates at runtime to an integer, T (the boolean True constant), nil (the boolean False constant), or any keyword constant that could be returned by an RCXLisp function. Be careful to observe that although the *value* expression can include variables, other constants, and function calls, all of the components must have a value at runtime. Thus, declarations such as

```
(defvar threshold (+ 3 (sensor 1 :raw)))
```

are legal because at runtime it is possible to check the value of a sensor port.

Defvar is one of the four forms that are allowed as toplevel forms in an “RCXLisp” program. The other toplevel forms are **defconstant**, **defthread**, and **defregister**. These forms cannot appear inside (i.e. in lower levels) other expressions; a compile error will be raised in such cases.

effector [RCXLisp Function]

Argument Spec: (**effector** #*effector* #*feature*)

Returns the current value of the feature named in *feature* of the effector port named in *effector* on the RCX unit referenced in *stream*. The *effector* argument can be :A, :B, or :C. Note that it is not possible to specify a list for this argument (as opposed to **set-effector-state**'s definition). The features that can be queried on an effector port are :speed, :direction, and :power. Note that both the *effector* and the *feature* arguments must be constants. A compiletime error is raised otherwise.

The possible keyword values that can be returned for the legal features are summarized below:

| | |
|------------|--|
| :SPEED | An integer between 0 (lowest setting) and 7 (highest setting) |
| :POWER | :ON, :OFF (which applies braking action), :FLOAT (which lets effectors "coast" to a stop) |
| :DIRECTION | :FORWARD :BACKWARD :TOGGLE (which reverses the effector's direction from whatever it currently is) |

Based on the table above, it is possible to write "RCXLisp" code such as

```
(if (equal (effector :B :power)
          :off)
    (progn
      (set-effector-state :A :speed 2)
      (set-effector-state '(:B :A) :power :on))
    (set-effector-state :B :power :off))
```

that uses **set-effector-state** and **effector** together to decide how to change an effector's settings based on what its current state is.

LOGICAL OPERATORS

[RCXLisp Functions]

Argument Specs: (**and** &rest forms) (**or** &rest forms) (**not** form) (**equal** form1 form2)
 (< number &rest nums) (<= number &rest nums) (= number &rest nums)
 (> number &rest nums) (>= number &rest nums) (/= number &rest nums)

The “RCXLisp” logical operators **and**, **or**, and **not** support the same semantics and argument lists as AND, OR, and NOT, respectively, in Common Lisp. This includes the boolean short circuit property. Thus, one can assume that an **and** expression will only evaluate its forms (starting from the leftmost form) until it encounters a form that returns nil – at which point remaining forms are not evaluated. This means, for example, that in the (somewhat contorted) expression

```
(progn
  (setq x 5) (setq y 11)
  (and (> 7 5) (> y 100) (setq x 10)))
```

the variable x will end with a value of 5 and not 10 because the third form in the **and** expression would not be evaluated due to the second form returning nil. The boolean short circuit property for **or** means that **or** expressions will evaluate their forms in order until one of them returns a non-nil value.

The **not** of anything non-nil (keywords, numbers, T) is nil; the **not** of nil is T (true).

The comparison operators of “RCXLisp” have similar semantics to their counterparts in Common Lisp, including support for arbitrary numbers of arguments. Note: if nil is passed via a variable or expression result into a comparison it is coerced to 32768; T is coerced to -32767.

let, let* [RCXLisp Forms]

Argument Specs: (**let** ((var₁ initform₁) ... (var_k initform_k)) &body *body*)
 (**let*** ((var₁ initform₁) ... (var_k initform_k)) &body *body*)

Let and **let*** are forms for declaring and optionally initializing local variables that will be local to all the expressions in *body*. Both forms have an initialization list as their second argument. This list is a list of pairs – one pair for each variable. Inside each of these pairs is the variable name (i.e. *var* in the argument spec above), followed optionally by an expression that will be evaluated to obtain an initial value for the local variable.

The difference between **let** and **let*** is that **let**'s local variables' *initforms* are all evaluated sequentially before their results are assigned to the local variables. This means that in a **let**, it is illegal to use a local variable's value in the *initform* of another local variable.

| | |
|---|---|
| <pre>(defvar global-limit 50) (defthread (dummy) () (let ((x (sensor 1)) (y (+ 2 (message)))) (if (> x global-limit) (set-effector-state '(:A :C) :power :on) (play-system-sound y))))</pre> | <pre>(defvar global-limit 52) (defthread (smarty) () (let* ((x (if (<= (sensor 2) 400) 3 5)) (y (* 12 x))) ;ok to use x in initform in let* (if (> y global-limit) (set-effector-state '(:A :C) :power :on) (play-system-sound x))))</pre> |
|---|---|

The value returned by **let** and **let*** is the value of the last expression in *body*. It is illegal to try to use **return** to exit from a **let/let*** form.

Readers with Common Lisp experience should note that, except for the lack of DECLARE forms, **let** and **let*** follow the CLTL2 specifications of LET and LET*, respectively.

MATH OPERATORS

[RCXLisp Functions]

(+ &rest forms) (- &rest forms) (* &rest forms) (/ &rest forms) (**rem** expr1 expr2)
 (**abs** expr) (**signum** expr) (**1+** expr) (**1-** expr)
 (**logior** &rest numberforms) (**logand** &rest numberforms) (**lognot** expr)

The “RCXLisp” arithmetic functions **+**, **-**, *****, and **/** duplicate the 16-bit integer-arithmetic semantics of the Common Lisp functions of the same names. **Rem** returns the remainder from dividing its *expr1* argument by its *expr2* argument. **Abs** returns the absolute integer value of an expression [-32768 is arbitrarily given an absolute value of 32767]. **Signum** returns the sign [1,0, or -1] of its argument. (**1+** *expr*) is equivalent to (**+** *expr* 1), and (**1-** *expr*) to (**-** *expr* 1).

Logior and **logand** will perform bitwise-or and bitwise-and operations on bitstrings represented as numbers. **Lognot** will perform bitwise complement on a bitstring represented as a number. All bitwise operations assume 16-bit accuracy.

Note: if nil is passed via a variable or expression result into a numeric expression it is coerced to 32768; T is coerced to -32767. The **rcx-compile-*** functions, however, will not allow one to pass T or nil as constant arguments to numeric expressions.

INTEGER CONSTANTS: ALTERNATIVE BASE REPRESENTATION

Integer constants can be represented in decimal base, or in other bases using standard Common Lisp qualifiers. For example, the numeric value of twenty-two can be represented in base two (binary) as #b10110, in base eight (octal) as #o26, in base 16 (hexadecimal) as #x16, or in good ol’ decimal as 22.

message [RCXLisp Function]

Argument Spec: (**message**)

Returns the current integer value of the message register of the RCX unit. The RCX message register is 8 bits long; hence any integer value returned will be between 0 and 255 inclusive. The message register always contains the value of the most recently received message. If the same message is received twice in a row, there is no way to detect this .

Note that if the RCX unit used **accept-only** to restrict what network numbers it will listen to, this will impact what values are stored into the message register. Only 8-bit messages from an RCX with an “accepted” network number will be placed into the register.

play-system-sound **[RCXLisp Function]**

Argument Spec: (**play-system-sound** *sound*)

Causes the RCX to play a system sound. *Sound* can be a constant integer between 0 and 5, inclusive, or a variable or evaluated expression. The following system sounds are available:

| <u>Sound Number</u> | <u>System Sound</u> |
|---------------------|----------------------|
| 0 | “Blip” |
| 1 | Double beep |
| 2 | Descending chirp |
| 3 | Ascending chirp |
| 4 | “Buzz” (Error sound) |
| 5 | Fast Ascending chirp |

Play-system-sound returns the value in *sound*. If a value outside the range of the numbers in the table above is passed in, **play-system-sound** plays no sound, but still returns the passed-in value.

play-tone **[RCXLisp Function]**

Argument Spec: (**play-tone** #*frequency* #*duration*)

Causes the RCX to play a tone at *frequency* Hertz for *duration* hundredths of a second. Both required arguments must be integers in “RCXLisp 1.0”, although a later version of the language will allow the frequency argument to be a variable. *Frequency* must be an integer between 0 and 32767, inclusive; *duration* must be an integer between 0 and 255, inclusive. Practically speaking, most humans can only detect RCX tones at or below 15000. The longest tone duration is 2.55 seconds.

Note that the RCX buffers at most 5 **play-tone** requests. If you find that some tones in a sequence of tone are dropped, it is probably because the tone buffer overflowed. In such cases you might need to insert delays between consecutive **play-tone** requests.

random **[RCXLisp Function]**

Argument Spec: (**random** #*max*)

This function returns a random number between 0 and *max*, exclusive. *Max* can be any constant integer between 0 and 32767, inclusive.

Random does not adhere to the complete specification of Common Lisp's RANDOM function. Among other differences, **random** does not support any means of initializing the random number generator seed, so there is no mechanism for repeatedly testing an "RCXLisp" program on the same "random" sequence of numbers by initializing a seed to the same value for each test.

send-message [RCXLisp Function]

Argument Specs: (**send-message** *msg* &optional (*#target* 0))

This function works with the Mnet firmware only. It sends the 8-bit number in *msg* (which can be a variable or a constant) into the message register of the RCX unit whose network number appears in *target* (which must be a constant). If you wish to find out what is currently stored in the message register of an RCX, use the **message** function. **Send-message** always returns the value *msg*.

When initially installed on an RCX, the Mnet firmware assumes that the RCX will accept messages from any entity (other RCX or desktop) without regard to what the Mnet network ID of the entity is. Mnet firmware (but not standard LEGO firmware) supports the RCXLisp command **accept-only**, which allows programs running on the RCX to limit the IDs from which they will accept messages. Thus, an RCX will not receive a message via **send-message** from a desktop or another RCX if the RCX's onboard program has invoked **accept-only** and the desktop's **RCX::*local-rcx-id*** value (or the other RCX's network number set by **change-rcx-id**) does not match the RCX's **accept-only** value.

sensor [RCXLisp Function]

Argument Spec: (**sensor** *sensor* &optional (#*mode* :default))

This function returns the current integer value measured by the sensor port whose number is in the *sensor* argument (which can be a variable or other evaluated expression!). Sensor ports are numbered from 1 to 3, inclusive. The range for the returned value depends on how the sensor port was configured by the most recent **set-sensor** invocation, or on the *mode* optional argument. Note that the *mode* argument is recommended for use in temporarily overriding the sensor's base configuration. See **set-sensor** for more information about configuring sensors. The *mode* argument in **sensor** does not permanently reconfigure a sensor.

The following keywords are recognized for the *mode* argument:

| <u>Keyword</u> | <u>Comments</u> |
|----------------|--|
| :boolean | Reports sensor's value as 0 or 1, depending on which extremum (0 or 1023) the current value is closer to. |
| :raw | Returns sensor's value as an untranslated value between 0 and 1023, inclusive. Due to the manufacturing process, not all sensors may be capable of generating the extrema. |
| :default | Returns sensor's value according to sensor's type & mode set by set-sensor . |

If the value in *sensor* falls outside of the allowable range (1, 2, or 3), nil is returned.

set-clock-fields **[RCXLisp Function]**

Argument Spec: (**set-clock-fields** *#hour #minute*)

Sets the clock onboard the RCX unit to the military time specified in *hour* and *minute*. The value of *minute* can be a constant integer 0 through 59, inclusive, and the value of *hour* can be a constant integer 0 through 23, inclusive. The RCX unit's LCD clock display will show this new military time. Note that setting the clock ahead may cause the RCX unit to shut down if the "elapsed time" exceeds any shutdown delay specified by **shutdown**.

Set-clock-fields returns the clock's new value in minutes (e.g. $hour*60 + minute$). The *hour* and *minute* arguments are both required. If it is desired to change either the hour field or the minute field of the clock independently of the other field, use the **set-clock-hour** or **set-clock-minute** functions.

This function can be used as a low-tech debugging aid similar to printing "trace" messages onscreen for desktop programming. At given debugging checkpoints, simply call **set-clock-fields** to display predefined numeric messages on the RCX LCD display.

set-current-program **[RCXLisp Function]**

Argument Spec: (**set-current-program** *#prog-num*)

This function instructs the RCX unit to set its current program to be that named in *prog-num* (a constant number between 1 and 5, inclusive). Note that if *prog-num* is different from the number of the currently running program, this will have the effect of stopping the current program. However, it will not cause the newly-assigned current program to start running. To make the RCX run the new current program, the “Run” button on the RCX unit must be pressed, or the primary thread of the new program must be told to start by a “Remote RCXLisp” program on a desktop using **start-rcx-thread**. The primary thread of the current program is assumed to be thread number 0. The function returns the value passed by *prog-num*.

set-effector-state [Remote RCXLisp Function]

Argument Spec: (**set-effector-state** #*effector* *feature* *value*
&optional (*stream* RCX::**standard-rcx-io**))

This function configures an effector port named in *effector*. The *effector* argument can be :A, :B, :C, or a quoted list of one or more of these – order is irrelevant. If a list is given, that means that the specified feature/value setting is applied simultaneously to all the effectors in the list.

The features that can be set for an effector (i.e. the allowable keyword values that can be passed in the *feature* argument) are :SPEED, :POWER, and :DIRECTION. The valid values for these features are summarized below:

| | |
|------------|--|
| :SPEED | An integer between 0 (lowest setting) and 7 (highest setting) |
| :POWER | :ON, :OFF (which applies braking action), :FLOAT (which lets effectors “coast” to a stop) |
| :DIRECTION | :FORWARD :BACKWARD :TOGGLE (which reverses the effector’s direction from whatever it currently is) |

An effector port retains its configuration even after a program stops running.

The function returns NIL if an error in transmitting the command is encountered. If the command is successfully transmitted, it returns 0.

set-sensor-state [RCXLisp Function]

Argument Spec: (**set-sensor-state** *#sensor* &key (#*type* :raw) (#*mode* :default)
(#*slope* 0) (#*clear* nil))

This function configures the sensor port in *sensor* (one of 1, 2, or 3) on the RCX unit. Configuring a sensor port requires that one or more of the keyword arguments be supplied.

The *type* of a sensor port describes what kind of sensor is attached to the port. This argument's value can be one of the following keywords - :raw, :touch, :ht-compass, :ht-ultrasound, :light, rotation, :temperature. While the majority of these keywords' meanings may seem obvious, note that setting a sensor port to be of type :raw means that the sensor's reported values will be untranslated from the sensor's register. That is, a :raw sensor can return a number between 0 and 1023, inclusive. It is possible to "stack" two or more sensors on the same port. If any of these are a light sensor, the sensor port's *type* must be :light, otherwise the RCX will not supply power to the port to make this active sensor work properly.

The :ht-* types refer to sensors manufactured by HiTechnic (www.hitechnic.com) for the MindStorms platform. Currently RCXLisp supports the magnetic compass sensor and the ultrasonic distance sensor. The compass sensor returns a value between 0 and 99 – 0 indicating magnetic North, 25 indicating East, 50 indicating magnetic South, and 75 indicating West. The sensor updates its value 6 times per second. The ultrasonic sensor also returns a value in the 0 – 100 range. This value, however, must be translated according to the following formula to get the
(continued)

set-sensor-state

(continued)

correct distance between the sensor and any obstacle whose face is parallel to it:

$$\text{Distance-in-inches} = (\text{value}/2) + 6$$

From the formula, it is clear that the sensor cannot accurately measure distances shorter than 6 inches.

Both ht-* sensors should only use the :percent mode. The next paragraph explains what the mode of a sensor is.

The *mode* of a sensor port describes how the sensor's values will be reported by calls to the **sensor** function. This argument's value can be one of the following keywords - :raw, :boolean, :percent, :celsius, :fahrenheit, :angle, :pulse, :edge, :default. The :fahrenheit and :celsius modes are only useful for the temperature sensor. The :percent mode reports sensor values scaled to the [0, 100] integer range; note that 50 is NOT necessarily the same as the halfway point between the raw value range of [0,1023]. The :boolean mode reports sensor values as 0 or 1, depending on which extremum the raw value is closer to. The :angle mode is only useful for the rotation sensor, and reports the number of "clicks" the sensor has counted since it was last cleared (see below) – there are 16 clicks per rotation. The :edge mode causes a sensor port to report the running number of boolean transitions it detected since it was last cleared (see below). That is, each time a sensor's value flips from 0 to 1 or from 1 to 0, the count is incremented. So, depressing and releasing a touch sensor attached to such a port would add 2 to the count. The

(continued)

set-sensor-state

(continued)

`:pulse` mode causes a sensor to report the running number of 0-1-0 transitions. So, depressing and releasing a touch sensor attached to such a port would only add 1 to the count.

The *slope* of a sensor port should be set to 0, unless you are interested in recalibrating the sensor's sensitivity. According to Kekoa Proudfoot's "RCX Internals" website:

The slope value controls 0/1 detection for the three boolean modes [`:boolean`, `:pulse`, and `:edge`]. A slope of 0 causes raw sensor values greater than 562 to cause a transition to 0 and raw sensor values less than 460 to cause a transition to 1. The hysteresis prevents bouncing between 0 and 1 near the transition point. A slope value in 1..31, inclusive, causes a transition to 0 or to 1 whenever the difference between consecutive raw sensor values exceeds the slope value. Increments larger than the slope result in 0-transitions, while decrements larger than the slope result in 1-transitions. Note the inversions: high raw values correspond to a boolean 0, while low raw values correspond to a boolean 1.

[from http://graphics.stanford.edu/~kekoa/rcx/opcodes.html#set_sensor_type]

The *clear* argument of **set-sensor-state** is used to reset the sensor's running count, if it is in a counting mode (`:pulse`, `:edge`, or `:angle`). If *clear* is given a non-nil constant value, the port in *sensor* will have its count forced to 0.

Set-sensor-state returns 0 unconditionally.

set-transmit-range **[RCXLisp Function]**

Argument Spec: (**set-transmit-range** *#range*)

This function tells the RCX to set the transmission range of its infrared (IR) port to the setting in *range*. *Range* can be one of two possible constant values – the keyword :LONG or the keyword :SHORT. :SHORT indicates short range (less than 2 feet), :LONG indicates long range (up to 12 feet). A :SHORT range will require less of a battery power drain to the IR port than a :LONG range will.

Set-transmit-range returns a 0 unconditionally. There is no support in either “Remote RCXLisp” or “RCXLisp” for querying what the current setting of the IR port’s range is.

shutdown **[RCXLisp Function]**

Argument Spec: (**shutdown** &optional (*#delay* 0))

Instructs the RCX unit to shut itself down after *delay* minutes of inactivity. *Delay* can be either an integer between 0 and 255, inclusive, or the keyword :INFINITY. If *delay* is :INFINITY, the RCX will not shut down until its battery power runs down (or it is turned off manually) and subsequent **shutdown** invocations will be ignored. A delay of 0 means immediate shutdown. Time is measured from when the program last invoked **shutdown**, or from when the last command from a “Remote RCXLisp” program was received.

Each time a new “Remote RCXLisp” request is received, the countdown is reset. Thus, if an RCX unit wants to prevent interruption of the shutdown countdown, it should also invoke **accept-only** or (only as a last resort!) **change-rx-id** to make it ignore all future commands and 8-bit messages.

Note that the delay is initialized to 5 minutes when the LEGO firmware is loaded (also true for Mnet firmware).

sleep **[RCXLisp Function]**

Argument Spec: (**sleep** *duration*)

The **sleep** function causes the thread in which it is invoked to pause execution for a period of time indicated by *duration* (this is not limited to being a constant argument – it can also be a variable!). The duration is expressed in increments of 10ms. Thus, a *duration* argument of 123 would mean the calling thread must sleep for $123 * 10\text{ms} = 1230 \text{ ms} = 1.23 \text{ seconds}$. *Duration* can range from 0 to 32767. If a negative number is passed in (which is what any number greater than 32767 is treated as), the function has no effect. **Sleep** unconditionally returns the value that was passed in as *duration*.

start-rcx-thread **[RCXLisp Function]**

Argument Spec: (**start-rcx-thread** *#thread*)

Starts the thread named in *thread* (a constant name symbol – not a number as with the “Remote RCXLisp” version of this function – or the :ALL keyword) in the current program in the RCX unit. A compiletime error is raised if the name in *thread* is not assigned to any thread. Lego firmware and Mnet firmware allow up to 10 threads to be declared in an RCXLisp program. **Defregister** is used to assign symbolic names to the low-level numeric identifiers (0 through 9) that the firmware recognizes. If the :ALL keyword is supplied as the value of *thread*, all threads in the current program of the RCX are started in sequence, from thread “0” through thread “9”.

Some conventions about RCX threads should be noted here. All RCX programs are assumed by Lego firmware and by Mnet firmware to have thread 0 as their primary (i.e. controlling) thread. This assumption means that only when thread 0 is started will the stick figure on the LCD display begin walking – even if other RCX threads have already been started! Since it is possible for an RCXLisp program to specify a non-0 thread as the primary thread, be careful not to rely on the walking-figure display to tell whether an RCXLisp program is actually running.

See the “Remote RCXLisp” function **start-rcx-program** for a convenient idiom for changing the current program and immediately starting its 0 thread. Also see the “RCXLisp” special form **defthread** for information on how to link symbolic thread names in an RCXLisp program to the 0-9 thread number references that “Remote RCXLisp” supports.

start-timer **[RCXLisp Function]**

Argument Spec: (**start-timer** *timer*)

This function instructs the RCX unit to initialize to 0 the timer whose number is in *timer*, and start the timer counting 1/10's of a second. RCX timers are 15-bit registers, which means that you can use an RCX timer to measure up to 3276.7 seconds of time. After that point the timer wraps around to 0. RCXs have four timer registers, numbered 0 through 3.

Start-timer always returns 0, even if the value of *timer* falls outside the allowable range of values. Note that the *timer* argument can be a variable or other evaluated expression – it is not limited to being a constant.

stop-rcx-thread **[RCXLisp Function]**

Argument Spec: (**stop-rcx-thread** *thread-name*)

Stops the thread named in *thread-name* (a symbol defined by **defthread** in the current program, or the :ALL keyword) in the current program in the RCX unit. Returns 0 unconditionally.

If the :ALL keyword is supplied as the value of *thread-index*, all threads in the current program of the RCX are stopped, simultaneously, which is different from the definition of **start-rcx-thread**.

Stopping a thread and then starting it again does not cause it to continue from whatever function it was executing when it was stopped; it will be restarted at its beginning.

timer **[RCXLisp Function]**

Argument Spec: (**timer** *timer*)

RCX timers are 15-bit registers that measure the passage of time in units of 1/10's of a second. One can use an RCX timer to measure up to 3276.7 seconds of time. After that point the timer wraps around to 0. RCXs have four timer registers, numbered 0 through 3. Timers can be reset with the **start-timer** function.

Note that the *timer* argument can be a variable, or any evaluated expression.

The **timer** function will return the current value of the timer register numbered in *timer* on the RCX unit. If *timer* falls outside the range of allowable values, nil is returned.

Appendix A: “Remote RCXLisp” Programming Examples

The following six examples are intended to show how to combine the use of RCX streams, serial ports, threading, Common Lisp, and Remote RCXLisp. Built-in functions in Common Lisp are capitalized. All examples assume that the Mnet firmware is being used, which is why all invocations of **with-open-rcx-stream** have an rcx-unit specified. If we were using the examples on standard LEGO firmware, we would have to delete the :rcx-unit argument in **with-open-rcx-stream**. In addition, only examples 1, 2, and 4 will work with standard LEGO firmware because those are the only ones that do not try to control more than one RCX unit.

Example 1:

; This function assumes that there is a motor on effector port :A and another on port :C

(DEFUN full-speed-ahead (s dir)

“This function will make its RCX go at speed S in direction DIR until touch sensor on its ‘2’ port returns a 1.”

(LET ((result 0))

(with-open-com-port (out 2) ; we assume serial port 2 is where the IR tower is plugged in

(with-open-rcx-stream (str out :rcx-unit 1)

(set-effector-state ‘(:A :B :C) :power :off str) ;in case things are in an inconsistent state, turn everything off first

(set-effector-state ‘(:A :C) :speed s str)

(set-effector-state ‘(:A :C) :direction dir str) ; :forward, :backward, or :toggle

; no motion will occur until the next call to set-effector-state

(set-sensor-state 2 :type :touch :mode :boolean :stream str)

(set-effector-state ‘(:A :C) :power :on str)

(LOOP ;this loop will repeat forever until sensor 2 returns a 1

(SETF result (sensor 2 :default str))

(WHEN (AND (NUMBERP result) ;needed to keep = from causing error if sensor function returns nil.

(= result 1))

(RETURN)))

(set-effector-state ‘(:A :C) :power :float str))))))

Example 2:

```
(DEFUN full-speed-ahead2 (r s dir)
```

“This will make the rcx in R go at speed S in direction DIR until touch sensor on its ‘2’ port returns 1.”

```
(LET ((result 0))
  (using-rcx r
    (set-effector-state ‘(:A :B :C) :power :off) ;in case things are in an inconsistent state, turn everything off first
    (set-effector-state ‘(:A :C) :speed s)
    (set-effector-state ‘(:A :C) :direction dir) ; :forward, :backward, or :toggle
      ; no motion will occur until the next call to set-effector-state
    (set-sensor-state 2 :type :touch :mode :boolean)
    (set-effector-state ‘(:A :C) :power :on)
    (LOOP ;this loop will repeat forever until sensor 2 returns a 1
      (SETF result (sensor 2))
      (WHEN (AND (NUMBERP result) ;needed to keep = from causing error if sensor function returns nil.
        (= result 1))
        (RETURN)))
    (set-effector-state ‘(:A :C) :power :float))))))
```

```
(DEFUN test ()
```

“This will make unit 1 go backward at speed 5 until its touch sensor is pushed. Only after that happens will unit 2 go forward at speed 1 until its touch sensor is pushed.”

```
(with-open-com-port (p 1)
  (with-open-rcx-stream (rcx1 p :rcx-unit 1)
    (with-open-rcx-stream (rcx2 p :rcx-unit 2)
      (full-speed-ahead2 rcx1 5 :backward)
      (full-speed-ahead2 rcx2 1 :forward))))))
```

Example 3:

; This is a nice function to define for when you're trying out individual Remote RCXLisp commands at the Listener.

```
(DEFUN tell-rcx (id fn &rest args)
  (with-open-com-port (port 1 :interval-timeout 70)
    (with-open-rcx-stream (rcx port :rcx-unit id :flip t)
      (using-rcx rcx
        (apply fn args))))))
```

;example invocations:

;(tell-rcx 3 #sensor 2) → will return the value at sensor 2 on RCX unit 3

;(tell-rcx 1 #set-effector-state :B :power :on) → will turn on effector port :B of RCX unit 1

;It would however be a bad idea to write long "program snippets" like

```
;( (COND ((> (tell-rcx 1 #sensor 1 :RAW) 100)
;         (FORMAT t "LIGHT SENSOR IS HIGH"))
;       ((= (tell-rcx 1 #sensor 2) 1)
;         (FORMAT t "TOUCH SENSOR IS ON.")
;       (tell-rcx 1 #set-effector-state '(A :C) :direction :backward)))
```

; because of all the overhead [e.g. time delay] in repeatedly opening and closing the ports and streams.

;;the only reason that the :flip argument is specified for **with-open-rcx-stream** is that in case we use "tell-rcx" twice in a row to

;;send the same command to the RCX, the sequence bit of the two requests will be forced to be different.

;;

;;Remember, each time an rcx-stream is opened, (assuming no flipping) its sequence bits for all commands are set to 0. Thus,

;; without the :flip argument, successive calls to "tell-rcx" would always start with the 0 sequence bit. Two 0 bits (or two 1bits) in a

;;row for the same command requested twice in a row will cause the RCX to not recognize the second request as a distinct new

;;repeated request. Hence, the need for :flip. Note that "flip forces the flipping of the bits in the global table from which the

;;**Initialization** of new stream will occur; this means that successive flips in a function will cause the global init tables to

;;"remember the old state and flip according across stream creations done between different function invocations.

Example 4:

;;NOTE: As written, this program will NOT work in MCL. You'll need to figure out how
 ;;to create processes/threads in MCL to replace the functions here called from the MP package.

```
(DEFUN test-rcx1 (port)
```

```
  "This function is defined to only communicate with RCX unit number 1."
```

```
  (with-open-rcx-stream (rcx1 port :retries 4 :rcx-unit 1) ;communicate with RCX #1 over the serial port in PORT
    (using-rcx rcx1
      (FORMAT t "~%RCX1 var 3--->~d~%" (var 3))
      (play-tone 800 250) (play-system-sound 4) (set-var 3 80))))
```

```
(DEFUN test-rcx2 (port)
```

```
  "This function is defined to only communicate with RCX unit number 5."
```

```
  (with-open-rcx-stream (rcx2 port :retries 4 :rcx-unit 5) ;communicate with RCX #5 over the serial port in PORT
    (using-rcx rcx2
      (FORMAT t "~%RCX2 var 3--->~d~%" (var 3))
      (play-tone 300 250) (play-system-sound 3) (set-var 3 100))))
```

```
(DEFUN trial1 ()
```

```
  "This function will invoke each test in sequence, using the serial port stream created in PORTSTREAM."
```

```
  (with-open-com-port (portstream 1) ; assume IR tower is connected to serial port 1 here.
    (test-rcx2 portstream) (test-rcx1 portstream)))
```

```
(DEFUN trial2 ()
```

```
  "This function will cause the creation of a thread for each test. Each test will use the same serial port.
  Since all the Remote RCXLisp commands are threadsafe, there is no danger of two commands from
  different threads grabbing the serial port at the same time."
```

```
  (with-open-com-port (p 1)
    (LET ((a (mp:process-run-function "fn1" nil #'test-rcx1 p))
          (b (mp:process-run-function "fn2" nil #'test-rcx2 p)))
      (LOOP ; for an explanation of the need for this loop, see Example 5.
        (WHEN (AND (NOT (mp::process-alive-p a))
                  (NOT (mp::process-alive-p b)))
          (RETURN t))))))
```

```
;; You'll need to look up mp::process-run-function and mp::process-alive-p in
;; Xanalis' User Guide to understand how multitasking is set up in Common Lisp
```

Example 5:

;; NOTE: This program will NOT work in MCL. You'll need to figure out how to
 ;; spawn threads/processes in MCL to replace the functions called here from the
 ;; "mp" package.

(DEFUN monitor-time (rcx-stream)

"This function loops continuously until clock of rcx unit in RCX-STREAM has more than 3 minutes recorded."

(set-clock 0 rcx-stream)

(LOOP

(WHEN (AND (NUMBERP (clock rcx-stream)) ; in case a NIL is returned
 (> (clock rcx-stream) 3))

(RETURN))

(mp::process-allow-scheduling)) ; this allows interleaving of another process with the one executing monitor-time

(FORMAT t "~%Time test passed!~%"

'time-test)

(DEFUN monitor-program (rcx-stream)

"This function loops continuously until current-program of the rcx unit in RCX-STREAM is set to 1."

(set-current-program 2 rcx-stream)

(LOOP

(WHEN (AND (NUMBERP (current-program rcx-stream))
 (NOT (= (current-program rcx-stream) 1)))

(RETURN))

(mp::process-allow-scheduling))

(FORMAT t "~%Program test passed!~%"

'program-test)

(DEFUN threadify-time (thread-namestring stream)

"Runs monitor-time function in its own thread within LispWorks."

(mp:process-run-function thread-namestring nil #'monitor-time stream))

(DEFUN threadify-program (thread-namestring stream)

"Runs monitor-program function in its own thread within LispWorks."

(mp:process-run-function thread-namestring nil #'monitor-program stream))

(DEFUN threadtest1 ()

(LET ((p1 nil) (p2 nil))

(with-open-com-port (p 1 :interval-timeout 50)

(with-open-rcx-stream (rcx0 p :rcx-unit 1 :flip t)

(SETF p1 (threadify-time "TIME-CHECK" rcx0)

p2 (threadify-program "PROG-CHECK" rcx0))

(LOOP ; the loop is to keep the 'parent thread' running threadtest1 from

; terminating before one of the children threads. If this loop were not here,

; then threadtest1 would terminate "immediately" after spawning the two

; child threads, which would mean that the stream in RCX0 would no longer

; be an open stream, which would then cause the Remote RCXLisp commands

; in the child threads to generate errors.

(mp::process-allow-scheduling)

(WHEN (NOT (mp::process-alive-p p1))

(mp:process-kill p2)

(RETURN p1)) ; return the process that finished first

(WHEN (NOT (mp::process-alive-p p2))

(mp:process-kill p1)

(RETURN p2)) ; return the process that finished first

))))

Example 6

```
;;this is actually a two-part example to show a BAD use of the :flip argument in with-open-rcx-stream
;;a good example can be found in example #3, which implements "tell-rcx". "Tell-rcx" is intended as a
;;shortcut for sending single commands to the RCX without having to type out all the "with-open-..."
;;statements
```

```
;;first, a good example. This version of the function remotely makes the RCX with network number 12
;;play all 6 of its system sounds.
```

```
(DEFUN play-all-sounds ( )
  (with-open-com-port (p 1)
    (with-open-rcx-stream (s p :rcx-unit 12)
      (DOTIMES (num 6)
        (play-system-sound num s)
        (DOTIMES (x 10000) ;;! this oop is just to waste time on the desktop between play-sound requests so that
          (* x 1)))))) ;;all the play-sound requests don't overrun the RCX's buffer.
```

```
;;now, a bad example.
```

```
(DEFUN play-all-sounds-bad ( )
  (with-open-com-port (p 1)
    (with-open-rcx-stream (s p :rcx-unit 12 :flip t)
      (DOTIMES (num 6)
        (play-system-sound num s)
        (DOTIMES (x 10000)
          (* x 1))))))
```

```
;;this second example is intended to show how over-reliance on :flip can cause problems. This code might
;;be written by a programmer who observes that there is only one "Remote RCXLisp" command in the
;;code, so they'd want to be sure that the next time the function "play-all-sounds-bad" is called, the
;;sequence bit from the last play-system-sound request of the last invocation would be different from the
;;first play-system-sound request of the new invocation. Thus, they'd think they'd need the :flip argument.
```

```
;;In fact, what the second function will do is consistently cause the first system sound to never happen.
;;Here's why. Let's assume that the first invocation has the sequence bit set to 0. Since there are an even
;;number of play-system-sound calls, the bit will flip 0-1-0-1-0-1 during the function's execution. Without
;;the :flip, the next time the function is invoked, the sequence bit for the "play-system-sound" request will
;;be back at 0, which is different from the 1 from the previous invocation. With the :flip argument, however,
;;the next time the function is invoked, the sequence bit is "pre-flipped" from its 0 to 1. Since this 1 is not
;;different from the last "play-system-sound" sequence bit of the earlier function invocation, the first "play-
;;system-sound" request of the immediate next invocation of "play-all-sounds-bad" will be ignored by the
;;RCX. The next invocation of "play-system-sound," however, would automatically have its sequence bit
;;flipped to 0, which means that the second system-sound (and later requests in that loop) would play just
;;fine.
```

Appendix B: “RCXLisp” Programming Examples

Example 1:

```

; This program shows how one could make an RCXLisp program that does the same actions as the
; “Remote RCXLisp” program in example 1 of appendix A.
;; assumes that there is a motor on effector port :A and another on port :C
;;; NOTICE that there are no stream arguments for any of the functions here!
(defconstant s 3)
(defconstant dir :forward)
(defthread (full-speed-ahead :primary t) ()
  “This thread will make its RCX go at speed S in direction DIR until touch sensor on its ‘2’ port returns a 1.”
  (let ((result)
    (set-effector-state ‘(:A :B :C) :power :off) ;in case things are in an inconsistent state, turn everything off first
    (set-effector-state ‘(:A :C) :speed s)
    (set-effector-state ‘(:A :C) :direction dir) ; :forward, :backward, or :toggle
      ; no motion will occur until the next call to set-effector-state
    (set-sensor-state 2 :type :touch :mode :boolean)
    (set-effector-state ‘(:A :C) :power :on)
    (loop ;this loop will repeat forever until sensor 2 returns a 1
      (setq result (sensor 2)
        (when (= result 1)
          (return)))
    (set-effector-state ‘(:A :C) :power :float)))

```

Example 2:

```

; This program loops on variable I to generate all 6 system sounds.
;;; NOTICE that there are no stream arguments for any of the functions here!

(defthread (make-noises :primary t) ()
  “This thread will make its RCX play all .6 system sounds, separated in time by 1 second”
  (dotimes (n 6) ;; n gets iteratively bound to 0,1,2,3,4, and 5.
    (play-system-sound n)
    (sleep 100)))

```

Example 3

;;This example was borrowed from Markus Overmars' Not Quite C programmer guide. The

;;original NQC code is

```
;; #define MOVE_TIME 100
```

```
;; #define TURN_TIME 85
```

```
;;
```

```
;; task main
```

```
;; {
```

```
;; while(true)
```

```
;; {
```

```
;; Fwd(OUT_A+OUT_C,7);
```

```
;; Sleep(MOVE_TIME);
```

```
;; If (Random(1) == 0)
```

```
;; {
```

```
;; Rev(OUT_C,7);
```

```
;; }
```

```
;; else
```

```
;; {
```

```
;; Rev(OUT_A,7);
```

```
;; }
```

```
;; Sleep(TURN_TIME);
```

```
;; }
```

```
;;}
```

;; The purpose of both programs is to make an RCX perform a "random walk" by randomly
 ;; turning left or right every second.

```
(defconstant *MOVE-TIME* 100)
```

```
(defconstant *TURN-TIME* 85)
```

```
(defthread (main :primary t) ( )
```

```
  (set-effector-state '(:A :C) :speed 7)
```

```
  (set-effector-state '(:A :C) :power :on)
```

```
  (loop
```

```
    (set-effector-state '(:A :C) :direction :forward)
```

```
    (sleep *MOVE-TIME*)
```

```
    (if (= (random 1) 0)
```

```
      (set-effector-state :C :direction :backward)
```

```
      (set-effector-state :A :direction :backward))
```

```
    (sleep *TURN-TIME*)))
```

Example 4

;;This code is an expansion of the earlier example to show how to use COND to cover several
 ;;possible random behaviors.

```
(defconstant *MOVE-TIME* 100)
(defconstant *TURN-TIME* 40)

(defthread (main :primary t) ( )
  (let ((r 0))
    (set-effector-state '(:A :C) :speed 7)
    (set-effector-state '(:A :C) :power :on)
    (loop
      (set-effector-state '(:A :C) :direction :forward)
      (sleep *MOVE-TIME*)
      (setq r (random 100))
      (cond ((< r 20)
              (set-effector-state :C :direction :backward) ;;20% of the time turn right for long time
              (sleep (* 2 *TURN-TIME*)))
            ((< r 40)
              (set-effector-state :C :direction :backward) ;;20% of the time turn right for short time
              (sleep *TURN-TIME*))
            ((< r 60)
              (sleep (* 3 *TURN-TIME*))) ;;20% of the time keep going straight
            ((< r 80)
              (set-effector-state :A :direction :backward) ;;20% of the time turn left for short time
              (sleep *TURN-TIME*))
            (t
              (set-effector-state :A :direction :backward) ;;20% of the time turn left for long time
              (sleep (* 2 *TURN-TIME*)))))))))
```


This page intentionally left blank.