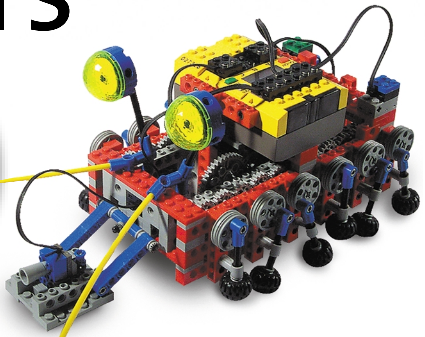
SYNGRESS®

1 YEAR UPGRADE
BUYER PROTECTION PLAN

SYNGRESS.COM 1 YEAR UPGRADE BUYER PROTECTION PLAN

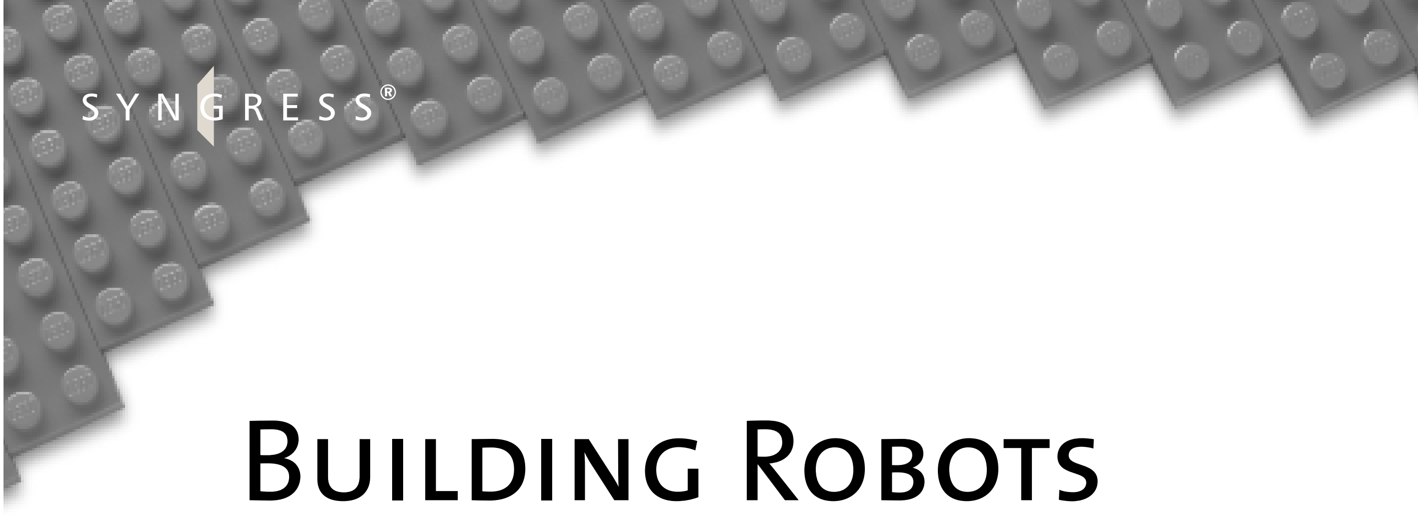
Building Robots with LEGO® Mindstorms™



The ULTIMATE Tool for Mindstorms Maniacs!

- Discover the Undocumented Secrets behind the Design of the Mindstorms System
- Become Inspired by the Techniques of World-Class Mindstorms Masters
- Build a Competitive Edge for Your Next Mindstorms Robotic Competition

Mario Ferrari
Giulio Ferrari
Ralph Hempel
Technical Editor

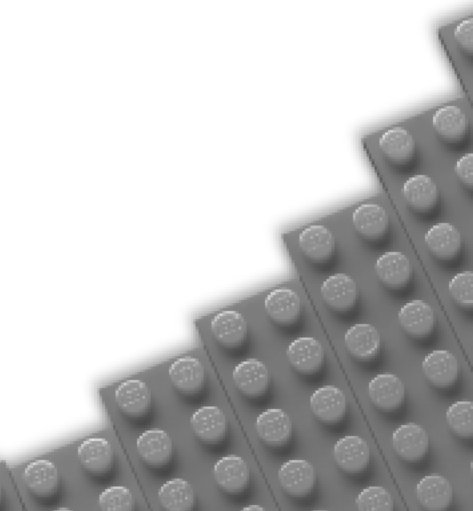


SYNGRESS®

Building Robots with LEGO® MINDSTORMS

The ULTIMATE Tool for MINDSTORMS Maniacs!

Mario Ferrari
Giulio Ferrari
Ralph Hempel Technical Editor



KEY	**SERIAL NUMBER**
001	B8EL495GK4
002	2NVA4UHBBJ
003	CJGE946M43
004	3BVNAM7L4T
005	D384NSARSD
006	4ZMWAQEKFK
007	FMAPPW8GN9
008	XSLEKRK2FB
009	QMV9DSRUJT
010	5KNAPFRPAR

PUBLISHED BY
Syngress Publishing, Inc.
800 Hingham Street
Rockland, MA 02370

Building Robots with LEGO MINDSTORMS

Printed in the United States of America

1 2 3 4 5 6 7 8 9 0

ISBN: 1-928994-67-9

Technical Editor: Ralph Hempel
Co-Publisher: Richard Kristof
Acquisitions Editor: Catherine B. Nolan
Developmental Editor: Kate Glennon
Freelance Editorial Manager: Maribeth Corona-Evans

Cover Designer: Michael Kavish
Page Layout and Art by: Shannon Tozier
Copy Editor: Michael McGee
Indexer: Robert Saigh

Distributed by Publishers Group West in the United States and Jaguar Book Group in Canada.

Letter from the Publisher

When I co-founded Syngress in 1997 with Amorette Pedersen, we decided to forego the opportunity to include the ubiquitous "Letter from the Publisher" in the front of Syngress books. Our books are of the highest quality, written by content experts, and they've spoken quite well for themselves without any help from us.

However, the publication of *Building Robots with LEGO MINDSTORMS* entitles me to a one-time exemption from our rule. I am lucky enough to be the father of nine-year-old Sam Williams, who has taught me (among many important things) the joy of building with LEGO. Since helping Sam put together his first bricks at two years old to programming our latest MINDSTORMS robot (the optimistically named "Chore-Doer 3000"), I have derived hundreds of hours of pleasure creating projects with Sam. Perhaps the most ingenious thing about LEGO products, particularly the MINDSTORMS, is that the same product can be as challenging and enjoyable to a 43 year old as it is to a nine year old.

When presented with the chance to publish Mario and Giulio Ferrari's book, I jumped at the opportunity. As I read the manuscript, I could sense the authors had the same passion for creating with LEGO MINDSTORMS that Sam and I have. I knew immediately that there was a market of at least two people for the book!

I had the opportunity to meet Mario Ferrari at the Frankfurt Book Fair just weeks prior to this book's publication. I am American and Mario is Italian, but the language we spoke was that of two parents who have discovered a common passion to share with our nine and ten year old sons.

I hope you enjoy reading this book as much as we have enjoyed publishing it.

—*Chris Williams*
President, Syngress Publishing

FPO

Letters from the Authors

October 1998. It was a warm and sunny October and I remember it as if it was just yesterday. Giovanni, a colleague of mine, returns home to Italy from his honeymoon in New York. He carries in the office an enormous blue box whose cover reads "LEGO MINDSTORMS Robotics Invention System." When Giovanni opens the box and shows me the contents, I already know I must have one.

Let me go back to the late 70s. I was a high school student and had left my many years of LEGO play behind me. I was enthusiastically entering the rising personal computing era. Many of you are probably simply too young to remember that period, but "using" a computer mainly meant *programming* it. The computers of that time had few resources and rather primitive user interfaces; they were essentially mass storage devices, or something like a large unreliable cassette recorder. We programmers had to count and save every single byte, and even the most trivial tasks were very challenging. But at the same time, of course, it was great fun!

I developed a very strong interest in computer programming, and in Artificial Intelligence in particular. Machines and mechanical devices had always fascinated me, and it came quite naturally to me to turn to robotics as an expansion of this interest. There were some relatively cheap and compact computing devices that could provide a brain for my creatures, but unfortunately I discovered very soon all the technical problems involved in building the hardware of even a very simple robot. Where could I find motors? Which were the right ones? Where could I learn how to control them? What kind of gearing did I need? Imagine spending months folding aluminum plates, mounting bearings, assembling electronic circuits, connecting wires… and assuming you're able to do all those things, what do you get? A simple tin box that can run across the room and change direction when it hits an obstacle. The effort was definitely far greater than the results. Another problem was that constructing a new project meant starting again from scratch, with new materials. I wasn't patient enough, so I decided that a hobby in robotics was not for me.

The dream of robotics remained a dream. Until Giovanni opened *that* box. As soon as I got my hands on my first LEGO MINDSTORMS Robotics Invention System (RIS) set, it proved to be the fast and flexible robotics system that I was looking for. I found that the microcomputer, called the RCX, was very simple to use but powerful enough to let me drive complex devices. I became more and more intrigued by this toy, and through the Internet I soon discovered that I was not alone. It seemed an entire world of potential robotics fans had just been waiting for this product, and the LEGO company itself sold much more of them than expected.

From that October on, many things happened: I discovered LUGNET, the fantastic LEGO Users Group Network, the best resource ever for LEGO fans of any kind. I created a small Web site where I published pictures and information about my robotic creatures. Through these channels every day I got in touch with new people, and with some of them friendships have sprung up that go beyond our common interests in LEGO robotics. This is really the most special and valuable thing MINDSTORMS have given to me: Good friends all over the world.

—*Mario Ferrari*

October 1999. Another warm and sunny October, but on one particular day the Media Lab at the Massachusetts Institute of Technology (MIT) in Cambridge, MA has a different look. One large room at the facility is filled with exhibition tables with piles of colorful LEGO pieces and strange constructions on them and, there are hundreds of adults and children, LEGO bricks in hand, showing off their robotic creations and discussing the characteristics of their favorites. This is the world's biggest gathering of LEGO MINDSTORMS fans—the Mindfest!

When and how did all this start? It seems only yesterday to me, but a year had passed since I discovered MINDSTORMS for the first time. My brother Mario called me on the phone one evening, knowing I was about to leave on a short trip to New York, and asked me to bring him home a new product from LEGO, a sort of a programmable brick that could be controlled via a standard computer. I have to say that I was very curious, but nothing more: I thought it might be a great new toy to play around with, but I didn't completely understand its possibilities. When I saw the Robotics Invention System (RIS) in the toy store, though, I immediately realized how great it could be, and that I must have one, too. My own addiction to the LEGO MINDSTORMS began in that moment.

Like nearly everyone under the age of 40, I'd built projects from the many LEGO theme kits in my childhood. I had the advantage of using the large quantity of bricks that my older brothers and sisters had accumulated during the years, plus some new pieces and sets of the 80s. Castles, pirates, trains… hours and hours of pure fun, creating a large number of any kind of building and adventures. When I was a little older, I discovered the TECHNIC series, a wonderful world of machines, gears, mechanical tools, and vehicles, with endless construction possibilities. Then, like many other people, I abandoned LEGO as a young adult, and it remained out of my life—until I bought that big blue box in New York that day.

Why do I like LEGO MINDSTORMS so much? For me, it is mainly because it requires different skills and combines different disciplines: computer programming, robotics, and hands-on construction. You have to combine theory and practice, and to coordinate the design, construction, software, and testing processes. You can exercise your creativity and your imagination, and you have a great tool for doing this—a tool that is at the same time easy to use and very powerful, and most important, that doesn't limit your ideas.

And there's even more to the rewards of MINDSTORMS than that. Let's go back to Mindfest for a moment. Why would such an extraordinary group of people of different ages, cultures, and nationalities travel from all over the world to spend an entire weekend playing with LEGO? What exactly do they have in common? Why do some of the most famous Artificial Intelligence experts seriously discuss every feature of this product? There must be something really special about this "toy."

Joining an international community is one of the best things about playing with LEGO. It is not only a toy, but also a way of thinking and living. Just play with the MINDSTORMS for a while—you'll see for yourself!

—Giulio Ferrari

Author Acknowledgements

We would first like to thank Brian Bagnall for suggesting our names to Syngress Publishing when he heard Syngress was looking for an author to write a book focused on ideas and techniques for building MINDSTORMS robots. We are very grateful to Syngress Publishing for having turned this suggestion into a real opportunity, and for having allowed us the great freedom of deciding the shape and content of the book.

Additional appreciation goes to Jonathan Knudsen, who encouraged us in embarking upon the adventure of writing a book, and who helped us in understanding the world of technical publishing. Another friend, Guido Truffelli, patiently read every page of the manuscript. Many thanks, Guido—your comments and suggestions were very valuable in making the book more complete and more useful.

When Ralph Hempel accepted the offer to perform the technical edit of the book, we were really excited. Ralph's contributions to MINDSTORMS robotics are impressive, and range from mechanical solutions to extreme programming. His involvement proved to be even more significant that we had even imagined.

This was our first authoring experience, and all the Syngress staff has been incredibly patient with us and very supportive. A very special thank you goes to Kate Glennon, our Developmental Editor, for having taught us how to transform a collection of concepts and ideas into a book. Mario wants to also thank his employer, EDIS, which granted him the time to focus more attention on the book.

This book would have not been written without the contributions of the entire LUGNET MINDSTORMS Robotics community. Its members are incredibly creative, competent, helpful, and friendly, and they are always willing to share ideas and solutions with other people. We have attempted to give proper credit to all the people whose ideas we mentioned in the book, and we apologize in advance for those people who have been unintentionally left out.

Last but not least, we'd like to express enormous gratitude to our families, who encouraged and supported us through every moment of these intense months of writing.

Syngress Acknowledgements

We would like to acknowledge the following people for their kindness and support in making this book possible.

Richard Kristof and Duncan Anderson of Global Knowledge, for their generous access to the IT industry's best courses, instructors, and training facilities.

Karen Cross, Lance Tilford, Meaghan Cunningham, Kim Wylie, Harry Kirchner, Kevin Votel, Kent Anderson, and Frida Yara of Publishers Group West for sharing their incredible marketing experience and expertise.

Mary Ging, Caroline Hird, Simon Beale, Caroline Wheeler, Victoria Fuller, Jonathan Bunkell, and Klaus Beran of Harcourt International for making certain that our vision remains worldwide in scope.

Anneke Baeten and Annabel Dent of Harcourt Australia for all their help.

David Buckland, Wendi Wong, Daniel Loh, Marie Chieng, Lucy Chong, Leslie Lim, Audrey Gan, and Joseph Chan of Transquest Publishers for the enthusiasm with which they receive our books.

Kwon Sung June at Acorn Publishing for his support.

Ethan Atkin at Cranbury International for his help in expanding the Syngress program.

A special thanks to Sam Williams, who comes to the office every week with a backpack full of LEGOs. Watching the look on his face when he opens a new kit is a joyous event.

Contributors

Called the "DaVincis of LEGOs," Mario and Giulio Ferrari are world-renowned experts in the field of LEGO MINDSTORMS robotics.

Mario Ferrari received his first Lego box around 1964, when he was 4. Lego was his favorite toy for many years, until he thought he was too old to play with it. In 1998, the LEGO MINDSTORMS RIS set gave him reason to again have LEGO become his main addiction. Mario believes LEGO is the closest thing to the perfect toy and estimates he owns over 60,000 LEGO pieces. The advent of the MINDSTORMS product line represented for him the perfect opportunity to combine his interest in IT and robotics with his passion for LEGO bricks. Mario has been a very active member of the online MINDSTORMS community from the beginning and has pushed LEGO robotics to its limits. Mario is Managing Director at EDIS, a leader in finishing and packaging solutions and promotional packaging. He holds a bachelor's degree in Business Administration from the University of Turin and has always nourished a strong interest for physics, mathematics, and computer science. He is fluent in many programming languages and his background includes positions as an IT manager and as a project supervisor. Mario works in Modena, Italy, where he lives with his wife Anna and his children Sebastiano and Camilla.

Giulio Ferrari is a student in Economics at the University of Modena and Reggio Emilia, where he also studied Engineering. He is fond of computers and has developed utilities, entertainment software, and Web applications for several companies. Giulio discovered robotics in 1998, with the arrival of MINDSTORMS, and held an important place in the creation of the Italian LEGO community. He shares a love for LEGO bricks with his oldest brother Mario, and a strong curiosity for the physical and mathematical sciences. Giulio also has a collection of 1200 dice, including odd-faced dice and game dice. He studies, works, and lives in Modena, Italy.

Technical Editor

Ralph Hempel (BASc.EE, P.Eng) is an Independent Embedded Systems Consultant. He provides systems design services, training, and programming to clients across North America. His specialty is in deeply embedded microcontroller applications, which include alarm systems, automotive controls, and the LEGO RCX system. Ralph provides training and mentoring for software development teams that are new to embedded systems and need an in-depth review of the unique requirements of this type of programming. Ralph holds a degree in Electrical Engineering from the University of Waterloo and is a member of the Ontario Society of Professional Engineers. He lives in Owen Sound, Ontario with his family, Christine, Owen, Eric, and Graham.

Contents

Foreword xxv

Preface xxvii

Part I Tools 1

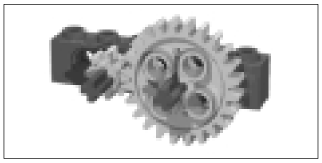
Chapter 1 Understanding LEGO® Geometry 3

Introduction 4
Expressing Sizes and Units 4
Squaring the LEGO World: Vertical Bracing 6
Tilting the LEGO World: Diagonal Bracing 10
Expressing Horizontal Sizes and Units 12
Bracing with Hinges 14
Summary 15

Chapter 2 Playing with Gears 17

Introduction 18
Counting Teeth 18
Gearing Up and Down 19
Riding That Train: The Geartrain 21
Worming Your Way: The Worm Gear 23
Limiting Strength with the Clutch Gear 24
Placing and Fitting Gears 26
Using Pulleys, Belts, and Chains 31
Pulleys and Belts 32
Chains 35
Making a Difference: The Differential 36
Summary 39

Learn about Lego Gears



Chapter 3 Controlling Motors **41**
Introduction 42
Pacing, Trotting, and Galloping 43
Mounting Motors 46
Wiring Motors 49
Controlling Power 52
Braking the Motor 53
Coupling Motors 55
Summary 56

Chapter 4 Reading Sensors **57**
Introduction 58
Touch Sensor 58
Light Sensor 64
Measuring Reflected Light 66
Line Following 67
Proximity Detection 70
Rotation Sensor 71
Temperature Sensor 74
Sensor Tips and Tricks 75
Emulating a Touch Sensor 75
Emulating a Rotation Sensor 75
Connecting Multiple Sensors to the Same Port 78
Other Sensors 80
Summary 81

Chapter 5 Building Strategies **83**
Introduction 84
Locking Layers 84
Maximizing Modularity 87
Loading the Structure 88
Putting It All Together: Chassis, Modularity, and Load 90
Summary 94

Explore LEGO Sensors

LEGO sensors come in two families: *active* and *passive* sensors. Passive simply means they don't require any electric supply to work. The touch and temperature sensors belong to the passive class, while the light and rotation sensors are members of the active class.

Chapter 6 Programming the RCX 97
Introduction 98
What Is the RCX? 98
A Small Family of Programmable Bricks 100
Using LEGO RCX Code 101
Using the NQC Language 102
Using Other Programming Languages 103
Using legOS 104
Using pbForth 104
Using leJOS 105
Using Other Programming Tools and Environments 105
Divide and Conquer: Keeping Your Code Organized 106
Running Independent Tasks 113
Summary 114

Understand the Benefits of Designing Modular Code

- Readability
- Reusability
- Testability

Chapter 7 Playing Sounds and Music 117
Introduction 118
Communicating through Tones 118
Playing Music 119
Converting MIDI files 122
Converting WAV Files 124
Summary 124
Chapter 8 Becoming Mobile 127
Introduction 128
Building a Simple Differential Drive 128
Keeping a Straight Path 130
Using Sensors to Go Straight 130
Using Gears to Go Straight 131
Using Casters to Go Straight 134
Building a Dual Differential Drive 136
Building a Skid-Steer Drive 138
Building a Steering Drive 140
Building a Tricycle Drive 144
Building a Synchro Drive 146

Other Configurations 150
Summary 151

Chapter 9 Expanding Your Options with Kits and Creative Solutions 153
Introduction 154
Acquiring More Parts 154
Introducing Some Specialized Components 155
Buying Additional Parts 161
Creating Custom Components 163
Building Custom Sensors 164
Solving Port Limitations 170
Creative Solutions When More RCX Ports Are Needed 172
Summary 177

Chapter 10 Getting Pumped: Pneumatics 179
Introduction 180
Recalling Some Basic Science 180
Pumps and Cylinders 181
Controlling the Airflow 187
Building Air Compressors 190
Building a Pneumatic Engine 194
Summary 198

Chapter 11 Finding and Grabbing Objects 199
Introduction 200
Operating Hands and Grabbers 200
Transferring Motion Using Tubing 203
Understanding Degrees of Freedom 205
Finding Objects 209
Summary 212

Chapter 12 Doing the Math 213
Introduction 214
Multiplying and Dividing 215
Averaging Data 218
Simple Averages 218
Weighted Averages 223

Create Custom Components

Explore extra parts, custom sensors, and tricks for using the same motor for more than one task:

- Extra parts come from either sets or service packs.
- Custom sensors are a new frontier, and reveal a whole new world of possibilities.
- Mechanical tricks enable you to use the same motor to power multiple mechanisms.

Using Interpolation 226
Understanding Hysteresis 229
Summary 232

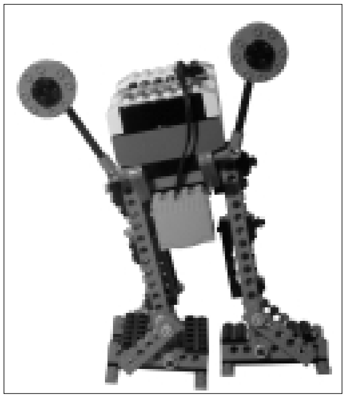
Chapter 13 Knowing Where You Are 233
Introduction 234
Choosing Internal or External Guidance 234
Looking for Landmarks: Absolute Positioning 236
Following the Beam 239
Measuring Movement: Relative Positioning 243
Summary 246

Part II Projects 247

Chapter 14 Classic Projects 249
Introduction 250
Exploring Your Room 250
Detecting Edges 253
Variations on Obstacle Detection 255
Following a Line 256
Further Optimization of Line Following 264
Modeling Cars 265
Front-Wheel and Four-Wheel Drives 270
Switching Gears 274
Using the Gear Switch 276
Summary 277

Chapter 15 Building Robots That Walk 279
Introduction 280
The Theory behind Walking 280
Building Legs 287
Building a Four-Legged Robot 289
Building a Six-Legged Steering Robot 293
Designing Bipeds 299
Interlacing Legs 299
COG Shifting 302
Ankle Bending 306
Making Bipeds Turn 309
Summary 310

Use Ankle Bending Techniques



Chapter 16 Unconventional Vehicles 311
Introduction 312
Creating Your Own SHRIMP 312
Building a SHRIMP 320
Creating a Skier 321
Creating Other Vehicles 328
Elevator 328
Train 328
Cable Railway or Gondola 329
Boat 329
Sailing Tricycle 330
Summary 331

Chapter 17 Robotic Animals 333
Introduction 334
Creating a Mouse 334
Improvements Upon the Mouse's Construction 338
Creating a Turtle 339
Improvements Upon the Turtle's Construction 344
Creating Other Animals 345
Summary 346

Chapter 18 Replicating Renowned Droids 349
Introduction 350
Building an R2-D2-Style Droid 350
Programming the Droid 356
Variations on the Construction 359
Building a Johnny Five-Style Droid 361
Variations on the Construction 366
Summary 368

Chapter 19 Solving a Maze 371
Introduction 372
Finding the Way Out 373
Using the Left Side—Right Side Strategy 374
Applying Other Strategies 376

Use Angle Connectors

There are currently six types of angle connectors in the LEGO line, numbered 1 to 6. In case you're wondering how the numbers relate to angles, here are the correspondences: 1 = 0°, 2 = 180°, 3 = 157.5°, 4 = 135°, 5 = 112.5°, 6 = 90°. They go by increments of 22.5°, a quarter of a right angle.

Building a Maze Runner 377
  Constructing the Maze Runner 377
  Programming the Runner 380
  Creating the Maze 381
  Variations on the Maze Runner 381
Building a Maze Solver 382
  Constructing the Maze Solver 383
  Programming the Solver 387
Summary 389

Chapter 20 Board Games 391
Introduction 392
Playing Tic-Tac-Toe 392
  Building the Hardware 393
  Writing the Program 394
  Improving Your Game 396
Playing Chess 396
  Building a Visual Interface 397
  Building a Mechanical Interface 400
    Connecting and Programming Broad Blue 406
  Variations on the Construction 408
Playing Other Board Games 409
Summary 410

Chapter 21 Playing Musical Instruments 411
Introduction 412
Creating a Drummer 412
  Building the Drummer 413
  Programming the Drummer 416
  Variations 417
Creating a Pianist 417
  Building the Pianist 417
  Programming the Pianist 422
  Changes and Improvements 423
Other Suggestions 424
Summary 424

Build a Pianist

This robot requires a lot of extra parts, mainly beams and plates used to make the structure solid enough to withstand the forces involved in the performance.

Chapter 22 Electronic Games 425
Introduction 426
Creating a Pinball Machine 426
Building the Pinball Machine 426
Programming the Pinball Machine 431
Improvements on the Construction 432
Creating a Simon-Says Game 432
Building a Simon-Says Game 432
Programming the Simon-Says Game 434
Variations 438
Other Electronic Game Suggestions 438
Summary 439

Chapter 23 Drawing and Writing 441
Introduction 442
Creating a Logo Turtle 442
Building the Turtle 443
Programming the Turtle 448
Choosing the Proper Language 453
Variations 454
Tape Writer 455
Building the Writer 455
Programming the Writer 462
What to Write 462
Variations 463
Further Suggestions 464
Copying 464
Emulating Handwriting 464
Learning by Example 465
Summary 465

Chapter 24 Simulating Flight 467
Introduction 468
Introducing the Forces Involved in Flight 468
Designing the Simulator Project 471
Building the Hardware 473
Programming the Simulator 480

Understand Infrared Communication

Infrared (IR) light is of the same nature as visible light, but its frequency is below that perceivable by the human eye. Provided the intensity is high enough, we usually feel IR radiation as heat.

Operating the Simulator 488
Downsizing the Project 489
Upsizing the Project 491
Summary 492

Chapter 25 Constructing Useful Stuff 493
Introduction 494
Building a Floor Sweeper 494
Constructing the Sweeper 494
Programming the Sweeper 497
Improvements on the Floor Sweeper 497
Building a Milk Guard 498
Making the Milk Guard 499
Programming and Using the Milk Guard 501
Improvements on the Milk Guard 502
Building a Plant Sprinkler 502
Making the Sprinkler 503
Programming and Using the Sprinkler 504
Improvements on the Plant Sprinkler 505
Designing Other Useful Robots 508
Summary 509

Part III Contests 511

Chapter 26 Racing Against Time 513
Introduction 514
Hosting and Participating in Contests 514
Optimizing Speed 516
Drag Racing 516
Combining Speed with Precision 518
Line Following 519
Wall Following 520
Other Races 522
Summary 523

Chapter 27 Hand-to-Hand Combat 525
Introduction 526
Building a Robotic Sumo 526
Setting the Rules 527

Design Other Useful Robots

- Alarm Clock
- Baby Entertainer
- Pet Feeder
- Dog Trainer

Maximizing Strength and Traction 528
Attack Strategies 530
Finding the Enemy 530
Using Speed 532
Using a Transmission 533
Other Sumo Tricks 533
Getting Defensive 534
Testing Your Sumo 535
Summary 536

Chapter 28 Searching for Precision 537
Introduction 538
Precise Positioning 538
Finding and Collecting Things 540
Maxwell's Demons 540
Stealing the Cube 541
Variations on Collecting 543
Playing Soccer 543
Summary 545
Chapter 28 Searching for Precision 537
Introduction 538
Precise Positioning 538
Finding and Collecting Things 540
Maxwell's Demons 540
Stealing the Cube 541
Variations on Collecting 543
Playing Soccer 543

Summary 545
Appendix A Resources 547
Introduction 548
Bibliography 548
General Interest Sites 549
Chapter 1 Understanding LEGO Geometry 551
Chapter 2 Playing with Gears 551
Chapter 3 Controlling Motors 551
Chapter 4 Reading Sensors 552

Find Useful Lego Sites

- www.brickshelf.com
- http://fredm.www.media.mit.edu/people/fredm/mindstorms/index.html
- www.crynwr.com/lego-robotics/
- www.bvandam.net

Chapter 5 Building Strategies 552
Chapter 6 Programming the RCX 553
Chapter 7 Playing Sounds and Music 556
Chapter 8 Becoming Mobile 556
Chapter 9 Expanding Your Options with Kits and Creative Solutions 557
Chapter 10 Getting Pumped: Pneumatics 559
Chapter 11 Finding and Grabbing Objects 559
Chapter 12 Doing the Math 560
Chapter 13 Knowing Where You Are 560
Chapter 14 Classic Projects 561
Chapter 15 Building Robots That Walk 562
Chapter 16 Unconventional Vehicles 562
Chapter 17 Robotic Animals 563
Chapter 18 Replicating Renowned Droids 563
Chapter 19 Solving a Maze 564
Chapter 20 Board Games 564
Chapter 21 Playing Musical Instruments 565
Chapter 22 Electronic Games 565
Chapter 23 Drawing and Writing 565
Chapter 24 Simulating Flight 566
Chapter 25 Building Useful Stuff 566
Chapter 26 Racing Against Time 566
Chapter 27 Hand-to-Hand Combat 567
Chapter 28 Searching for Precision 567

Appendix B Matching Distances 569

Appendix C Note Frequencies 575

Appendix D Math Cheat Sheet 577

Sensors 578
Averages 578
Interpolation 578
Gears, Wheels, and Navigation 579

Index 581

Foreword

Like many other programmers, I credit my early years of playing with LEGO as a major factor in my future career path. As my family and I watched the United States launching the Apollo 11 rocket, I was playing with a LEGO truck—it was my birthday and I was 7 years old. What I could not know at the time was that 30 years later I would hold in the palm of my hand a microcontroller with more raw speed and memory than the one the astronauts used to get to the moon and back. That computer would be encased in yellow ABS plastic and would change the world of hobby and educational robotics.

The story of my involvement with the LEGO MINDSTORMS is a familiar one. Discussion of building a custom controller for LEGO TECHNIC creations was a frequent topic in Lugnet (the LEGO Users Group) discussion forums. I had doubts about our ability to make a controller that everyone could afford. Then LEGO released MINDSTORMS in the fall of 1988—and I just had to have one.

Within weeks of the release, Kekoa Proudfoot had "cracked" the protocol between the RCX brick and the desktop computer, and he soon had a complete disassembly of the object code online. Using this as a base, intrepid programmers like Marcus Noga and Dave Baum soon had alternative programming environments for the RCX—including my own contribution, called pbForth. On the hardware front, Michael Gasperi figured out how the sensor and motor ports worked and contributed his knowledge freely.

LEGO had an unbelievable hit on their hands. The sales of the MINDSTORMS kits exceeded their wildest predictions, and more than half the sales were to adults! When the Massachusetts Institute of Technology (MIT) asked me to participate in a panel at the Mindfest gathering in 1999, I was honored to be there with the likes of Dave Baum, Michael Gasperi, Marcus Noga, and Kekoa Proudfoot. In our panel discussion, we discussed how the Internet had made it possible for widely separated people to work together.

While at Mindfest, I met Mario and Giulio Ferrari. They had their Tic Tac Toe robot set up for demonstrations and it was a big hit. The brothers immediately struck me as energetic and dedicated LEGO hobbyists. The other members of the Italian ITLug group have provided LUGNET readers a steady stream of wonderful robots in the past few years.

I have had the pleasure of watching children and adults of all ages build machines and robots with their MINDSTORMS kits. In almost all cases their initial attempts ended in frustration with their *mechanical* skills. In fact, many builders never even get to the stage of programming their robots. This book will be a welcome addition to their libraries because of the vast amount of information it contains. From basic bracing techniques to drive and grip mechanisms—it's all here. Even if a particular robot does not appeal to the reader, the ideas used in its construction may be transferred to other robots in unusual and surprising ways.

As a co-author of *Extreme Mindstorms*, a book about programming the RCX, I appreciate the effort that went into this book. Mario and Giulio have taken the time to guide the reader through the basics of building their creations by setting realistic performance goals and then experimenting with different methods. This important skill goes by the unassuming name of *tinkering*, and cannot be underestimated. The MINDSTORMS system gives the hobby and educational market a modular and inexpensive way to develop these important tinkering skills.

As the technical editor of this volume, I have had my own creativity sparked by some of the robots Mario and Giulio have documented. I am amazed at the sheer volume of ideas, the quality of the photos, and the careful presentation of ideas that many readers will encounter for the first time. The staff at Syngress Publishing has been a pleasure to work with, and they deserve credit for bringing the hard work of the Ferrari brothers to the wide audience that I'm sure this book will enjoy.

So clear some space on a table, open this book and get out your MINDSTORMS set, and start tinkering!

—Ralph Hempel

Preface

Why Robotics?

What's so special about robotics? Why have LEGO MINDSTORMS experienced such great success? Each one of us might have our own answers.

Robotics is an interdisciplinary subject, combining different fields of study that in traditional educational systems you usually examine separately: physics, mathematics, electronics, and computer programming, just to name a few. Robotics is a hobby through which you can find a practical application for many of the concepts you studied in school—or, if you didn't study them, or don't have an aptitude for them, it offers a great way to learn by experience and by having some fun. The most important point, however, is that robotics is *more* than the sum of the basic notions you're required to know. It gives you a precise and concrete idea of how these notions integrate and complete each other. So it happens that when you're looking for a solution to a problem, by following your intuition and knowledge it's almost a given that you'll find a solution different from that devised by someone else.

Let's say you have just built your first line-following robot (we'll discuss this topic in detail in Part II). You discover that your robot works, but it makes too many corrections to its steering and this affects its resulting speed. What could you do to fix it? If you have a talent for mechanics, your first approach might be to try and modify the structure and architecture of your robot. You might observe that the wheels are too close to each other in your differential drive, and for this reason your robot turns very fast and tends to over-correct its steering. Or you might decide that the differential drive architecture after all is not the best option for line following. You may even discover that the position of the light sensor in the robot greatly affects its performance.

If you are an experienced programmer, you might instead work out your code to correct the robot's behavior. You feel at ease with timers and counters, so you change the program to introduce some delay in the route changes, then you spend some in time in testing and trimming it until you find an optimum value for the constants you used.

At the same time, if you have a decent understanding of physics, you could reach into your knowledge base for something useful, and discover a model you were taught when studying magnetism: hysteresis (if you don't know what hysteresis is, don't worry, we'll explain it in Part I!). You realize that you can make your robot follow a different scheme when going from black to white rather then from white to black. You think that this might improve its performance—and it actually does.

What lesson should we learn from this example? That there's no one unique solution, there are many of them. And the more you are able to open your mind and explore new possible approaches, the higher your chances of working out a solution. Robotics does not involve a list of techniques to follow in order, rather it is a process in which your creativity plays a very strong role, allowing you to follow a new path to the goal each time.

There's another element that makes robotics so interesting to us and, I suspect, to many other people as well. It forces you to look at the world with new eyes, those of a child's.

If you observe babies exploring the environment, you will notice that they are surprised by everything. They don't take anything for granted. They try everything, continuously developing new concepts by testing new approaches. We adults usually laugh at most of those attempts, to our mature minds they seem absurd, either because we already know that a specific thing is impossible to do, or because we know the solution to the problem the child is tackling. When approaching tasks in robotics, we are forced to become children again, to rediscover the world with different senses.

Let's look at this concept using another example: You are new to robotics, facing your first project, but are wise enough to decide on a very simple task. You want to create a robot that's able to move around your house. You naturally want your robot to be able to detect obstacles when it hits them, so it can change direction and toddle off on a new path. You design your mechanical marvel so it can go forward, backward, and change direction. Then you add a simple bumper to detect obstacles, something that closes a switch when pressed. Finally, you write some code so your robot is ready for its debut on the living room floor—but wait, you forgot about the shag carpet, and carpet loops get into your gears and mess everything up. You decide testing might be better in the kitchen. Now your robot runs well; it hits a wall, turns on itself, and spins off in another direction. Up to this point, it's a pleasure to watch…but then it runs up against a sideboard, and the upper part of the robot gets blocked by the furniture, preventing the lower bumper from detecting the obstacle. Okay, so you have to improve the bumper. In the meantime, you break down and manually turn the robot in a new direction. Hey! Pay attention! It's heading to the basement stairs! Rescue it and add edge detection to your list of improvements. You will learn quickly that even a simple action like climbing the stairs is the result of a very, very complex balancing of weights and strengths, precise positioning, and coordination.

If you have kids in your circle of family and friends, you will have the precious opportunity to watch how they interact with your robots. In our experience, young kids tend to expect a lot from robots—a lot more than what simple inexpensive robots can currently do. They have forgotten all the difficulties they had to overcome themselves, and they're still naïve enough to believe that all the amazing things robots do in movies can be carried out by your robots as well. They see any possible task or function as easy to implement. "Why don't you make one like the *real* R2-D2, Dad?"

MINDSTORMS provides a great way for kids to understand that even the most common activities are composed of many individual operations. If they don't understand, if they become frustrated by what the robots *can't* do, play an easy but funny game with them in which *you* are the robot and they have to "program" you using only a very simple vocabulary describing a few basic actions. They will laugh at all the stupid things you'll do and the unusual situations their commands will get you into—but they will very likely understand the point. This is an extra gift that robotics will provide to your family: showing your children how to deconstruct and analyze what they consider a single action.

When you're really at a loss for what robot you might build next, ask the kids! You're sure to get a bunch of fresh ideas. Most of us tend to design robots that move around, grab objects, find soda cans in a room or do any other activity we *expect* robots should do. Some of these projects are very challenging, and most are very instructive. But if you ask the kids what they would like to see, you get responses like: "Why don't we build a *skiing* robot, Dad?" Would you ever think of a skiing robot? Just the same, robots of this type are easy to make (see Chapter 16). They require only basic parts, they're fun, and like any MINDSTORMS challenge, they're definitely worth the time you spend on them.

Why LEGO?

If you've been raised with LEGO like we have, you already know what's special about it. But for those relatively new to the LEGO concept, including those who have yet to buy a MINDSTORMS set, let us explain why LEGO is an excellent choice for exploring the world of robotics.

The power of the LEGO system lies in its founding concept: reusability. The same basic brick can today be the foot of an elephant, tomorrow a block in an Egyptian pyramid, and the day after the nose of a robot. When you open a LEGO box, you see the parts that will form a LEGO model, but you also see an infinite number of possible models you might create with those parts.

The property that transforms these small plastic pieces into a construction system is their connectivity. You don't need glue, screws, or any special tools (other then your hands) to assemble (or dismantle) a LEGO model. The LEGO parts easily snap on to each other and stay firmly in place until you decide to take them apart. The parts won't be damaged, no matter how much you use them.

But what really makes LEGO easy to use is its modularity. Not only does one brick connect to another, but they do so at predefined, discrete positions. There are studs and holes that force you to assemble parts following a precise geometric scheme. This might seem a limitation at first, but it actually makes your life easier because of something called precise positioning. You don't need a ruler or a square—all that's required is that you can count!

- **LEGO is fast** You don't have to saw, cut, drill, solder, fold, file, or mill your components. They are ready to use, just pick up what you need from the box.
- **LEGO is clean** You don't produce filings, don't need any lubricants or paints, and when you have finished playing with it, your room looks exactly as it did before. This is a very important point to make to the people who live with you if you want them to be tolerant of your hobby!
- **LEGO is cost-effective** You can use and reuse your LEGO parts as needed to produce many generations of robots. And should you ever eventually tire of your LEGO pieces, they will still have a market value. There are other easy-to-assemble robotic kits on the market, but they usually only permit you to build one specific model. Beyond that, there's nothing more you can get from their kits.
- **LEGO is ecological** We don't mean that its ABS plastic is easy to recycle. It is, but that's not the point. You simply don't need to recycle it, because you'll never throw it away. After all, this is the most respectful approach to the environment: making products with a long life span, that don't exhaust their function and don't require recycling or disposal. We still use many of the LEGO bricks we received during childhood.

To return to robotics, some of you may believe that LEGO MINDSTORMS is too limited a system to build sophisticated projects. This is true if you mean *really* sophisticated systems! Others may observe that LEGO is not suitable for building robots that perform actual work. This, again, can be true, although we will show some examples in Chapter 25 that are indeed useful work projects. The LEGO MINDSTORMS kit is definitely more than a toy—it is probably the most fun and effective educational tool for learning the scientific principles behind robotics. There are indeed limitations, but this is part of the fun, challenging you to use your imagination, to find esoteric solutions for seemingly unsolvable problems.

Suppose you're an experienced programmer, asked to write the umpteenth version of an invoicing software—just the thought of it puts you to sleep. But then your employer adds "Oh, by the way, it has to run on a machine with 3 K of RAM. *Now* you're interested! After all, there's nothing like a challenge.

So, don't feel limited by the constraints you find in the system, feel inspired. Create a robot that makes your friends say "I didn't think it was possible to make such an incredible thing with LEGO!" Because you can.

Using this Book

This book is about building robots using LEGO bricks and components. The chapters in Part I are about *how* to build a robot. Here, we provide a set of *tools* you'll need to explore the world of robotics. We'll review basic knowledge about mechanics, motors, sensors, pneumatics, and navigation. We will compare different standard architectures, discuss solutions to common recurring problems, and will suggest how to organize complex projects in terms of subsystems.

Part II will face the tough question, "I've got my MINDSTORMS kit, I've learned how to use it—so *what* do I build?" Here we will show you a large survey of possible ideas, but do not expect to find complete models to build step by step. The goal of this book is *not* to teach you to re-create our models, instead it is to stimulate your imagination to create your own. Imagination and creativity cannot be *taught*, but it can be inspired. We hope that our approach might help you see the world with different eyes. The same is true for understanding the mechanics of robotics: you will learn best by guided and informed experimentation. Actively participating in the process, not simply cloning our models, will bring you the greatest rewards.

Part III takes you into the world of robotics contests. These contests offer LEGO builders a challenge beyond the initial goal of building a working robot—they provide a means to inspire ideas, share solutions, and just have fun, whether with your own friends, in a local group, or even internationally. There are different ways to attend a robotic contest: you can compare robots with friends in person, or you can take up a challenge someone has organized through the Internet, in which case you submit your solution in pictures or programming code. Either way, you will learn a great deal from your opponents. And from the rules, too: what really makes a contest exciting is trying to find an original but "legal" solution you hope your opponents haven't thought of.

The last part of the book consists of appendices that provide you with various technical resources we hope will be helpful to you.

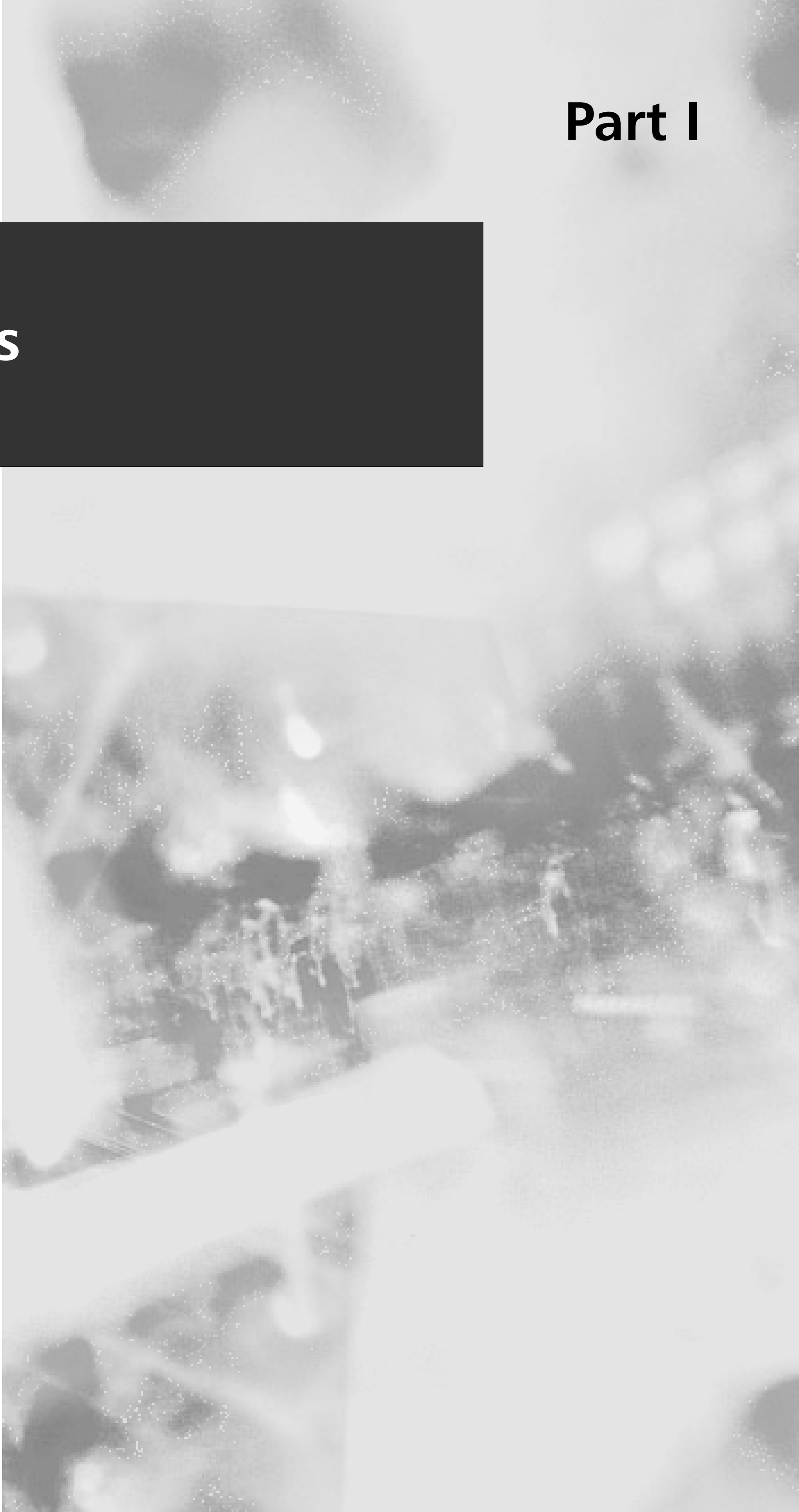
There is a key element to robotics that you will *not* find in this book: comprehensive coverage of programming and electronics. We made a conscious choice to focus this book on construction solutions and to cover only as much programming as was necessary—a limited amount of coverage is indeed required, because you cannot successfully design and build your robots without taking into consideration the role that electronics will play. Because there are various programming options you can choose from, depending on your level of programming experience, we have written our code using NQC, a very widespread C-like textual language that you can easily translate into your favorite language.

One of the nicest things about MINDSTORMS robots is that you're not required to be an electrical engineer to design them—we're not! If you are interested in expanding your RCX possibilities on that side, we will point you to the right resources.

Please note that we don't expect you read the book sequentially from cover to cover: feel free to jump to a specific page or topic. When we cite a concept or technique explained in a previous chapter, we'll tell you where to find it. The only things we expect from you are the following:

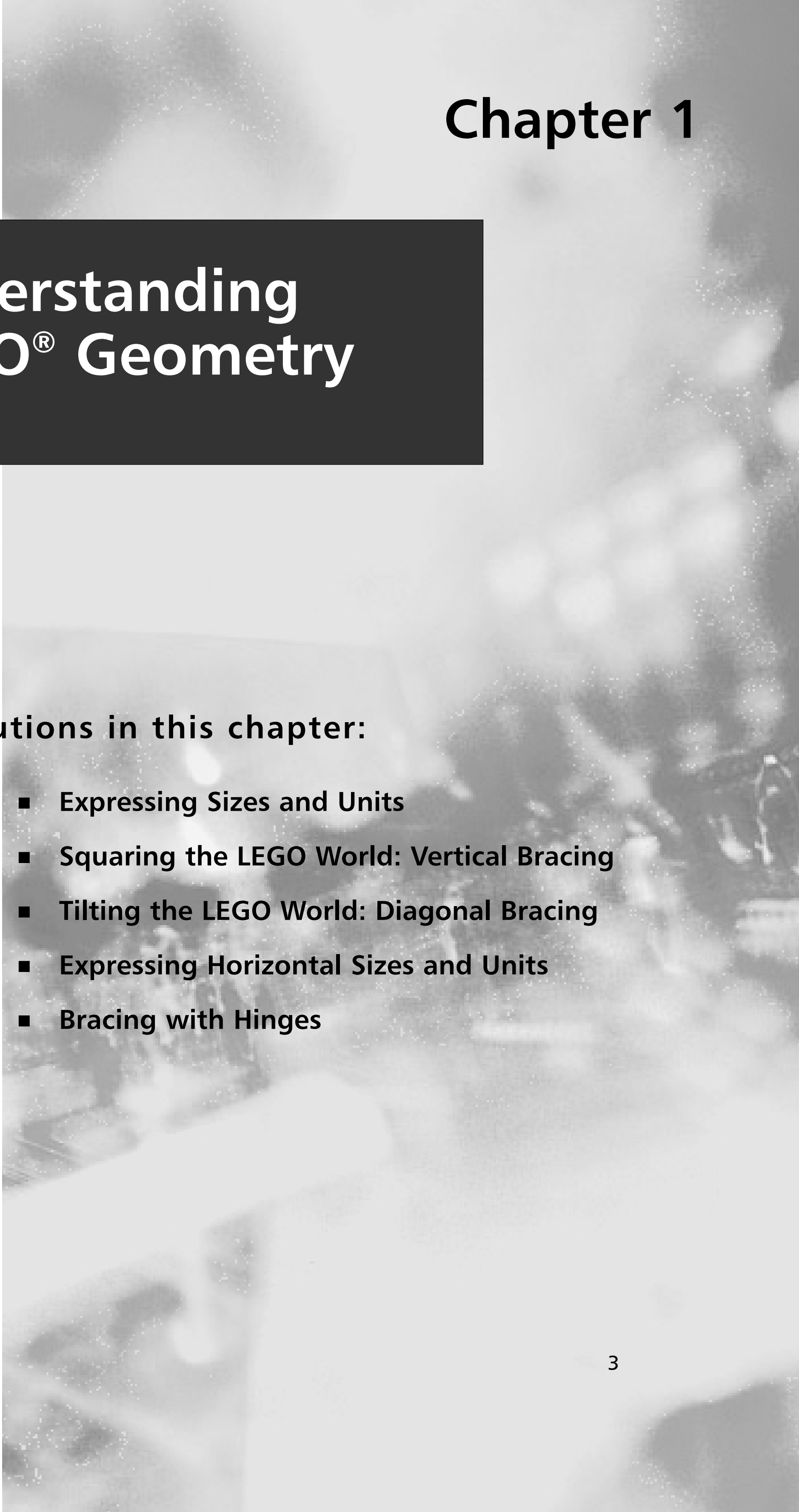
- That you own a MINDSTORMS Robotic Invention System kit or you are seriously interested in buying one. Many of the tips and ideas are, however, applicable to other LEGO programmable bricks (such as Scout and Cybermaster) or to nonrobotic LEGO TECHNIC models.
- That you already have some basic skill in assembling LEGO TECHNIC parts and in programming your RCX. Doing the lessons included in the MINDSTORMS CD-ROM and being familiar with the Constructopedia will be all the background you need.

Enjoy our book!



Part I

Tools



Chapter 1

Understanding LEGO® Geometry

Solutions in this chapter:

- Expressing Sizes and Units
- Squaring the LEGO World: Vertical Bracing
- Tilting the LEGO World: Diagonal Bracing
- Expressing Horizontal Sizes and Units
- Bracing with Hinges

Introduction

Before you enter the world of LEGO® robotics, there are some basic geometric properties of the LEGO bricks we want to be sure you know and understand. Don't worry, we're not going to test you with complex equations or trigonometry, we'll just discuss some very simple concepts and explain some terminology that will make assembling actual systems easier from the very beginning.

You will discover which units LEGO builders use to express sizes, what the proportions of the bricks are, and how this affects the way you can combine bricks with different orientations into a solid structure.

We encourage you to try and reproduce all the examples we show in this chapter with your own LEGO parts. Keep your MINDSTORMS box handy so you can pick up the parts you need, which in this chapter will actually be nothing more than a few bricks and plates.

If, for any reason, you feel the stuff here is too complex or boring, don't force yourself to read it, skip the chapter and go to another one. You can always come back and use this chapter as a sort of glossary whenever it's needed.

Expressing Sizes and Units

LEGO builders usually express the size of LEGO parts with three numbers: *width*, *length*, and *height*, in that order. The standard way to use LEGO bricks is "studs up." When expressing sizes, we always refer to this orientation, even when we are using the bricks upside down or when rotating them in 3-D space.

Height is the simplest property to identify, its the vertical distance between the top and bottom of the basic brick. Width, by convention, is the shorter of the two dimensions that lie on the horizontal plane (length is the other one). Both width and length are expressed in terms of *studs*, also called *LEGO units*. Knowing this, we can describe the measurements of the most traditional brick, the one whose first appearance dates back to 1949, which is 2 x 4 x 1 (see Figure 1.1).

LEGO bricks, although their measurements are not expressed as such, are based on the metric system—a stud's width corresponds to 8mm and the height of a brick (minus the stud) to 9.6mm. These figures are not important to remember—what's important is that they do not have equal values, meaning you need two different units to refer to length and height. Their *ratio* is even more important: dividing 9.6 by 8 you get 1.2 (the vertical unit corresponds to 1.2 times the horizontal one). This ratio is easier to remember if stated as a proportion between whole numbers: It is equivalent to 6:5. We will explore the relevance of this ratio in the next section.

Figure 1.1 Measurements of a Traditional LEGO Brick

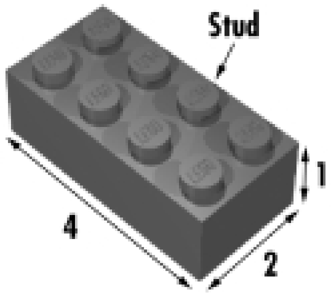
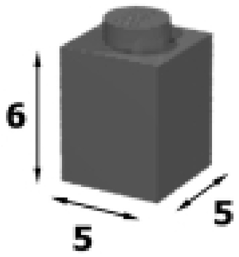


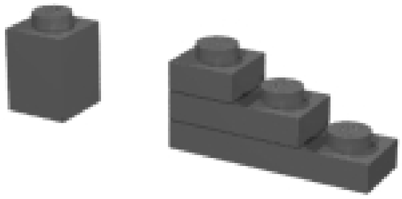
Figure 1.2 shows the smallest LEGO brick, described in LEGO units as a 1 x 1 x 1. For the reasons explained previously this LEGO "cube" is not a cube at all.

Figure 1.2 Proportions in a 1 x 1 x 1 LEGO Brick



The LEGO system includes a class of components whose height is one-third of a brick. The most important element of this class is the *plate*, which comes in a huge variety of rectangular sizes and in some special shapes, too. If you stack three plates, you get the height of a standard brick (see Figure 1.3).

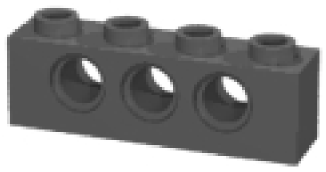
Figure 1.3 Three Plates Make One Brick in Height



Squaring the LEGO World: Vertical Bracing

Why do we care about all these relationships? To answer this, we must travel back to the late seventies when the LEGO TECHNIC line was created. Up to that time, LEGO was designed and used to build things made of horizontal layers: Bricks and plates integrate pretty well when stacked together. Every child soon learns that three plates count for a brick, and this is all they need to know. But in 1977, LEGO decided to introduce a new line of products targeting an older audience: LEGO TECHNIC. They gave the common 1xN brick holes and turned it into what we call a TECHNIC brick, or a *beam* (Figure 1.4). These holes allow *axles* to pass through them, and also permit the beams to be connected to each other via *pegs*, thus creating an entire new world of possibilities.

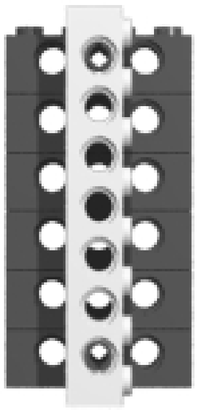
Figure 1.4 The LEGO TECHNIC Beam



Suppose you want to mount a beam in a vertical position, to brace two or more layers of horizontal beams. Here's where we must remember the 6 to 5 ratio. The holes inside a beam are spaced at exactly the same distance as the studs, but are shifted over by half a stud. So, when we stand the beams up, the holes follow the horizontal units and not the vertical ones. Consequently, they don't match the corresponding holes of the layered beams. In other words, the holes in the vertical beam cannot line up with the holes in the stack because of the 6:5 ratio. At least not with all the holes. But let's take a closer look at what happens. Count the vertical units by multiples of 6 (6, 12, 18, 24, 30…) and the horizontal ones by multiples of 5 (5, 10, 15, 20, 25, 30…). Don't count the starting brick and the starting hole, they are your reference point; you are measuring the *distances* from that point. You see? After counting 5 vertical units you reach 30, which is the same number you reach after counting 6 horizontal units (see Figure 1.5).

Is there a general rule we can derive from this? A sort of theorem? Yes: *In a stack of horizontal beams, at every fifth beam the holes align to those of a perpendicular beam.*

Figure 1.5 Matching Horizontal and Vertical Beams



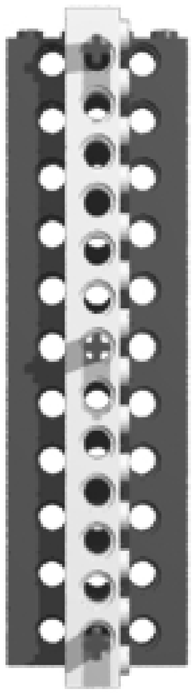
Now you can build a stack with some of your beams, brace them with another long one, and verify this rule in practice. If you put an axle in the first connecting hole and then try to put it again in the following holes, you'll find that the holes of the crossed beam match at the starting brick plus five and at the starting brick plus ten (see Figure 1.6).

This technique of crossing beams is extremely important. It's what enables us to build solid models, because the vertical beam locks all the beams in between the two horizontal beams. It's a pity we need to stack 6 beams before we can lock them with a traverse beam. Couldn't we build something more compact? The answer is, of course, yes.

Recall that the vertical unit has a subunit, the height of a plate. Three plates make a brick, so counting plates, we can increase the height by steps of 2 instead of 6 (2 is one-third of 6). Our progression in height now becomes: 2, 4, 6, 8, 10… after 5 vertical increments we reach the value 10. That's also in the horizontal scale of values, a spot where we know the holes will match. So our new and final theorem is: *every 5 plates in height, the holes of perpendicular beams match.* If there's a single thing you should remember from this chapter, this is it.

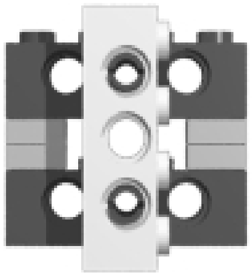
Unfortunately a plate cannot be used *as is* to connect a vertical beam, for the simple reason it hasn't any holes! But a beam is equivalent to three plates in height. Knowing this, we can state our rule in operational terms: Starting from the beam at the bottom (don't count it), add 1 for each plate and 3 for each beam, and keep at least a beam at the top. If the result is a multiple of 5, the holes can be matched by a perpendicular beam.

Figure 1.6 Every Five Bricks in Height the Holes Match



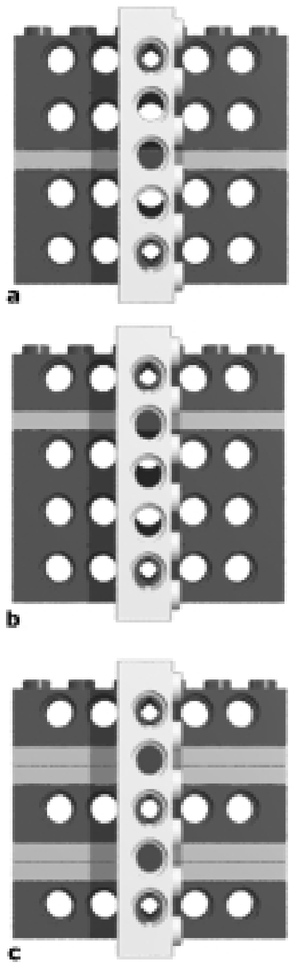
The most compact scheme that allows you to lock your horizontal layers with a vertical beam is the one shown in Figure 1.7: a beam and two plates, corresponding to five plates. Two holes per five plates is the only way you can connect bracing beams at this distance. You can find it recurring in all TECHNIC models designed by LEGO engineers, and we will use it extensively in the robots in this book.

Figure 1.7 The Most Compact Locking Scheme



Upon increasing the distances, the possibilities increase; the next working combination is 10 plates/4 holes. But there are many ways we can combine beams and plates to count 10 plates in height; you can see some of them in Figure 1.8.

Figure 1.8 The Standard Grid



First question: Is there a best grid, a preferred one? Yes, there is, in a certain sense. The most versatile is version c in Figure 1.8, which is a multiple of our basic scheme from Figure 1.7, because it lets you lock the beams in an intermediate point, also. So, when you build your models, the sequence 1 beam + 2 plates + 1 beam + 2 plates… is the one that makes your life easier: Connections are possible at every second hole of the vertical beam. This is what Eric Brok on his Web site calls a *standard grid* (see Appendix A), a grid that maximizes your connection possibilities. Second question: Should you always stay with this scheme? Absolutely not! Don't curb your imagination with unnecessary constraints. This is just a tip that's useful in many circumstances, especially when you start something and don't know yet what exactly you're going to get! In many, many cases we use different schemes, and the same will be true for you.

Tilting the LEGO World: Diagonal Bracing

Who said that the LEGO beams *must* connect at a right angle to each other? The very nature of LEGO is to produce squared things, but diagonal connections are possible as well, making our world a bit more varied and interesting, and giving us another tool for problem solving.

You now know that you can cross-connect a stack of plates and beams with another beam. And you know how it works in numerical terms. So how would you brace a stack of beams with a diagonal beam?

You must look at that diagonal beam as if it was the hypotenuse of a right-angled triangle. Look at or build a stack like that in Figure 1.9. Now proceed to measure its sides, remembering not to count the first holes, because we measure lengths in terms of distances from them. The base of the triangle is 6 holes. Its height is 8 holes: Remember that in a standardized grid every horizontal beam is at a distance of two holes from those immediately below and above (we placed a vertical beam in the same picture to help you count the holes). In regards to the hypotenuse, it counts 10 holes in length.

For those of you who have never been introduced to Pythagoras, the ancient Greek philosopher and mathematician, the time has come to meet him. In what is probably the most famous theorem of all time, Pythagoras demonstrated that there's a mathematical relationship between the length of the sides of right-angled triangles. The sides composing the right angle are the catheti—let's call them A and B. The diagonal is the hypotenuse—let's call that C. The relationship is:

$$A^2 + B^2 = C^2$$

Now we can test it with our numbers:

$$6^2 + 8^2 = 10^2$$

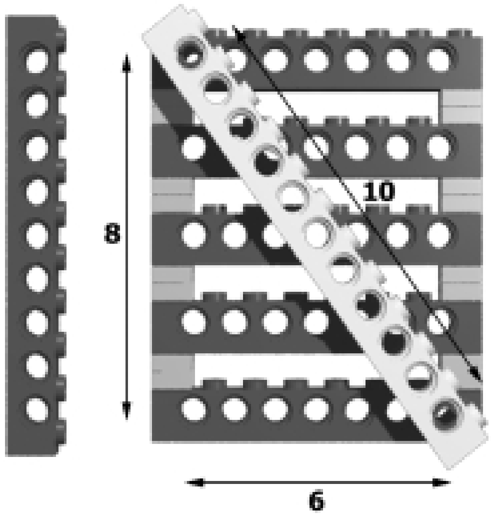
This expands to:

(6 x 6) + (8 x 8) = (10 x 10)

36 + 64 = 100

100 = 100

Figure 1.9 Pythagoras' Theorem



Yes! This is exactly why our example works so well. It's not by chance, it's good old Pythagoras' theorem. Reversing the concept, you might calculate if any arbitrary pair of base and height values brings you to a working diagonal. This is true only when the sum of the two lengths, each squared, gives a number that's the perfect square of a whole number. Let's try some examples (Table 1.1).

Table 1.1 Verifying Working Diagonal Lengths

A (Base)	B (Height)	A^2	B^2	$A^2 + B^2$	Comments
5	6	25	36	61	This doesn't work.
3	8	9	64	73	This doesn't work.
3	4	9	16	25	This works! 25 is 5 x 5.
15	8	225	64	289	This works too, though 289 is 17 x 17, this would come out a very large triangle.

Continued

Table 1.1 Continued

A (Base)	B (Height)	A^2	B^2	$A^2 + B^2$	Comments
9	8	81	64	145	145 is not the square of a whole number, but it is so close to 144 (12 x 12) that if you try and make it your diagonal beam it will fit with no effort at all. After all, the difference in length is less than 1 percent.

At this point, you're probably wondering if you have to keep your pocket calculator on your desk when playing with LEGO blocks, and maybe dig up your old high school math textbook to reread. Don't worry, you won't need either, for many reasons:

- First, you won't need to use diagonal beams very often.
- Most of the useful combinations derive from the basic triad 3-4-5 (see the third line in Table 1.1). If you multiply each side of the triangle by a whole number, you still get a valid triad. By 2: 6-8-10 (the one of our first example), by 3: 9-12-15, and so on. These are by far the most useful combinations, and are very easy to remember.
- We provide a table in Appendix B with many valid side lengths, including some that are not *perfect* but so close to the right number that they will work very well without causing any damage to your bricks.

We suggest you take some time to play with triangles, experimenting with connections using various angles and evaluating their rigidity. This knowledge will prove precious when you start building complex structures.

Expressing Horizontal Sizes and Units

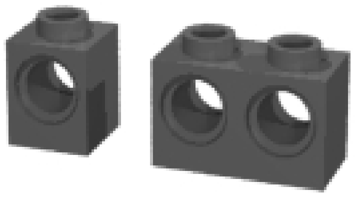
So far we've put a lot of attention into the vertical plane, because this technique of layers locked by vertical beams is the most important tool you have to build rock solid models. Well, almost rock solid, considering it's just plastic!

Nevertheless there are some other ideas you'll find useful when using bricks in the horizontal plane, that is, all studs up.

We said that the unit of measurement for length is the *stud*, meaning that we measure the length of a beam counting the number of studs it has. The holes in the beams are spaced at the same distance, so we can equally say "a length of three studs" or "a length of three holes." But looking at your beams, you have probably already noticed that the holes are interleaved with the studs, and that there is one hole less then the number of studs in each beam.

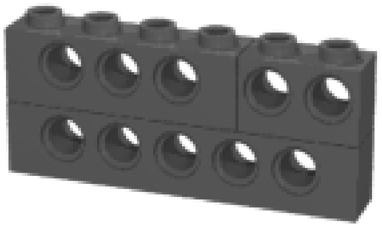
There are two important exceptions to this rule: the 1 x 1 beam with one hole, and the 1 x 2 beam with two holes (Figure 1.10). You won't find any of them in your MINDSTORMS box, but they're so useful you'll likely need some sooner or later.

Figure 1.10 The 1 x 1 Beam with 1 Hole and the 1 x 2 Beam with 2 Holes



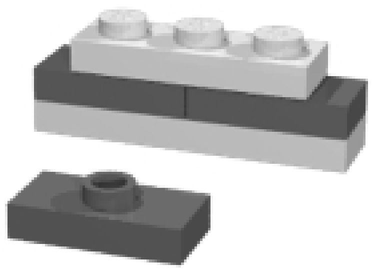
In these short beams, the holes align under the studs, not between them, and when used together with standard beams, they allow you to get increments of half a hole (Figure 1.11). We will see some practical applications of this in the next chapter when talking about gearings.

Figure 1.11 How to Get a Distance of Half a Hole



Another piece that carries out the same function is the 1 x 2 plate with one stud. This one also is not included in your MINDSTORMS kit, but it's definitely a very easy piece to find. As you can see in Figure 1.12, it's useful when you want to adjust by a distance of half a stud, and can help you a lot when fine tuning the position of touch sensors in your model. We'll see some examples of usage later on in this book.

Figure 1.12 The Single Stud 1 x 2 Plate

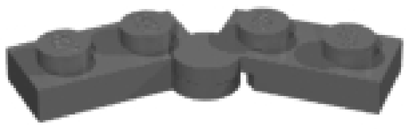


Bracing with Hinges

To close the chapter, we return to triangles. Before you start to panic, just think—you already have all the tools you need to manage them painlessly. There's nothing actually new here, just a different application of the previous concepts. Let us say in addition, that it's a technique you can survive without. But for the sake of completeness, we want to introduce it also.

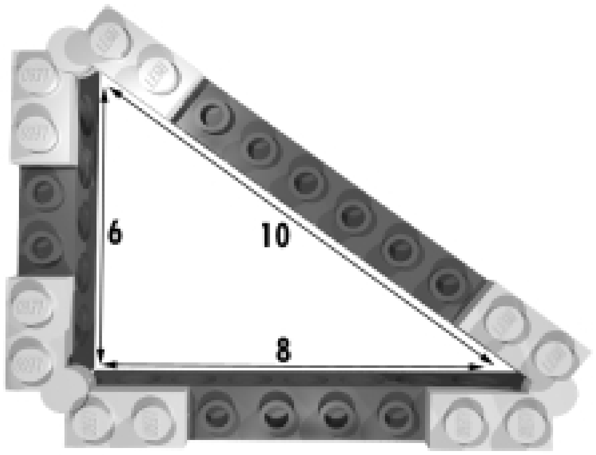
First of all we need yet another special part, a *hinge* (Figure 1.13). Using these hinges you can build many different triangles, but once again our interest is on right-angle triangles, because they are by far the most useful triangle for connections. Their catheti align properly with lower or upper layers of plates or beams, offering many possibilities of integration with other structures.

Figure 1.13 The LEGO Hinge



The LEGO hinges let you rotate the connected beams, keeping their inner corners always in contact. Therefore, using three hinges, you get a triangle whose vertices fall in the rotation centers of the hinges. The length of its *inner* sides is the length of the beams you count (Figure 1.14). Regarding right-angled triangles: You're already familiar with the Pythagorean Theorem, and it applies to this case as well. The same combinations we have already seen work in this case: 3-4-5, 6-8-10, and so on.

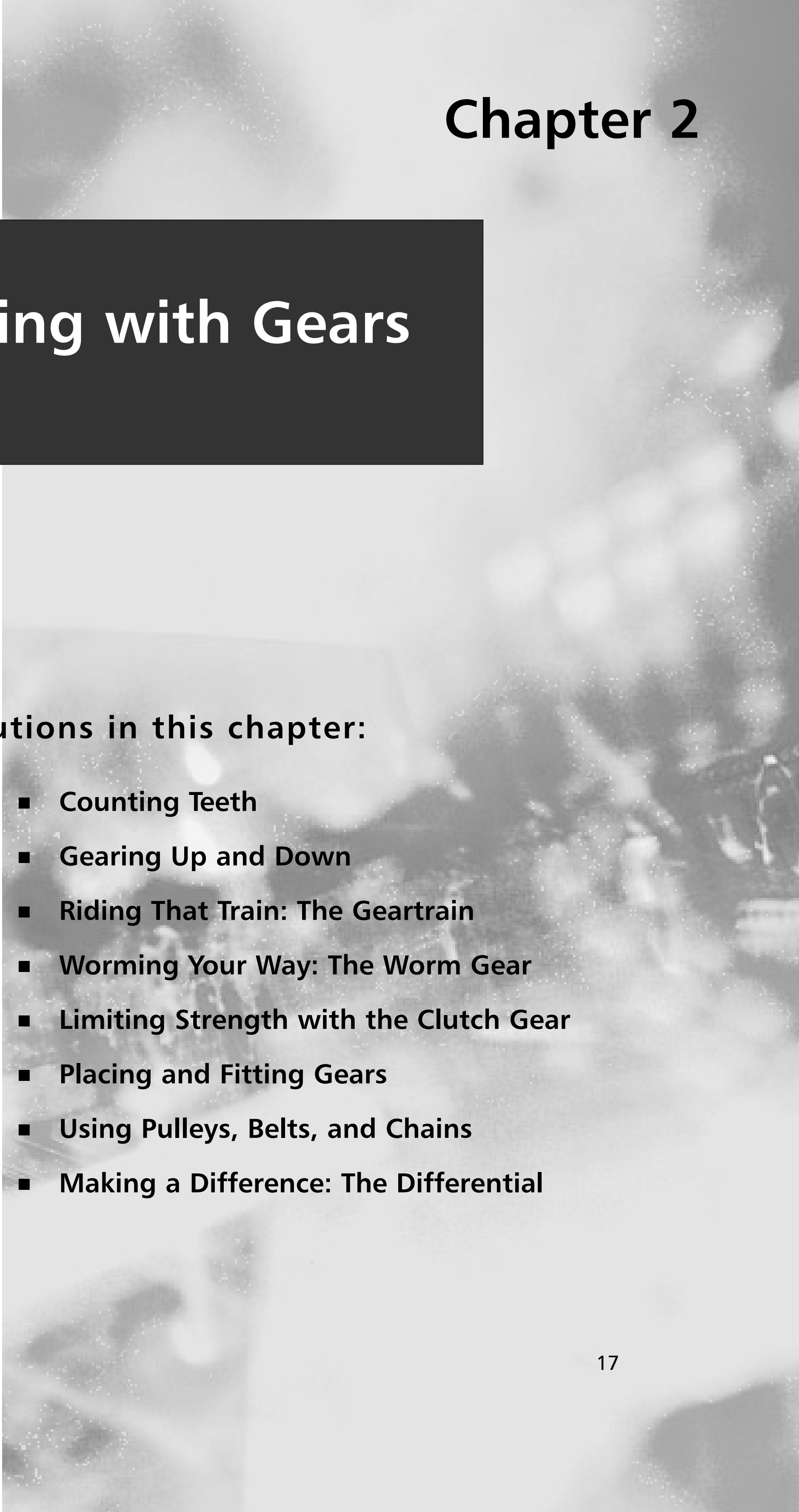
Figure 1.14 Making a Triangle with Hinges



Summary

Did you survive the geometry? You can see it doesn't have to be that hard once you get familiar with the basics. First, it helps to know how to identify the bricks by their proportions, counting the length and width by studs, and recognizing that the vertical unit to horizontal unit ratio is 6 to 5. Thus, according to the simple ratio, when you're trying to find a locking scheme to insert axles or pins into perpendicular beam holes, you know that every 5 bricks in height, the holes of a crossed beam match up. Also, because three plates match the height of a brick, the most compact locking scheme is to use increments of two plates and a brick, because it gives you that magic multiple of 5. If you stay with this scheme, the standard grid, everything will come easy: one brick, two plates, one brick, two plates...

To fit a diagonal beam, use the Pythagorean Theorem. Combinations based on the triad of 3-4-5 constitute a class of easy-to-remember distances for the beam to make a right triangle, but there are many others. Either use the rules explained here, or simply look up the connection table provided in Appendix B.



Chapter 2

Playing with Gears

Solutions in this chapter:

- Counting Teeth
- Gearing Up and Down
- Riding That Train: The Geartrain
- Worming Your Way: The Worm Gear
- Limiting Strength with the Clutch Gear
- Placing and Fitting Gears
- Using Pulleys, Belts, and Chains
- Making a Difference: The Differential

Introduction

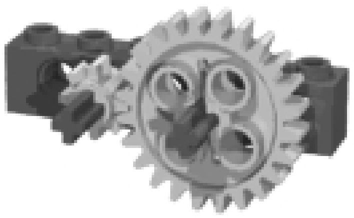
You might find yourself asking: Do I really *need* gears? Well, the answer is yes, you do. Gears are so important for machines that they are almost their symbol: Just the sight of a gear makes you think *machinery*. In this chapter, you will enter the amazing world of gears and discover the powerful qualities they offer, transforming one force into another almost magically. We'll guide you through some new concepts—velocity, force, torque, friction—as well as some simple math to lay the foundations that will give you the most from the machinery. The concepts are not as complex as you might think. For instance, the chapter will help you see the parallels between gears and simple levers.

We invite you once again to experiment with the real things. Prepare some gears, beams, and axles to replicate the simple setups of this chapter. No description or explanation can replace what you learn through hands-on experience.

Counting Teeth

A single gear wheel alone is not very useful—in fact, it is not useful at all, unless you have in mind a different usage from what it was conceived for! So, for a meaningful discussion, we need at least two gears. In Figure 2.1, you can see two very common LEGO gears: The left one is an 8t, while the right is a 24t. The most important property of a gear, as we'll explain shortly, is its *teeth*. Gears are classified by the number of teeth they have; the description of which is then shortened to form their name. For instance, a gear with 24 teeth becomes "a 24t gear."

Figure 2.1 An 8t and a 24t Gear



Let's go back to our example. We have two gears, an 8t and a 24t, each mounted on an axle. The two axles fit inside holes in a beam at a distance of two holes (one empty hole in between). Now, hold the beam in one hand, and with the other hand gently turn one of the axles. The first thing you should notice is that when you turn one axle, the other turns too. The gears are *transferring motion from one axle to the other.* This is their fundamental property, their very nature. The second important thing you should notice is that you are not required to apply much strength to make them turn. Their teeth match well and there is only a small amount of friction. This is one of the great characteristics of the LEGO TECHNIC system: Parts are designed to match properly at standard distances. A third item of note is that the two axles turn in opposite directions: one clockwise and the other counterclockwise.

A fourth, and more subtle, property you should have picked up on is that the two axles revolve at different speeds. When you turn the 8t, the 24t turns more slowly, while turning the 24t makes the 8t turn faster. Lets explore this in more detail.

Gearing Up and Down

Let's start turning the larger gear in our example. It has 24 teeth, each one meshing perfectly between two teeth of the 8t gear. While turning the 24t, every time a new tooth takes the place of the previous one in the contact area of the gears, the 8t gear turns exactly one tooth, too. The key point here is that you need to advance only 8 teeth of the 24 to make the small gear do a complete turn (360°). After 8 teeth more of your 24, the small gear has made a second revolution. With the last 8 teeth of your 24, the 8t gear makes its third turn. This is why there is a difference in speed: For every turn of the 24t, the 8t makes three turns! We express this relationship with a ratio that contains the number of teeth in both gears: 24 to 8. We can simplify it, dividing the two terms by the smaller of the two (8), so we get 3 to 1. This makes it very clear, in numerical terms, that one turn of the first corresponds to three turns of the second.

You have just found a way to get more speed! (To be technically precise, we should call it *angular velocity*, not *speed*, but you get the idea). Before you start imagining mammoth gear ratios for racecar robots, sorry to disappoint you—there is no free lunch in mechanics, you have to pay for this gained speed. You pay for it with a decrease in *torque*, or, to keep in simple terms, a decrease in strength.

So, our gearing is able to convert torque to velocity—the more velocity we want, the more torque we must sacrifice. The ratio is exactly the same, if you get three times your original angular velocity, you reduce the resulting torque to one third.

One of the nice properties of gears is that this conversion is symmetrical: You can convert torque into velocity or vice versa. And the math you need to manage and understand the process is as simple as doing one division. Along common conventions, we say that we *gear up* when our system increases velocity and reduces torque, and that we *gear down* when it reduces velocity and increases torque. We usually write the ratio 3:1 for the former and 1:3 for the latter.

Bricks & Chips…

What Is Torque?

When you turn a nut on a bolt using a wrench, you are producing *torque*. When the nut offers some resistance, you've probably discovered that the more the distance from the nut you hold the wrench, the less the force you have to apply. Torque is in fact the product of two components: *force* and *distance*. You can increase torque by either increasing the applied force, or increasing the distance from the center of rotation. The units of measurement for torque are thus a unit for the force, and a unit for the distance. The International System of Units (SI) defines the newton-meter (Nm) and the newton-centimeter (Ncm).

If you have some familiarity with the properties of levers, you will recognize the similarities. In a lever, the resulting force depends on the distance between the application point and the fulcrum: the longer the distance, the higher the force. You can think of gears as levers whose fulcrum is their axle and whose application points are their teeth. Thus, applying the same force to a larger gear (that is to a longer lever) results in an increase in torque.

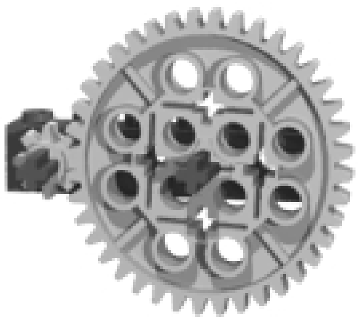
When should you gear up or down? Experience will tell you. Generally speaking, you will gear down many more times then you will gear up, because you'll be working with electric motors that have a relatively high velocity yet a fairly low torque. Most of the time, you reduce speed to get more torque and make your vehicles climb steep slopes, or to have your robotic arms lift some load. Other times you don't need the additional torque; you simply want to reduce speed to get more accurate positioning.

One last thing before you move on to the next topic. We said that there is no free lunch when it comes to mechanics. This is true for this conversion service as well: We have to pay something to get the conversion done. The price is paid in *friction*—something you should try and keep as low as possible—but it's unavoidable. Friction will always eat up some of your torque in the conversion process.

Riding That Train: The Geartrain

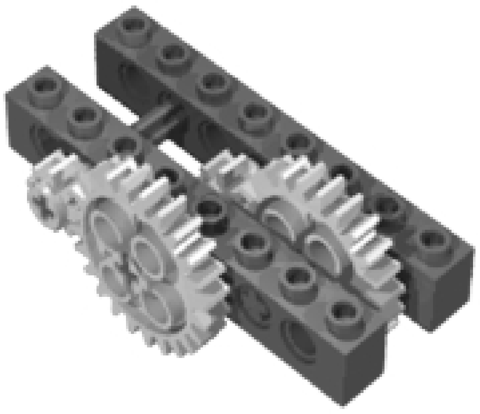
The largest LEGO gear is the 40t, while the smallest is the 8t (used in the previous discussion). Thus, the highest ratio we can obtain is 8:40, or 1:5 (Figure 2.2).

Figure 2.2 A 1:5 Gear Ratio



What if you need an even higher ratio? In such cases, you should use a *multi-stage reduction* (or multiplication) system, usually called a *geartrain*. Look at Figure 2.3. In this system, the result of a first 1:3 reduction stage is transferred to a second 1:3 reduction stage. So, the resulting velocity is one third of one third, which is one ninth, while the resulting torque is three times three, or nine. Therefore, the ratio is 1:9.

Figure 2.3 A Geartrain with a Resulting Ratio of 1:9



Geartrains give you incredible power, because you can trade as much velocity as you want for the same amount of torque. Two 1:5 stages result in a ratio of 1:25, while three of them result in 1:125 system! All this strength must be used with care, however, because your LEGO parts may get damaged if for any reason your robot is unable to convert it into some kind of work. In other words, if something gets jammed, the strength of a LEGO motor multiplied by 125 is enough to deform your beams, wring your axles, or break the teeth of your gears. We'll return to this topic later.

Designing & Planning…

Choosing the Proper Gearing Ratio

We suggest you perform some experiments to help you make the right decision in choosing a gearing ratio. Don't wait to finish your robot to discover that some geared mechanics doesn't work as expected! Start building a very rough prototype of your robot, or just of a particular sub-system, and experiment with different gear ratios until you're satisfied with the result. This prototype doesn't need to be very solid or refined, and doesn't even need to resemble the finished system you have in mind. It is important, however, that it accurately simulates the kind of work you're expecting from your robot, and the actual loads it will have to manage. For example, if your goal is to build a robot capable of climbing a slope with a 50 percent grade, put on your prototype all the weight you imagine your final model is going to carry: additional motors for other tasks, the RCX itself, extra parts, and so on. Don't test it without load, as you might discover it doesn't work.

NOTE

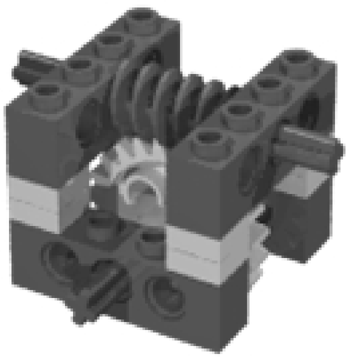
Remember that in adding multiple reduction stages, each additional stage introduces further *friction*, the bad guy that makes your world less than ideal. For this reason, if aiming for maximum efficiency, you should try and reach your final ratio with as few stages as possible.

Worming Your Way: The Worm Gear

In your MINDSTORMS box you've probably found another strange gear, a black one that resembles a sort of cylinder with a spiral wound around it. Is this thing really a gear? Yes, it is, but it is so peculiar we have to give it special mention.

In Figure 2.4, you can see a worm gear engaged with the more familiar 24t. In just building this simple assembly, you will discover many properties. Try and turn the axles by hand. Notice that while you can easily turn the axle connected to the worm gear, you can't turn the one attached to the 24t. We have discovered the first important property: The worm gear leads to an *asymmetrical system*; that is, you can use it to turn other gears, but it can't be turned *by* other gears. The reason for this asymmetry is, once again, friction. Is this a bad thing? Not necessarily. It can be used for other purposes.

Figure 2.4 A Worm Gear Engaged with a 24t



Another fact you have likely observed is that the two axles are perpendicular to each other. This change of orientation is unavoidable when using worm gears.

Turning to gear ratios, you're now an expert at doing the math, but you're probably wondering how to determine how many teeth this worm gear has! To figure this out, instead of discussing the theory behind it, we proceed with our experiment. Taking the assembly used in Figure 2.4, we turn the worm gear axle slowly by exactly one turn, at the same time watching the 24t gear. For every turn you make, the 24t rotates by exactly one tooth. This is the answer you were looking for: the worm gear is a 1t gear! So, in this assembly, we get a 1:24 ratio with a single stage. In fact, we could go up to 1:40 using a 40t instead of a 24t.

The asymmetry we talked about before makes the worm gear applicable only in reducing speed and increasing torque, because, as we explained, the friction of this particular device is too high to get it rotated by another gear. The same high friction also makes this solution very inefficient, as a lot of torque gets wasted in the process.

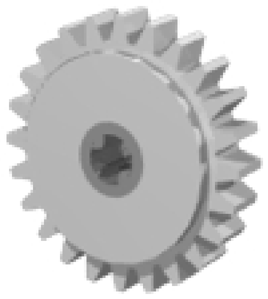
As we mentioned earlier, this outcome is not always a bad thing. There are common situations where this asymmetry is exactly what we want. For example, when designing a robotic arm to lift a small load. Suppose we use a 1:25 ratio made with standard gears: what happens when we stop the motor with the arm loaded? The symmetry of the system transforms the weight of the load (potential energy) into torque, the torque into velocity, and the motor spins back making the arm go down. In this case, and in many others, the worm gear is the proper solution, its friction making it impossible for the arm to turn the motor back.

We can summarize all this by saying that in situations where you desire precise and stable positioning under load, the worm gear is the right choice. And it's also the right choice when you need a high reduction ratio in a small space, since allows very compact assembly solutions.

Limiting Strength with the Clutch Gear

Another special device you should get familiar with is the thick 24t white gear, which has strange markings on its face (Figure 2.5). Its name is *clutch gear*, and in the next part of this section we'll discover just what it does.

Figure 2.5 The Clutch Gear



Our experiment this time requires very little work, just put the end of an axle inside the clutch gear and the other end into a standard 24t to use as a knob. Keep the latter in place with one hand and slowly turn the clutch gear with the other hand. It offers some resistance, but it turns. This is its purpose in life: to offer some resistance, then give in!

This clutch gear is an invaluable help to limit the strength you can get from a geared system, and this helps to preserve your motors, your parts, and to resolve some difficult situations. The mysterious "2.5·5 Ncm" writing stamped on it (as explained earlier, Ncm is a newton-centimeter, the unit of measurement for torque) indicates that this gear can transmit a maximum torque of about 2.5 to 5 Ncm. When exceeding this limit its internal clutch mechanism starts to slip.

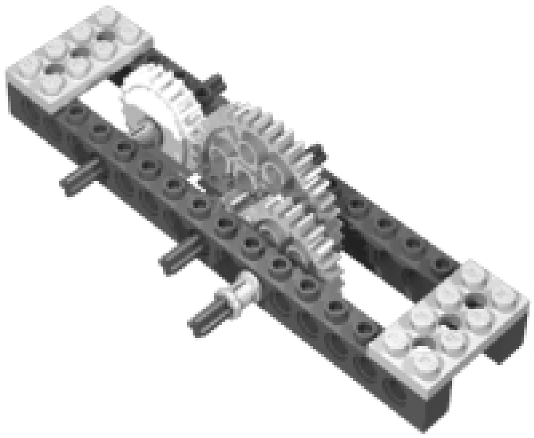
What's this feature useful for? You have seen before that through some reduction stages you can multiply your torque by high factors, thus getting a system strong enough to actually damage itself if something goes wrong. This clutch gear helps you avoid this, limiting the final strength to a reasonable value.

There are other cases in which you don't gear down very much and the torque is not enough to ruin your LEGO parts, but if the mechanics jam, the motor stalls—this is a very bad thing, because your motor draws a lot of current and risks permanent damage. The clutch gear prevents this damage, automatically disengaging the motor when the torque becomes too high.

In some situations, the clutch gear can even reduce the number of sensors needed in your robot. Suppose you build a motorized mechanism with a bounded range of action, meaning that you simply want your subsystem (arms, levers, actuators—anything) to be in one of two possible states: open or closed, right or left, engaged or disengaged, with no intermediate position. You need to turn on the motor for a short time to switch over the mechanism from one state to the other, but unfortunately it's not easy to calculate the precise time a motor needs to be on to perform a specific action (even worse, when the load changes, the required time changes, too). If the time is too short, the system will result in an intermediate state, and if it's too long, you might do damage to your motor. You can use a sensor to detect when the desired state has been reached; however, if you put a clutch gear somewhere in the geartrain, you can now run the motor for the approximate time needed to reach the limit in the worst load situation, because the clutch gear slips and prevents any harm to your robot and to your motor if the latter stays on for a time longer than required.

There's one last topic about the clutch gear we have to discuss: where to put it in our geartrain. You know that it is a 24t and can transmit a maximum torque of 5 Ncm, so you can apply here the same gear math you have learned so far. If you place it before a 40t gear, the ratio will be 24:40, which is about 1:1.67. The maximum torque driven to the axle of the 40t will be 1.67 multiplied by 5 Ncm, resulting in 8.35 Ncm. In a more complex geartrain like that in Figure 2.6, the ratio is 3:5 then 1:3, coming to a final 1:5; thus the maximum resulting torque is 25 Ncm. A system with an output torque of 25 Ncm will be able to produce a force five times stronger than one of 5 Ncm. In other words, it will be able to lift a weight five times heavier.

Figure 2.6 Placing the Clutch Gear in a Geartrain



From these examples, you can deduce that the maximum torque produced by a system that incorporates a clutch gear results from the maximum torque of the clutch gear multiplied by the ratio of the following stages. When gearing down, the more output torque you want, the closer you have to place your clutch gear to the source of power (the motor) in your geartrain. On the contrary, when you are reducing velocity, not to get torque but to get more accuracy in positioning, and you really want a soft touch, place the clutch gear as the very last component in your geartrain. This will minimize the final supplied torque.

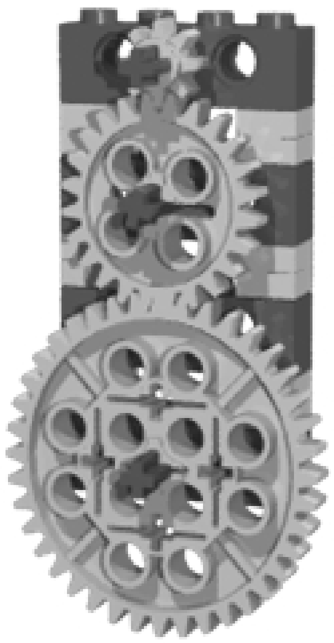
This might sound a bit complex, but we again suggest you learn by doing, rather than by simply reading. Prototyping is a very good practice. Set up some very simple assemblies to experiment with the clutch gear in different positions, and discover what happens in each case.

Placing and Fitting Gears

The LEGO gear set includes many different types of gear wheels. Up to now, we played with the straight 8t, 24t, and 40t, but the time has come to explore other kinds of gears, and to discuss their use according to size and shape.

The 8t, 24t, and 40t have a radius of 0.5 studs, 1.5 studs, and 2.5 studs, respectively (measured from center to half the tooth length). The distance between the gears' axles when placing them is the sum of their radii, so it's easy to see that those three gears make very good combinations at distances corresponding to whole numbers. 8t to 24t is 2 studs, 8t to 40t is 3 studs, and 24t to 40t equates to four studs. The pairs that match at an even distance are very easy to connect one above the other in our standard grid, because we know it goes by increments of two studs for every layer (Figure 2.7).

Figure 2.7 Vertical Matching of Gears



Another very common straight gear is the 16t gear (Figure 2.8). Its radius is 1, and it combines well with a copy of itself at a distance of two. Getting it to cooperate with other members of its family, however, is a bit more tricky, because whenever matched with any of the other gears it leads to a distance of some studs *and a half*, and here is where the special beams we discussed in the previous chapter (1 x 1, 1 hole, and 1 x 2, 2 holes) may help you (Figure 2.9).

Figure 2.8 The 16t Gear

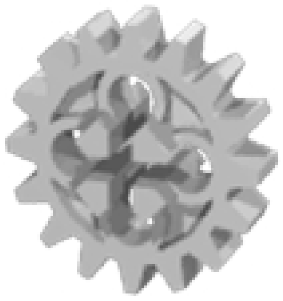
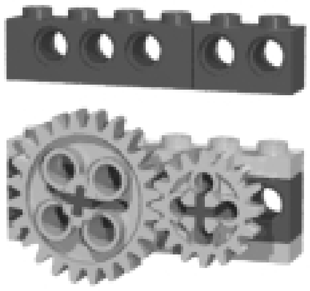


Figure 2.9 How to Match the 16t Gear to a 24t Gear



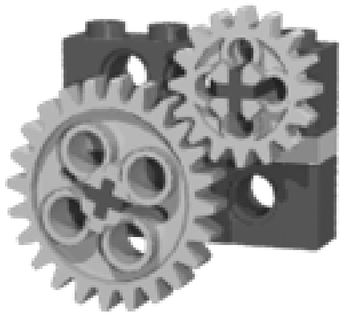
Bricks & Chips…

Idler Gears

Figure 2.7 offers us the opportunity to talk about *idler gears*. What's the ratio of the geartrain in the figure? Starting from the 8t, the first stage performs an 8:24 reduction, while the second is a 24:40. Multiplying the two fractions, you get 8:40, or 1:5, the same result you'd get meshing the 8t directly to the 40t. The intermediate 24t is an idler gear, which doesn't affect the gear ratio. Idler gears are quite common in machines, usually to help connect distant axles. Are idler gears totally lacking in effects on the system? No, they have one very important effect: They change the direction of the output!

As we've already said, you're not restricted to using the standard grid. You can try out different solutions that don't require any special parts, like the one showed in Figure 2.10.

Figure 2.10 A Diagonal Matching



When using a pair of 16t gears, the resulting ratio is 1:1. You don't get any effect on the angular velocity or torque (except in converting a fraction of them into friction), but indeed there are reasons to use them as a pair—for instance, when you want to transfer motion from one axle to another with no other effects. This is, in fact, another task that gears are commonly useful for. There's even a class of gears specifically designed to transfer motion from one axle to another axle perpendicular to it, called *bevel gears*.

Designing & Planning…

Backlash

Diagonal matching is often less precise than horizontal and vertical types, because it results in a slightly larger distance between gear teeth. This extra distance increases the *backlash*, the amount of oscillation a gear can endure without affecting its meshing gear. Backlash is amplified when gearing up, and reduced when gearing down. It generally has a bad effect on a system, reducing the precision with which you can control the output axle, and for this reason, it should be kept to a minimum.

The most common member of this class is the 12t bevel gear, which can be used *only* for this task (Figure 2.11), meaning it does not combine at all with any other LEGO gear we have examined so far. Nevertheless, it performs a very useful function, allowing you to transmit the motion toward a new direction, while using a minimum of space. There's also a new 20t bevel conical gear with the same design of the common 12t (Figure 2.12). Both of these bevel gears are half a stud in thickness, while the other gears are 1 stud.

Figure 2.11 Bevel Gears on Perpendicular Axles

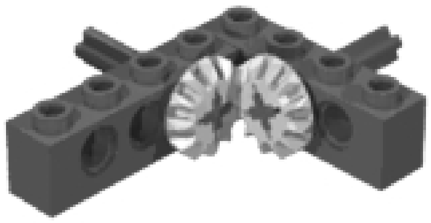
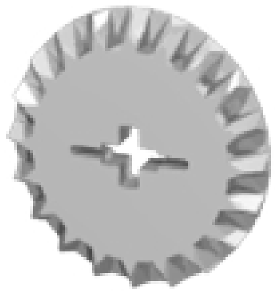


Figure 2.12 The 20t Bevel Gear



The 24t gear also exists in the form of a *crown gear*, a special gear with front teeth that can be used like an ordinary 24t, which can combine with another straight gear to transmit motion in an orthogonal direction (that is, composed of right angles), possibly achieving at the same time a ratio different from 1:1 (Figure 2.13).

To conclude our discussion of gears, we'll briefly introduce some recent types not included in the MINDSTORMS kit, but that you might find inside other LEGO sets. The two *double bevel* ones in Figure 2.14 are a 12t and a 20t, respectively 0.75 and 1.25 studs in radius. If you create a pair that includes one per kind of the two, they are an easy match at a distance of 2 studs.

Figure 2.13 The Crown Gear on Perpendicular Axles

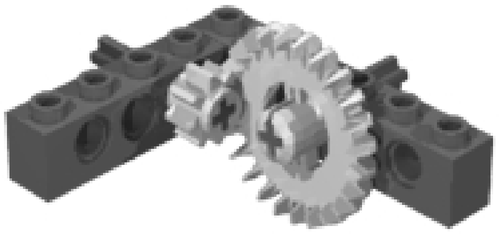
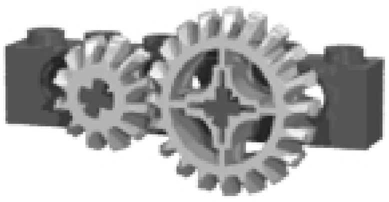
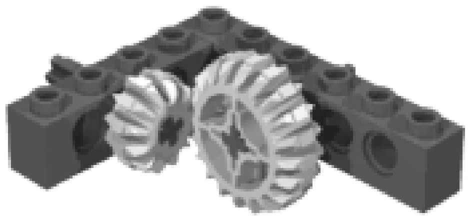


Figure 2.14 Double Bevel Gears



Things get a bit more complicated when you want to couple two of the same kind, as the resulting distance is 1.5 or 2.5. And even more complicated when combined with other gears, causing resulting distances that include a quarter or three quarters of a stud. These gears are designed to work well in perpendicular setups as well (Figure 2.15).

Figure 2.15 Double Bevel Gear on Perpendicular Axles



Using Pulleys, Belts, and Chains

The MINDSTORMS kit includes some *pulleys* and *belts*, two classes of components designed to work together and perform functions similar to that of gears—

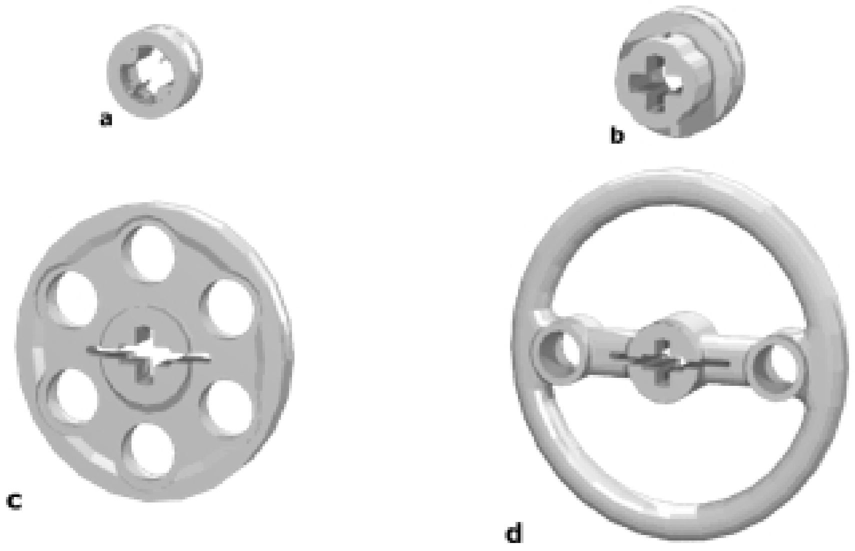
similar, that is, but not identical. They have indeed some peculiarities which we shall explore in the following paragraphs.

Chains, on the other hand, are not part of the basic MINDSTORMS kit. You will need to buy them separately. Though not essential, they allow you to create mechanical connections that share some properties with both geartrains and pulley-belt systems.

Pulleys and Belts

Pulleys are like wheels with a groove (called a *race*) along their diameter. The LEGO TECHNIC system currently includes four kinds of pulleys, shown in Figure 2.16.

Figure 2.16 Pulleys



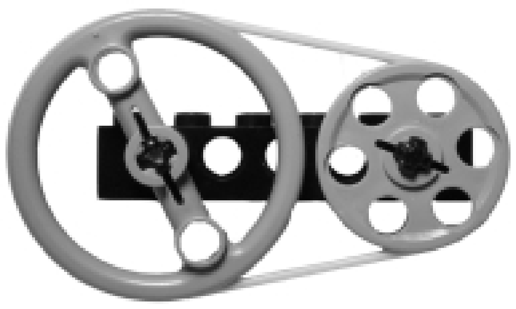
The smallest one (a) is actually the half-size bush, normally used to hold axles in place to prevent them from sliding back and forth. Since it does have a race, it can be properly termed a pulley. Its diameter is one LEGO unit, with a thickness of half a unit.

The small pulley (b) is 1 unit in thickness and about 1.5 units in width. It is asymmetrical, however, since the race is not in the exact center. One side of the axle hole fits a rubber ring that's designed to attach this pulley to the micro-motor. The medium pulley (c) is again half a unit thick and 3 units in diameter. Finally, the large pulley (d) is 1 unit thick and about 4.5 units in diameter.

LEGO belts are rings of rubbery material that look similar to rubber bands. They come in three versions in the MINDSTORMS kit, with different colors corresponding to different lengths: white, blue, and yellow (in other sets, you can find a fourth size in red). Don't confuse them with the actual rubber bands, the black ones you found in the kit: Rubber bands have much greater elasticity, and for this reason are much less suitable to the transfer of motion between two pulleys. This is, in fact, the purpose of belts: to connect a pair of pulleys. LEGO belts are designed to perfectly match the race of LEGO pulleys.

Let's examine a system made of a pair of pulleys connected through a belt (Figure 2.17). The belt transfers motion from one pulley to the other, making them similar to a pair of gears. How do you compute the ratio of the system? You don't have any teeth to count... The rule with pulleys is that the reduction ratio is determined by finding the ratio between their diameters (this rules applies to gears too, but the fact that their circumference is covered with evenly spaced teeth provides a convenient way to avoid measurement). You actually should consider the diameter of the pulley *inside* its race, because the sides of the race are designed specifically to prevent the belt from slipping from the pulley and don't count as part of the diameter the belt acts over.

Figure 2.17 Pulleys Connected with a Belt



You must also consider that pulleys are not very suitable to transmitting high torque, because the belts tend to slip. The amount of slippage is not easy to estimate, as it depends on many factors, including the torque and speed, the tension of the belt, the friction between the belt and the pulley, and the elasticity of the belt.

For those reasons, we preferred an experimental approach and measured some actual ratios among the different combination of pulleys under controlled conditions. You can find our results in Table 2.1.

Table 2.1 Ratios Among Pulleys

	Half Bush	Small Pulley	Medium Pulley	Large Pulley
Half bush	1:1	1:2	1:4	1:6
Small pulley	2:1	1:1	1:2.5	1:4.1
Medium pulley	4:1	2.5:1	1:1	1:1.8
Large pulley	6:1	4.1:1	1.8:1	1:1

Designing & Planning…

Finding the Ratio between Two Pulleys

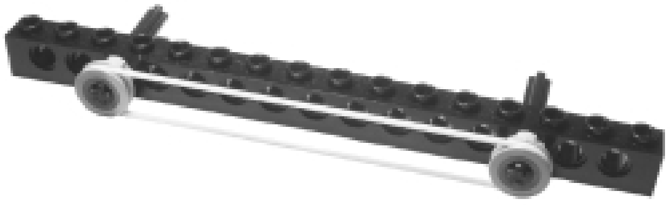
How did we find out the actual ratio between two pulleys? By simply connecting them with a belt and comparing the number of rotations when one of the two gets turned and drags the other. But turning pulleys by hand would have been quite a boring and time-consuming task, and could cause some counts to be missed. What better device for this job than our RCX, equipped with a motor and two rotation sensors? So, we built this very simple machine: a motor connected to a pulley, whose axle is attached to the first rotation sensor, and a second pulley, placed at a very short distance, with its axle attached to the second rotation sensor. We used some care to minimize the friction and maintain the same tension in the belt for all the pairs of pulleys.

When running the motor, the RCX counted the rotations for us. We stopped the motor after a few seconds, read the rotation sensor counts, and divided the two to get the ratio you see in Table 2.1.

These values may change significantly in a real-world application, when the system is under load. Because of this, it's best to think of the figures as simply an indication of a possible ratio for systems where very low torque is applied. Generally speaking, you should use pulleys in your first stages of a reduction system, where the velocity is high and the torque still low. You could even view the slippage problem as a positive feature in many cases, acting as a torque-limiting mechanism like the one we discussed in the clutch gear, with the same benefits and applications.

Another advantage of pulleys over gear wheels is that their distance is not as critical. Indeed, they help a great deal when you need to transfer motion to a distant axle (Figure 2.18). And at high speeds they are much less noisy than gears—a facet that occasionally comes in handy.

Figure 2.18 Pulleys Allow Transmission across Long Distances

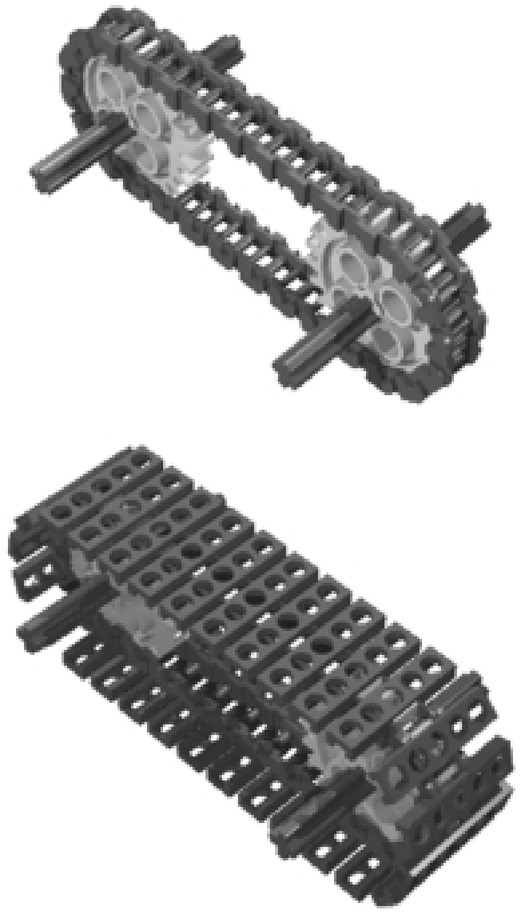


Chains

LEGO *chains* come in two flavors: *chain links* and *tread links* (as shown in Figure 2.19, top and bottom, respectively). The two share the same hooking system and are freely mixable to create a chain of the required length.

Chains are used to connect gear wheels as the same way belts connect with pulleys. They share similar properties as well: Both systems couple parallel axles without reversing the rotation direction, and both give you the chance to connect distant axles. The big difference between the two is that chain links don't allow any slippage, so they transfer *all* the torque. (The maximum torque a chain can transfer depends on the resistance of its individual links, which in the case of LEGO chains is not very high.) On the other hand, they introduce further friction into the system, and for this reason are much less efficient then direct gear matches. You will find chains useful when you have to transfer motion to a distant axle in low velocity situations. The ratio of two gears connected by a chain is the same as their corresponding direct connection. For example, a 16t connected to a 40t results in a 2:5 ratio.

Figure 2.19 Chain Links

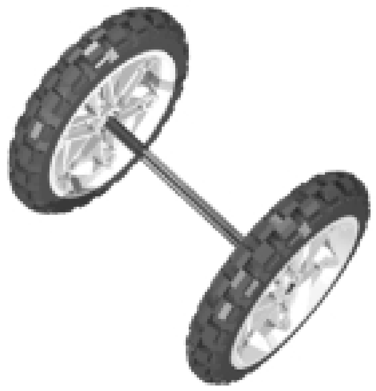


Making a Difference: The Differential

There's a very special device we want to introduce you to at this time: the *differential gear.* You probably know that there's at least one differential gear in every car. What you might not know is why the differential gear is so important.

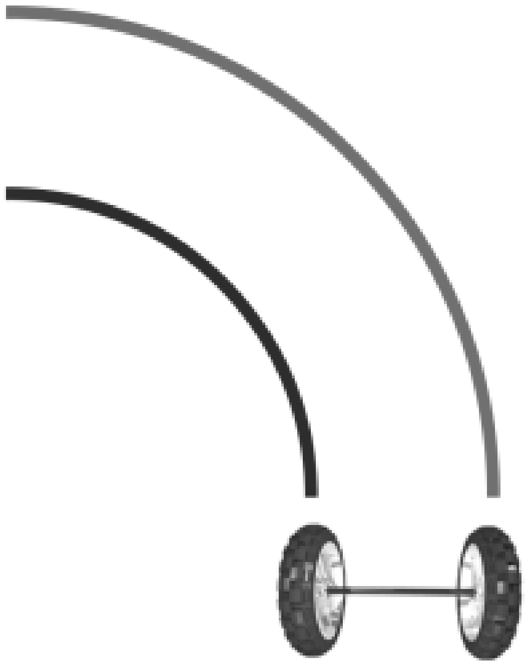
Let's do an experiment together. Take the two largest wheels that you find in the MINDSTORMS kit and connect their hubs with the longest axle (Figure 2.20). Now put the wheels on your table and push them gently: They run smoothly and advance some feet, going straight. *Very straight.* Keep the axle in the middle with your fingers and try to make the wheels change direction while pushing them. It's not so easy, is it?

Figure 2.20 Two Connected Wheels Go Straight



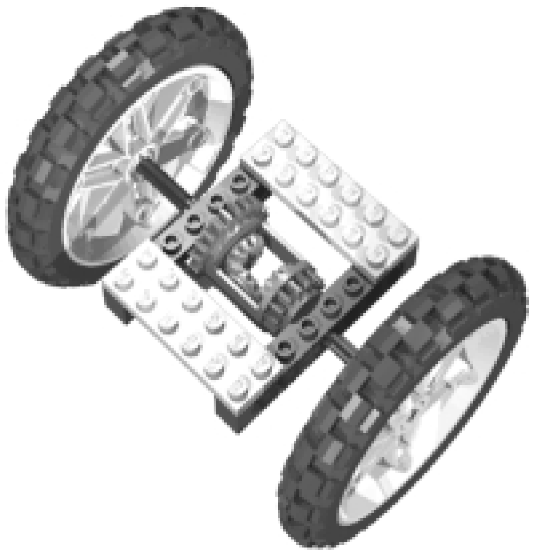
The reason is that when two parallel wheels turn, their paths must have different lengths, the outer one having a longer distance to cover (Figure 2.21). In our example, in which the wheels are rigidly connected, at any turn they cover the same distance, so there's no way to make them turn unless you let one slip a bit.

Figure 2.21 During Turns the Wheels Cover Different Distances



The next phase of our experiment requires that you now build the assembly shown in Figure 2.22. You see a differential gear with its three 12t bevel gears, two 6-stud axles, and two beams and plates designed to provide you with a way to handle this small system. Placing the wheels again on your table, you will notice that while pushing them, you can now easily turn smoothly in any direction. Please observe carefully the *body* of the differential gear and the central bevel gear: when the wheels go straight, the body itself rotates while the bevel gear is stationary. On the other hand, if you turn your system in place, the body stays put and the bevel gear rotates. In any other intermediate case, both of them rotate at some speed, adapting the system to the situation. Differentials offer a way to put power to the wheels without the restriction of a single fixed drive axle.

Figure 2.22 Connecting Wheels with the Differential Gear



To use this configuration in a vehicle, you simply have to apply power to the body of the differential gear, which incorporates a 24t on one side and a 16t on the other.

The differential gear has many other important applications. You can think of it as a mechanical adding/subtracting device. Again place the assembly from Figure 2.22 on your table. Rotate one wheel while keeping the other from turning; the body of the differential gear rotates half the angular velocity of the rotating wheel. You already discovered that when turning our system in place, the differential does not rotate at all, and then when both wheels rotate together, the differential rotates at the same speed as well. From this behavior, we can infer a simple formula:

(Iav1 + Iav2) / 2 = Oav

where *Oav* is the *output angular velocity* (the body of the differential gear), and *Iav1* and *Iav2* are the *input angular velocities* (the two wheels). When applying this equation, you must remember to use *signed* numbers for the input, meaning that if one of the input axles rotates in the opposite direction of the other, you must input its velocity as a negative number. For example, if the right axle rotates at 100 rpm (revolutions per minute) and the left one at 50 rpm, the angular velocity of the body of the differential results in this:

(100 rpm + 50 rpm) / 2 = 75 rpm

There are situations where you deliberately reverse the direction of one input, using idler gears, to make the differential sensitive to a difference in the speed of the wheels, rather than to their sum. Reversing the input means that you must make one of the inputs negative. See what happens to the differential when both wheels run at the same speed, let's say 100 rpm:

(100 rpm – 100 rpm) / 2 = 0 rpm

It doesn't move! As soon as a difference in speed appears, the differential starts rotating with an angular velocity equal to half this difference:

(100 rpm – 98 rpm) / 2 = 1 rpm

This is a useful trick when you want to be sure your wheels run at the same speed and cover the same distance: Monitor the body of the differential and slow the left or right wheel appropriately to keep it stationary. See Chapter 8 for a practical application of this trick.

Summary

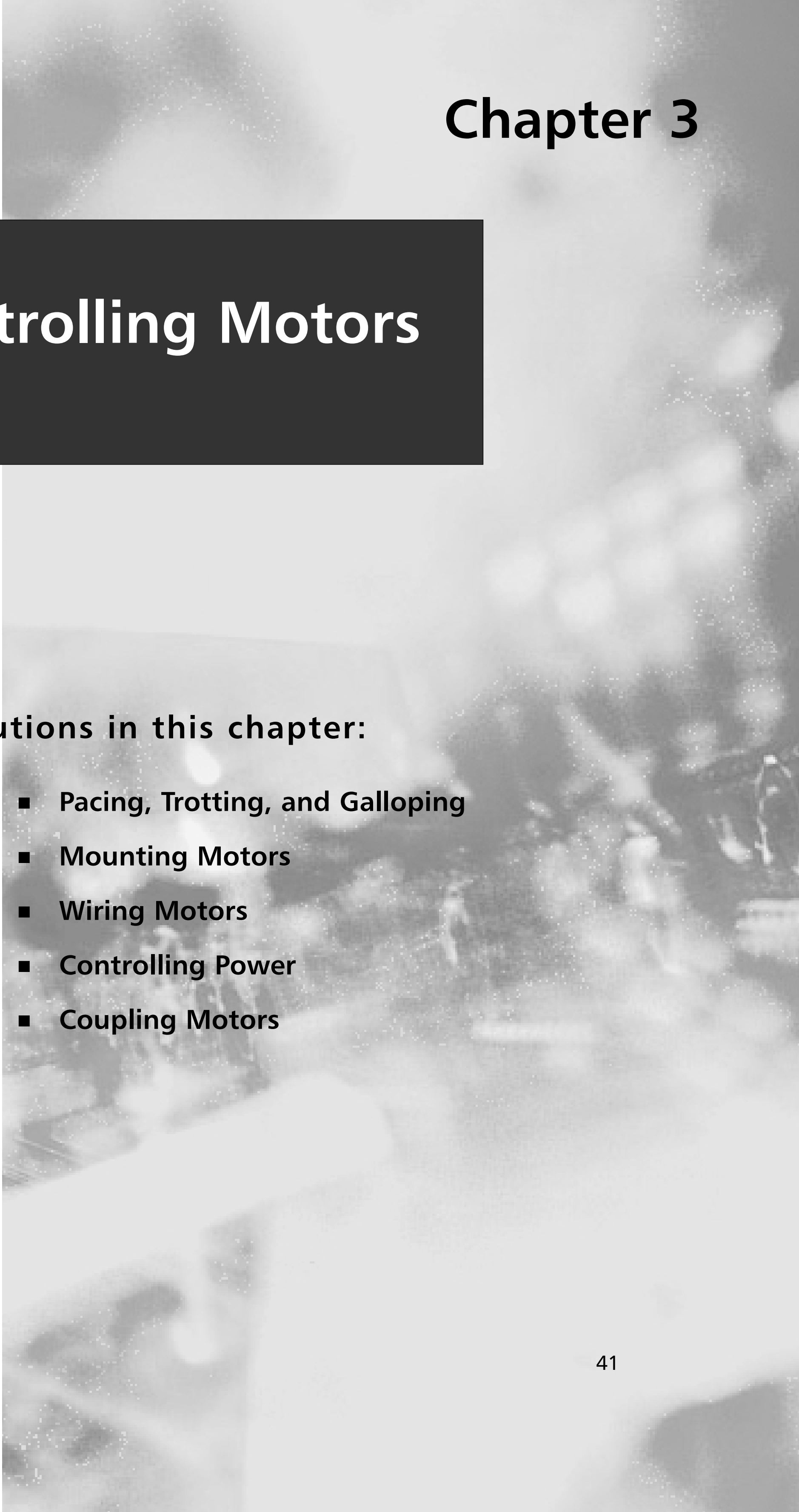
Few pieces of machinery can exist without gears, including robots, and you ought to know how to get the most benefit from them. In this chapter, you were introduced to some very important concepts: gear ratios, angular velocity, force, torque, and friction. Torque is what makes your robot able to perform tasks involving force or weight, like lifting weights, grabbing objects, or climbing slopes. You discovered that you can trade off some velocity for some torque, and that this happens along rules similar to those that apply to levers: the larger the distance from the fulcrum, the greater the resulting force.

The output torque of a system, when not properly directed to the exertion of work, or when something goes wrong in the mechanism itself, can cause damage to your LEGO parts. You learned that the clutch gear is a precious tool to limit and control the maximum torque so as to prevent any possible harm.

Gears are not the only way to transfer power; we showed that pulley-belt systems, as well as chains, may serve the same purpose and help you in connecting distant systems. Belts provide an intrinsic torque-limiting function and do well in high-speed low-torque situations. Chains, on the other hand, don't limit torque but do increase friction, so they are more suitable for transferring power at slow speeds.

Last but not least, you explored the surprising properties of the differential gear, an amazing device that can connect two wheels so they rotate when its body rotates, still allowing them to turn independently. The differential gear has some other applications, too, since it works like an adder-subtracter that can return the algebraic sum of its inputs.

If these topics were new to you, we strongly suggest you experiment with them before designing your first robot from scratch. Take a bunch of gears and axles and play with them until you feel at ease with the main connection schemes and their properties. This will offer you the opportunity to apply some of the concepts you learned from Chapter 1 about bracing layers with vertical beams to make them more solid (when you increase torque, many designs fall apart unless properly reinforced). You won't regret the time spent learning and building on this knowledge. It will pay off, with interest, when you later face more complex projects.



Chapter 3

Controlling Motors

Solutions in this chapter:

- **Pacing, Trotting, and Galloping**
- **Mounting Motors**
- **Wiring Motors**
- **Controlling Power**
- **Coupling Motors**

Introduction

Motors will be your primary source of power. Your robots will use them to move around, lift loads, operate arms, grab objects, pump air, and perform any other task that requires power. There are different kinds of electric motors, all of them sharing the property of converting *electrical energy* into *mechanical energy*. In this chapter, we will survey different kinds of LEGO motors and will discuss how to use, mount, connect, and combine them.

Before entering the world of motors, we would like to introduce you to some basic concepts about electricity. There's a very important distinction you should be aware of concerning electrical current: There are two types, *alternating current* (AC) and *direct current* (DC). Alternating current is the type of electricity that comes out of the wall outlets in your house, while batteries are the most typical source of direct current. All the electric LEGO devices, including motors, work with DC only.

To understand what DC is, imagine a stream of water going down a hill. Electricity flowing through a wire is not very different: When you connect a battery to a device like a lamp or a motor, you enable a circuit through which electricity flows more or less like water in a stream. You know that batteries have positive (+) and negative (–) signs stamped on them: they indicate the direction of the flow, which goes from minus to plus, as if the minus pole were the top of the hill. You can place a water mill along the stream to convert the energy of water into mechanical energy; similarly, an electric motor converts an electric flow into motion. What would happen to the water mill if you could reverse the direction of the stream? It would change its direction of rotation. The same happens to DC motors. Every motor has two connectors, one to attach to the negative pole and the other to connect to the positive end of a DC source. You can imagine the current flowing from the negative pole of the battery into the motor, making it move and then coming out again to return to the positive pole of the battery. If you reverse the *polarity*, that is, if you swap the wires between the motor and the battery, you will change the direction of the stream and thus the direction of the motor.

Continuing with our hydraulics metaphor, how would you describe the *quantity* of water that's flowing in a stream? It depends on two factors: the speed of the water, and the width of the stream. Both of them have an influence on the kind of work your mill can perform. In the realm of electricity, the speed of the stream is called *voltage*, and its width (its intensity) is called *current*. They are respectively expressed in Volts (V) and Ampere (A), or sometimes in their submultiples, *millivolt* (mV) and *milliampere* (mA). The amount of work that an electrical flow can perform, for example through a motor, depends on both these quantities. To be more precise, it depends on their product, called *power*, and is measured in Watts (W).

Every motor is designed to run at a specific voltage, but they are very tolerant when it comes to decreases in the supplied voltage. They simply turn more slowly. However, if you *increase* the voltage above the specific limit for a motor, you stand a good chance of burning it out.

Current has a different behavior. It's the motor that "decides" how much current to draw according to the work it's doing: the higher the load, the greater the current. The situation you should avoid at all costs when working with your RCX is to have the motor *stall* (it is connected to the power source but something prevents it from turning). What happens in this case is that the motor tries to win out against the resistance, drawing in more current so it can convert it into power, but as it doesn't succeed in the task, all that current becomes *thermal energy* instead of mechanical energy—in other words, heat. This is the most dangerous condition for an electric motor. And here is where the clutch gear described in Chapter 2 comes into play, limiting the maximum torque and thus preventing stall situations. You will discover later in the chapter that the RCX also has an active role in protecting your motors from dangerous draws of current.

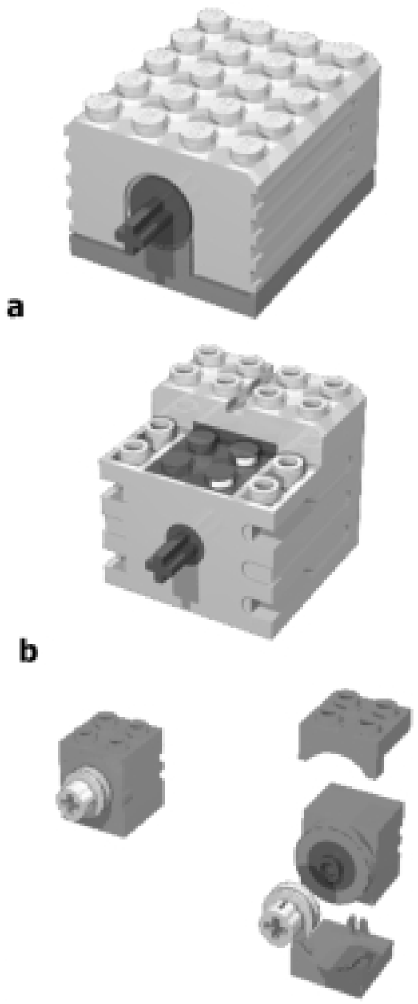
Pacing, Trotting, and Galloping

Every motor contains one or more coils and permanent magnets that convert electrical energy into mechanical energy, but you don't really need to know this level of detail. What you, as a robot builder, must remember is that every motor has a connector through which you can supply it energy, and an output *shaft* which draws the power. The current LEGO TECHNIC line includes three types of 9V DC motors (as shown in Figure 3.1): the ungeared motor (a), the geared motor (b), and the micromotor (c). There are other special motors as well: the train motor, the geared motor with battery pack, and the Micro Scout unit. These are less common, less useful, and less versatile to robotics than the first three, so we won't be examining them here. Table 3.1 summarizes the properties of these three motors.

Table 3.1 Properties of the LEGO TECHNIC Motors

Properties	Ungeared Motor	Geared Motor	Micromotor
Maximum voltage	9V DC	9V DC	9V DC
Minimum current (no load)	100 mA	10 mA	5 mA
Maximum current (stall)	450 mA	250 mA	90 mA
Maximum speed (no load)	4000 rpm	350 rpm	30 rpm
Speed under typical load	2500 rpm	200 rpm	25 rpm

Figure 3.1 The LEGO TECHNIC Motors



The ungeared motor (a) has been the standard LEGO TECHNIC motor for a long time. Its axle is simply an extension of the inner electric motor shaft, and for this reason we called it *ungeared*. Electric motors usually rotate at very high speeds, and this one is no exception, turning at more then 4000 rpm (revolutions per minute). This makes this motor a bit tricky to use, because it requires very high reduction ratios for almost any practical application, leading to very cumbersome and complex geartrains. Add the fact that it draws an amazing amount of current, and you get a pretty good picture of how difficult it can be.

This motor is still easy to find in the shops of many countries as an expansion pack (8720), but we strongly advise you against buying one for the reasons mentioned in the previous paragraph. In this book, you won't find any example that includes the ungeared motor. Nevertheless, if you already have one, you can safely use it; it won't damage your RCX or be damaged itself. The only risk you're taking is that, under heavy loads or stall situations, it drains your batteries very quickly.

The geared motor (b) is what we will generically refer to as a *motor*, the one we will use extensively in the following chapters. It features an internal multistage reduction geartrain and turns at about 350 rpm with no load (typically 200/250rpm with medium load). It's much more efficient than the older kind, and has low current consumption. It also uses more compact geartrains. If you have the MINDSTORMS kit, you already have two of these.

Bricks and Chips…

How to Release a Jammed Micromotor

A micromotor jams so easily, you should know what to do when it occurs. The following list should help:

1. Switch off the motor as soon as you can. Detach the cord or switch the power off; it's important not to leave a stalled motor under power for a long time because that could permanently damage it.
2. Decouple the motor from any connection (gearings, pulleys, and so on). Leave the small pulley attached to the motor shaft.
3. Holding the motor with your fingers, turn the pulley gently but firmly in the same direction the motor was turning when

Continued

it jammed. At the same time, push the pulley against the motor until you hear a click. Your motor should be okay now. If you don't know what direction the motor was rotating when jammed, try both directions.

This procedure usually works. If it doesn't, try to power on the motor in both directions with very brief current pulses, at the same time doing what's described in Step 3.

The micromotor (c) is a geared motor as well. It's geared down so much that its output shaft turns at approximately 30 rpm. Nevertheless, its torque is incredibly low, well below 1 Ncm. It is also surprisingly noisy, and very easy to jam. At this point, you might wonder why you should ever consider this motor, but the answer lies in its name: because it's micro. There are situations where the size of the motor is more critical than the amount of torque and speed needed. To be used, it requires some special mounting brackets, and a small pulley to connect to its shaft (Figure 3.1c).

Mounting Motors

The LEGO motor is 4 studs wide and 4 studs long, and has an irregularly shaped top with a height of 2.33 bricks in the low part and 3 bricks in the high one. The base of the motor is irregular too, because there's a convex area of 2 x 2 that makes a direct mount of the motor on a regular surface impossible. For these reasons mounting LEGO motors requires some experience. In the following paragraphs, we'll show a few common solutions, but many others will work as well.

Despite its irregular shape, the motor fits well enough in the standard grid. In Figure 3.2, you see that its lower part can be locked between two beams at a distance of four holes. It is very important that you build your motors inside a solid structure, otherwise they will become loose when you apply a load. You can also see in the figure that the shaft of the motor is two holes from the bottom beam, which is perfect for some of the gearing combinations discussed in the previous chapter: 8t to 24t, or 16t to 16t, to name a few.

In our second example (displayed in Figure 3.3), we show another very solid assembly. This time we extended the output axle of the motor in order to mount a worm gear on it so it can mesh with a 24t. While the previous case was suitable to drive wheels from the 24t axle, this would fit a slow speed/high torque application.

Figure 3.2 Locking a Motor between Beams

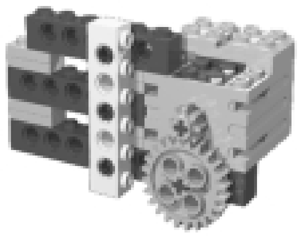
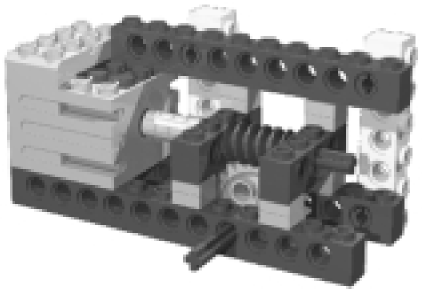


Figure 3.3 A Motor Connected to a Worm Gear



NOTE

The pictures here are mainly meant to highlight relations and distances. So, in order to let you see inside, we didn't lock the motor on both sides. In actual applications, you will complete the assembly and adjust the beams to the proper length for your needs.

Your MINDSTORMS kit contains eight 1 x 2 plates with rails which are specifically designed as brackets for the motors (Figure 3.4). They permit compact and solid attachments like the one in Figure 3.5. But even more importantly, they give you the ability to remove motors without dismantling your robot. In the example in Figure 3.5, if you remove the two 2 x 6 plates behind the motor, you can easily slip it off without altering the rest of the assembly. This is a very desirable property, allowing you to recycle your motors in other projects without being forced to take your robot apart. You will likely end up having more LEGO parts than those contained in your MINDSTORMS kit, so it's possible you'll have more than a single assembled robot at one time. Motors are among the most expensive LEGO components. Reusing them in different projects will help keep the cost of your hobby at a reasonable level!

Figure 3.4 1 x 2 Plates with Rails Provide a Convenient Mounting Solution

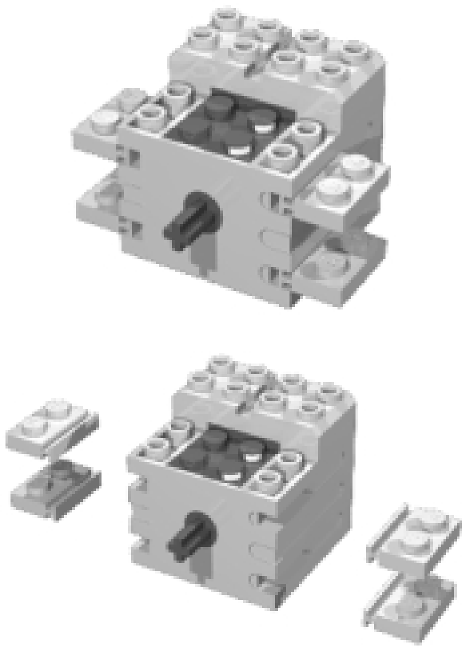
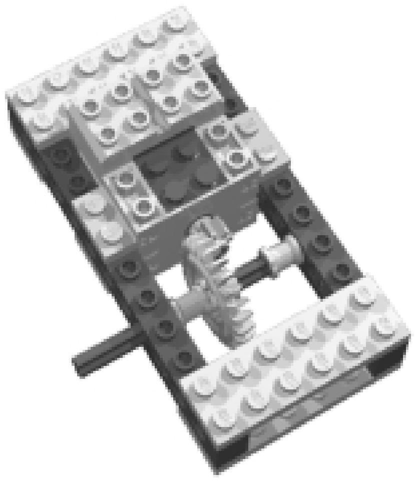


Figure 3.5 An Easily Removable Motor

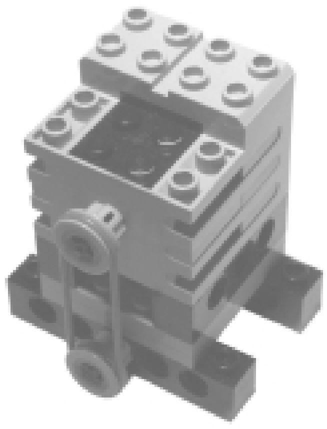


NOTE

We suggest that, when mounting motors, you keep the wire free to be removed. Don't block it together with the motor, unless you're sure your design won't change and you won't need a wire of different length.

Figure 3.6 illustrates our last example. You can see how two pulleys and a belt may solve the problem of transferring power to a distant axle through a narrow space. In this particular example, the motor does not need to be locked with a vertical beam because the torque on its shaft won't ever reach high values (belt slippage prevents this from happening). At the same time, the belt works like a rubber band, too, keeping the motor from coming off its foundation.

Figure 3.6 Belts Don't Require Very Solid Mountings



Wiring Motors

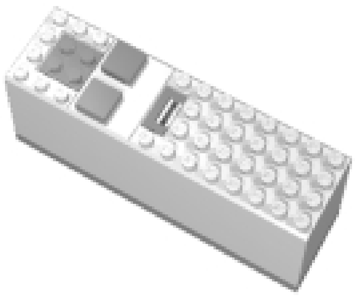
The LEGO wiring system is so easy to use you won't require any training. The cables end with 2 x 2 x 2/3 connectors that attach as easily as standard bricks and don't need any special knowledge to be used.

As we already explained, LEGO motors are DC motors, therefore they are sensitive to the *polarity* you connect them with, meaning it determines whether the motor turns clockwise or counterclockwise. Usually, you don't have to worry about this, since you can control this property from your program. However, the design of the LEGO connectors is very clever and not only prevents you from involuntarily short-circuiting the motor or the battery, but they also allow you to reverse the polarity by simply turning them 180 degrees.

How can you test your motors without adjusting programming? There are many different ways, as in the following:

- **RCX console** Press the **View** button until you select the port your motor is wired to. When the cursor (a small arrow) points to the proper port, don't release the button. Keeping the **View** button pressed, you can press **Prgm** or **Run** to power the motor in the desired direction.
- **Software** Browsing the Internet you can find and download many good freeware programs that allow full direct control of your RCX via your PC. They make running a motor as easy as a click of the mouse (see Appendix A for links and resources).
- **External battery box** Some LEGO TECHNIC sets include a battery box (Figure 3.7). If you want an extra motor and buy an 8735 TECHNIC Motor set, you'll get one. With this box you can test your motor with no need of the RCX.

Figure 3.7 The LEGO Battery Box



- **Remote control** This useful tool is not included in the MIND-STORMS kit, you have to buy it separately (Figure 3.8). It's currently sold inside the Ultimate Accessory Set that also contains additional parts. If you can afford it, it's a good buy. You can control all three output ports at the same time, which is very useful when testing your robot during the building phase.
- **Other sources** All the components of the LEGO 9V electric system are compatible with each other. If you have a LEGO train speed regulator, or

a Control Center unit, you can safely use them to run your motors. Don't use non-LEGO electricity sources. They might harm your motors.

Figure 3.8 The LEGO MINDSTORMS Kit Remote Control

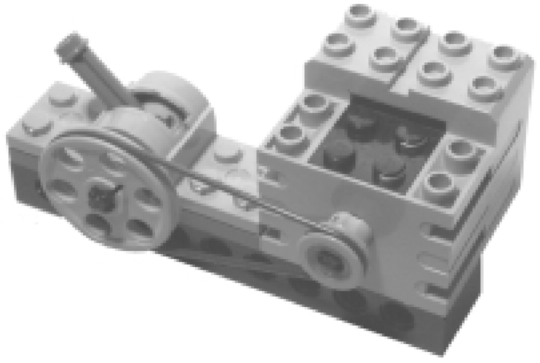
In some cases, you want to control more than a single motor from the same RCX output port. Is this safe for your RCX and your motors? Yes, and with no risk of damaging either item. The only thing to point out is that the RCX has a current-limiting device behind each port that prevents your motor from drawing too much current to avoid any possible damage during stall situations. When you connect two or more motors to the same port, they must share the maximum available current, thus limiting the work they can perform. Nevertheless, there are situations where splitting the load on two or more motors is the preferable option.

There is another possible approach that bypasses the current-limiting circuit: *indirect control*. Instead of supplying the motors from your RCX port, you control a motor that activates a switch that turns on the other motors. This sounds complicated, but it isn't. You just need some extra parts: a polarity switch and a battery box. In Figure 3.9, you see a system devised to drive the LEGO polarity switch with a motor and two pulleys. The belt coupling makes the system less critical about timing. If you accidentally power the controlling motor for longer than what's needed to activate the switch, the belt slips and your motor doesn't stall.

The polarity switch is actually a three-state switch: *forward*, *off*, and *reverse*. At one side, it switches the motors on, in the center it switches them off, while on the other side it switches them on again but with reversed polarity. Our simple assembly can control only two states (don't rely on timing to position the polarity switch precisely in the center!), so you have to choose whether you want an on/off system or a forward/reverse one.

As the battery box does not feature any current limiting device, your motors can draw as much current as they need out of the batteries. Remember that with this wiring the controlled motors are not protected against overloads, thus stall situations might permanently damage them.

Figure 3.9 Indirect Motor Control



Controlling Power

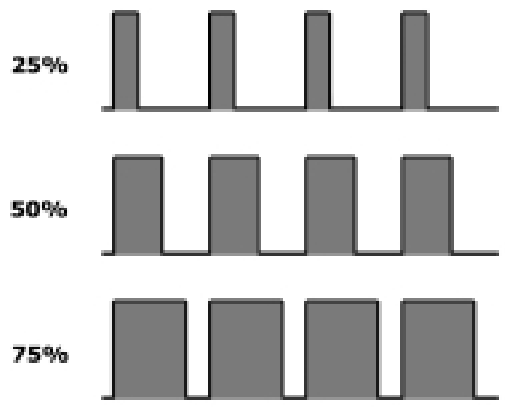
You know that your program can control the power of your motors. In fact, a specific instruction will set the power level in the range 0 to 7 (some alternative firmware, like legOS, provide higher granularity, e.g., 0 to 255). But what happens when you change this number? And why do we care?

There are different ways to control the power of an electric motor. The LEGO train speed regulator controls power through voltage: the higher the voltage, the higher the power. The RCX uses a different approach, called *pulse width modulation* (PWM).

To explain how this works, imagine that you continuously and rapidly switch your motor on and off. The power your motor produces in any given interval depends on how long it's been on in that period. Applying current for a short period of time (a *low duty cycle*) will do less work than applying it for a longer time. If you could switch it on and off hundreds of times a second, you would see the motor turning in an apparently normal way; but under load you would notice a decrease in its speed, due to a decrease in the supplied power (Figure 3.10).

This is exactly what the RCX does. Its internal motor controller can switch the power on and off very quickly (an on/off cycle every 8 milliseconds), at the same time varying the proportion between the on period and the off period. At power level 0, the motor is on for 1/8 of the cycle; at power level 1, for 2/8 of it; and so on until you reach level 7, when the motor is always on (8/8).

Figure 3.10 Pulse Width Modulation Power Levels



Why do we care about this technical stuff? Because this explains you aren't actually controlling speed, but power. LEGO motors are very efficient, and when the motor has no load or a very small one, lowering the power level won't decrease its speed very much. Under more load, you will see how the power level affects the resulting speed, too.

Braking the Motor

Controlling the power means also being able to brake your motor when necessary. For this purpose, the RCX features a sort of electric brake. Once again, let us explain how it works through an experiment.

You need a motor, a cable (any length), and a 24t gear. Assemble the three as shown in Figure 3.11, paying attention to the way the cable is looped: the ends of the wire go on opposite sides. Now try and turn the 24t with your fingers: it turns smoothly, and continues to spin for a while after you've stopped turning it.

Then remove the cable and reconnect it as shown in Figure 3.12: the ends of the wire go into the same side—this way the motor is short-circuited. We know that a *short circuit* sounds like a *bad* thing, but in this particular case we mean only that the circuit is *closed*. Don't worry, your motor is not at any risk. Now try and turn the 24t again. You see? The motor offers a lot of resistance, and as soon as you stop turning, it stops, too.

What happened? A LEGO motor is not only able to transform electricity into motion, it does the opposite, too: It can be used to *generate* electricity. In our experiment the generated current short-circuits back into the motor, producing the force that resists the motion. This is the simple but effective system the RCX implements to brake the motor: When you set them to *off*, the RCX not only switches the power off, it also short-circuits the port, making the motor brake.

Figure 3.11 In This Setup, the Motor Shaft Turns Smoothly

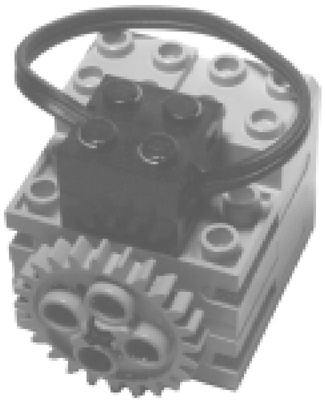
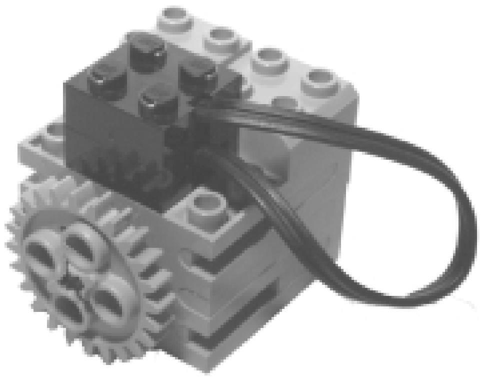


Figure 3.12 An Electric Brake



There's another condition, called *float* mode, where the RCX simply disconnects the motor without creating any brake effect. In this case, the motor will continue to turn for a few seconds after the power has been removed.

Bricks & Chips…

Using Motors as Generators

If you are not convinced that a motor works as a generator, too, perform this simple experiment. Connect one motor to another with a wire. Place a 24t on each shaft. Take one motor in your hands and turn the 24t while looking at the second motor. What happens? The first motor converts the mechanical energy coming from your fingers into electric current, which makes the second motor turn.

Coupling Motors

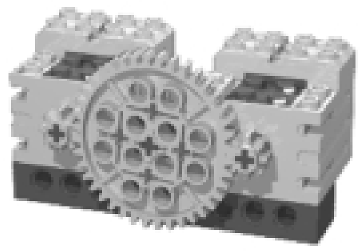
We previously discussed the case in which you want to wire two motors to the same port. If you do this to get more power for a task, you will very likely need to mechanically *couple* the motors as well, meaning that they will work together to operate the same mechanism, sharing its load. It's like when you have to move something really heavy and call a friend to help you: each member of the party bears only half the total weight. Though this rule works for all electric motors in general, a specific limitation applies when attaching LEGO motors to the RCX: Its current-limiting device won't allow the motors to draw as much current as they want. Consider it a constraint to the maximum power each port can pay out.

In Figure 3.13, you see two motors acting upon the same 40t gear wheel. People often wonder whether connections like these are going to cause any problem to the motors. The answer is simply *no*. Unless you keep your motor stalled for more than a brief moment, they are not easy to damage. In applications like the one in Figure 3.13, you just have to be sure the motors don't oppose each other. With this in mind, we suggest you double-check both the connection and turning directions before actually coupling the motors to the same gear.

It is true that no two motors turn exactly at the same speed, or output the same torque either, but this doesn't cause any conflict. A motor doesn't *know* that there's another motor cooperating on the same task, it simply reacts to the load absorbing more current and trying to keep the speed. This works even if the motors are of different types, even if they are powered at different levels, and even if they are geared with different ratios.

If you're not convinced of this, think of a simple vehicle propelled by a single motor. When the path becomes steeper, the load on the motor increases, causing it to reduce its speed. Essentially, the motor adapts itself to the load. The same happens when two motors work together, they share the load and mutually adapt themselves.

Figure 3.13 Two Mechanically Coupled Motors



Have you ever tried riding a tandem bicycle? Your partner might be much weaker than you, but you would prefer him to pedal rather than simply ride along watching the landscape.

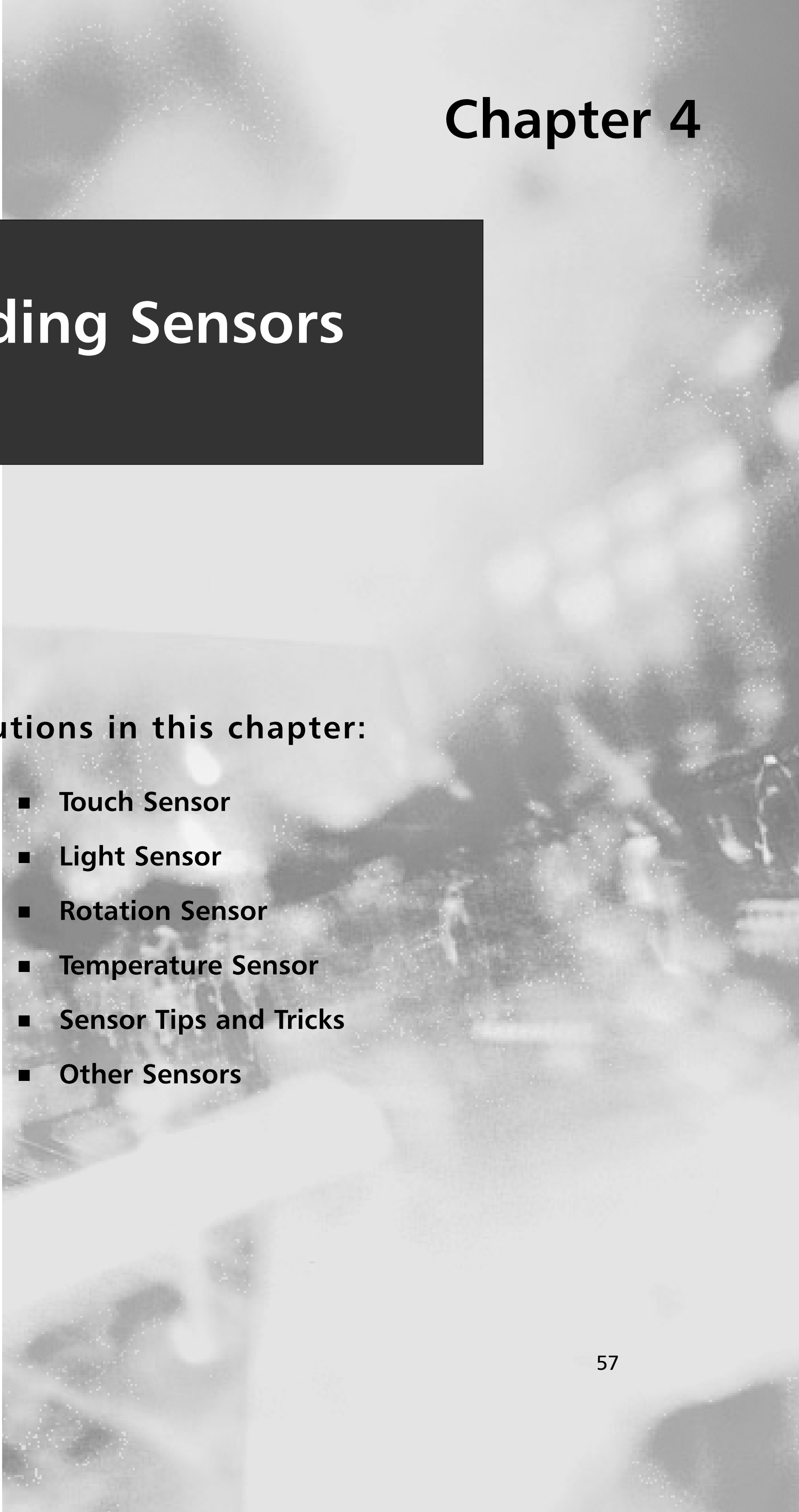
Summary

LEGO electric motors are easy and safe to use, but they require a bit of experience to get the most from them and avoid any possible damage. On this latter topic, the most important thing is to never let them stall for more than a few seconds and to never keep them powered when they've stalled. You already know from Chapter 2 that the clutch gear is a good ally in this venture, and you've now learned that the RCX has further protections that limit the maximum current and thus the risk that your motor will burn out.

You've seen that wiring LEGO motors is very simple: The special connectors prevent short circuits and allow easy control of polarity, which affects the direction in which a motor turns. The different mounting options require a bit of practice, the same as for gears. Don't forget to brace motors with vertical beams the way you were taught in Chapter 1: They produce enough torque to pull themselves apart if not solidly locked!

On the topic of coupling motors, this option is useful when you want to split a load over two or more of them to reduce their individual effort. The only important thing to remember is that you must control them from the same port, so as to avoid any dangerous conflict situation where one motor opposes to the other.

As a general tip, we suggest you make intense use of prototyping—don't wait to finish your robot to discover a motor is in the wrong place or has not been geared properly—test your mechanisms while you are building them.



Chapter 4

Reading Sensors

Solutions in this chapter:

- Touch Sensor
- Light Sensor
- Rotation Sensor
- Temperature Sensor
- Sensor Tips and Tricks
- Other Sensors

Introduction

Motors, through gears and pulleys, provide motion to your robot; they are the muscles that move its legs and arms. The time has come to equip your creature with *sensors*, which will act as its eyes, ears, and fingers.

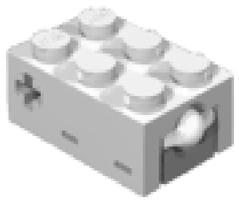
The MINDSTORMS box contains two types of sensors: the *touch sensor* (two of them) and the *light sensor*. In this chapter, we'll describe their peculiarities, and those of the optional sensors that you can buy separately: the *rotation sensor* and the *temperature sensor*. All these devices have been designed for a specific purpose, but you'll be surprised at their versatility and the wide range of situations they can manage. We will also cover the cases where one type of sensor can *emulate* another, which will help you replace those that aren't available. Using a little trick that takes advantage of the infrared (IR) light on the RCX, you will also discover how to turn your light sensor into a sort of radar.

We invite you to keep your MINDSTORMS set by your side while reading the chapter, so you can play with the real thing and replicate our experiments. For the sake of completeness, we'll describe some parts that come from MIND-STORMS expansion sets or TECHNIC sets. Don't worry if you don't have them now; this won't compromise your chances to build great robots.

Touch Sensor

The *touch* sensor (Figure 4.1) is probably the simplest and most intuitive member of the LEGO sensor family. It works more or less like the push button portion of your doorbell: when you press it, a circuit is completed and electricity flows through it. The RCX is able to detect this flow, and your program can read the state of the touch sensor, **on** or **off**.

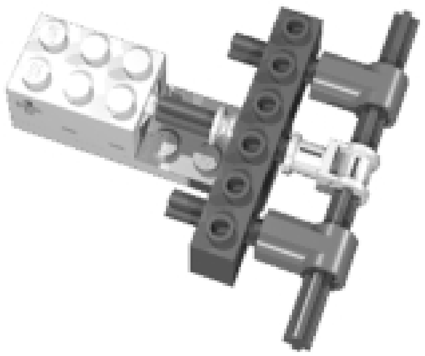
Figure 4.1 The Touch Sensor



If you have already played with your RIS, read the Constructopedia, and built some of the models, you're probably familiar with the sensors' most common application, as *bumpers*. Bumpers are a simple way of interacting with the environment; they allow your robot to detect obstacles when it hits them, and to change its behavior accordingly.

A bumper typically is a lightweight mobile structure that actually hits the obstacles and transmits this impact to a touch sensor, closing it. You can invent many types of bumpers, but their structure should reflect both the shape of your robot as well as the shape of the obstacles it will meet in its environment. A very simple bumper, like the one in Figure 4.2, could be perfectly okay for detecting walls, but might not work as expected in a room with complex obstacles, like chairs. In such cases, we suggest you proceed by experimenting. Design a tentative bumper for your robot and move it around your room at the proper height from the floor, checking to see if it's able to detect all the possible collisions. If your bumper has a large structure, don't take it for granted that it will impact the obstacle in its optimal position to press the sensor. Our example in Figure 4.2 is actually a bad bumper, because when contact occurs, it hardly closes the touch sensors at the very end of the traverse axle. It's also a bad bumper because it transmits the entire force of the collision straight to the switch, meaning an extremely solid bracing would be necessary to keep the sensor mounted on the robot.

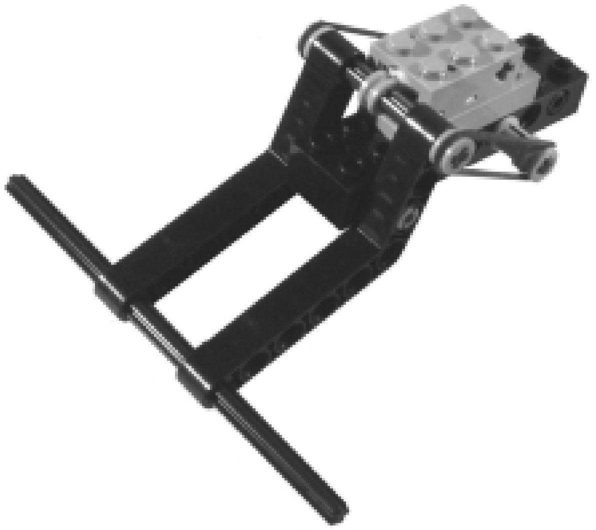
Figure 4.2 A Simple Bumper



Be empirical, try different possible collisions to see if your bumper works properly in any situation. You can write a very short program that loops forever, producing a beep when the sensor closes, and use it to test your bumper.

When talking of bumpers, people tend to think they should *press* the switch when an obstacle gets hit. But this is not necessarily true. They could also *release* the switch during a collision. Look at Figure 4.3, the rubber bands keep the bumper gently pressed against the sensor; when the front part of the bumper touches something, the switch gets released.

Figure 4.3 A Normally Closed Bumper



Actually, there are some important reasons to prefer this kind of bumper:

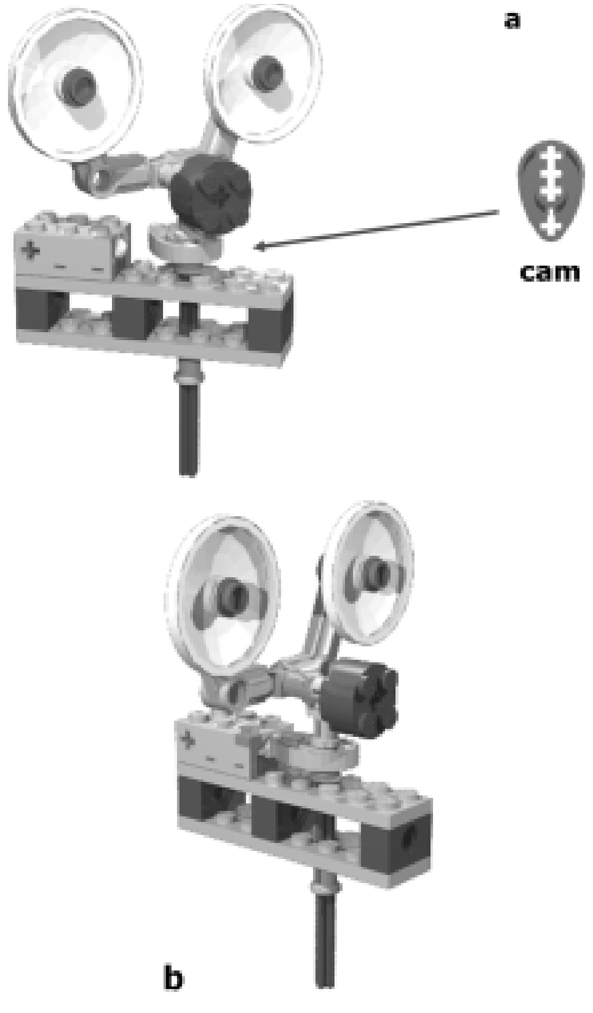
- The impact force doesn't transfer to the sensors itself. Sensors are a bit more delicate than standard LEGO bricks and you should avoid shocking them unnecessarily.
- The rubber bands absorbing the force of the impact preserve not only your sensor but the whole body of your robot. This is especially important when your robot is very fast, very heavy, very slow in reacting, or possesses a combination of these factors.

Bumpers are a very important topic, but touch sensors have an incredible range of other applications. You can use them like buttons to be pushed manually when you want to inform your RCX of a particular event. Can you think of a possible case? Actually, there are many. For example, you could push a button to order your RCX to "read the value of the light sensor *now*," and thus calibrate readings (we will discuss this topic later). Or you could use two buttons to give feedback to a learning robot about its behavior, *good* or *bad*. The list could be long.

Another very common task you'll demand from your switch sensors is *position control*. You see an example of this in Figure 4.4. The rotating head of our robot

(Figure 4.4a) mounts a switch sensor that closes when the head looks straight ahead (Figure 4.4b). Your software can rely on timing to rotate the head at some level (right or left), but it can always drive back the head precisely in the center simply waiting for the sensor to close. By the way, the *cam* piece we used in this example is really useful when working with touch sensors, as its three half-spaced crossed holes allow you to set the proper distance to close the sensor.

Figure 4.4 Position Control with a Touch Sensor



There would be many other possible applications in regards to position control. We'll meet some of them in the third part of this book. What matters here is to invite you to explore many different approaches before actually building your robot. Let's create another example to clarify what we mean. Suppose you're going to build an elevator. You obviously want your elevator to stop at any floor. Your first idea is to put a switch at every level, so when one of them closes you know that the cab has reached that level. Okay, nice approach. There's one small problem; however, you have just two touch sensors, and an elevator with only two floors doesn't seem like such an interesting project to you. You could buy a third sensor, but this simply pushes your problem one floor up, without solving the general case. Meanwhile, the three input ports of your RCX are all engaged. Suddenly, an idea occurs to you: Why not put the sensor on the booth instead of on the structure? With a single sensor on the booth, and pegs that close it at any floor, you can provide your elevator with as many floors as you like. You see, by reversing our original approach you found a much better solution. Are the two systems absolutely equivalent? No, they aren't. In the first, you could determine the absolute position of the booth, while in the second you are able to know only its relative position. That is, you do need a known starting point, so you can deduce the position of the cab counting the floors from there. Either require that the cab must be at a specific level when the program starts, or use a second sensor to detect a specific floor. For example, place a sensor at the ground level, so the very first thing your program has to do when started is to lower the elevator until it detects the ground level. From then on, it can rely on the cab sensor to detect its position.

Now your elevator is able to properly navigate up and down. You have one last problem to solve: How do you inform your elevator which floor it should go to? Placing a touch sensor at every floor to *call* the elevator there is impractical. You have only one input port left on your RCX. What could you do with a single sensor? Can you apply the previous approach here, too?

Yes. You can *count* the pushes on a single touch sensor. For example, three clicks means third floor, and so on. Now you are ready to actually build your elevator!

Bricks & Chips…

Counting Clicks

The following examples are written using a *pseudo-code*—that is, a code that does not correspond to any real programming language, but rather lies between a programming language and natural language. Using pseudo-code is a common practice among professional programmers;

Continued

you are "playing computer" and quickly stepping through an operation in your head to plan and understand what your program will do.

Counting how many times a touch sensor is pressed requires some tricks. Suppose you write some simple code, like this:

```
Counter = 0
repeat
  if Sensor1 is on then
    Counter = Counter+1
  end if
end repeat
```

Your code executes so fast on your RCX that during the short instant you keep the touch sensor pressed, it counts too many clicks. Thus, you need to have it wait for the button to be released before counting a new click:

```
Counter = 0
repeat
  if Sensor1 is on then
    Counter = Counter+1
    wait until Sensor1 is off
  end if
end repeat
```

Now, your code counts properly the transitions from off to on. There's one last feature you must introduce in your code: You want the counting procedure to end when it doesn't receive a click for a while. To do this, you employ a timer that measures the elapsed time from the last click:

```
Counter = 0
Interval = <a proper value>
reset Timer
repeat
  if Sensor1 is on then
    Counter = Counter+1
    wait until Sensor1 is off
```

Continued

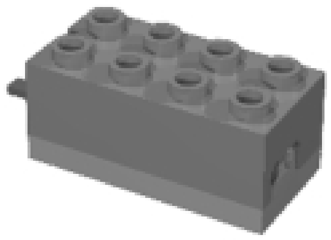
```
           or until Timer is greater then Interval
       reset Timer
     end if
   until Timer is greater then Interval
```

Let's say your interval is two seconds. When the counting procedure begins, it resets the timer and the counter to 0 then starts checking the sensor. If nothing happens in two seconds, it exits the **repeat** group. If a click occurs, it counts it, waits for the user to release the button, and resets the timer so the user has again two seconds for another click before the procedure ends.

Light Sensor

Saying that the *light sensor* (Figure 4.5) "sees" is definitely too strong a statement. What it actually does is detect light and measure its intensity. But in spite of its limitations, you can use it for a broad range of applications.

Figure 4.5 The Light Sensor



The most important difference between the touch sensor and the light sensor, is that the latter returns many possible values instead of a simple on/off state. These values depend on the intensity of the light that hits the sensor at the time you read its value, and they are returned in the form of percentages ranging from 0 to 100. The more light, the higher the percentage. What can you do with such a device? A possible application is to build a light-driven robot, a *light follower* as it's called, that looks around to find a strong (or the strongest) light source and directs itself toward it. Provided that the room is dark enough not to produce interference, you could then control your robot using a flashlight.

This ability to trace an external light source is interesting, but probably not the most amazing thing you can do with this sensor. We introduce here another feature of this device: not only does it detect light, but it *emits* some light as well. There is a small red LED that provides a constant source of light, thus allowing you to measure the reflected light that comes back to the sensor.

When you want to measure reflected light, you must be careful to avoid any possible interference from other sources. Remember that this sensor is very sensitive to IR light, too, like the one typically emitted by remote controls, video cameras, or the LEGO IR tower.

Designing & Planning…

Reading Ambient Light

The LEGO light sensor is actually not a great device to measure external sources, as its sensitivity is too low. The emitting red LED is so close to the detector that it strongly influences the readings. If your target is an external source, you might consider trying to reduce the effect of the emitting LED. A simple solution is to place a 1 x 2 one-hole brick just in front of the light sensor. Much more effective solutions require that you slightly modify your sensor. On his Web site, Ralph Hempel shows how to make modifications that neither permanently alter nor damage your sensor (see Appendix A).

The amount of light reflected by a surface depends on many factors, mainly its color, texture, and its distance from the source. A black object reflects less light than a white one, while a black matte surface reflects less light than a black shiny surface. Plus, the greater the distance of the objects from the sensor, the less light returns to the detector.

These factors are interdependent, meaning that with a simple reading from your light sensor, you cannot tell anything about them. But if you keep all the factors constant except one, you are now able to deduce many things from the readings. For example, if your light sensor always faces the same object, or objects with the same texture and color, you can use it to measure its *relative distance*. On the other hand, you can place different objects in front of the sensor, at a constant distance, to recognize their *color* (or, more accurately, their *reflection*).

Measuring Reflected Light

To illustrate the concept of measuring reflected light, let's prepare an experiment. Take your RCX, turn it on, attach a light sensor to any input port, and configure the port properly using the Test Panel of your MINDSTORMS box (the red LED should illuminate). Prepare the environment. You need a dark room, not necessarily completely dark but there should be as little light as possible. The RCX has a *console* mode that allows you to view the value of a sensor in real time. Press the **View** button on your RCX until a short arrow in the display points to the port the sensor is attached to. The main section of the display shows the value your sensor is reading. Now you can proceed. Put the light sensor on the table. Take some LEGO bricks of different colors and place them one by one at short distances from the sensor (about 0.5 in., or 1 to 1.5 cm). Keep all of them separated from each other at the same distance, and look at the readings. You will notice how different colors reflect a different amount of light (you might want to write down the values on a sheet).

For the second part of the experiment, take the white brick and move it slowly toward the sensor and then away from it, always looking at the values in the display. You see how the values decrease when you increase the distance. You can find a distance where the white brick reads the same value you have read for the black one at a shorter distance. This is what we meant to prove: You cannot tell the distance *and* the color at the same time, but if you know that one of the properties doesn't change, you can calculate the other. We stress again that in both cases you must do your best to shield your system from ambient light.

Bricks & Chips…

Understanding Raw Values

Understanding raw values is an advanced topic, and not strictly necessary to successfully using the MINDSTORMS system. That said, it does help to understand how to work with sensors.

The RCX converts the electrical signals coming from sensors (of any type) into whole numbers in the range of 0 to 1023, called *raw values*. When, in your program, you configure a port to host a specific kind of sensor, the RCX automatically scales raw values to a different range,

Continued

suitable for that particular kind of sensor. For example, readings from touch sensors become a simple 1 or 0 digit, meaning on or off, while readings from a temperature sensor convert into Celsius or Fahrenheit degrees. Similarly, light sensor readings are converted into percentages through use of the following equation:

Percentage = 146 - raw value / 7

Why should you need to know about this conversion? Well, for most applications the percentage light value returned by the RCX works well, but there are situations where you need all the possible resolution your sensor can provide, and this conversion into percentages masks some of the resolution your light sensor is capable of. Let's explain this with an example. Suppose that, in two different conditions, your light sensor returns raw values of 707 and 713. Convert these numbers into percentages, considering that RCX uses whole numbers only, and thus rounds the result of a division to the previous integer:

146 - (707 / 7) = 146 - 101 = 45

146 - (713 / 7) = 146 - 101 = 45

The 101 in the second equation should have been 101.857…, but it's been truncated to 101, and you lost the difference between the two readings. We agree that in most situations this granularity of readings is not very important, but there are others where even such a small interval matters.

If you program your RCX using RCX Code, the graphic LEGO environment, you must accept the scaled values, because you have no way to access raw values. But if you use *alternative* programming tools you can choose to receive the unprocessed raw values directly, taking advantage, when necessary, of their finer resolution.

Reading colors is a very common application for light sensors. We already explained that the sensor doesn't actually read colors, rather it reads the reflected light. For this reason, it's hard to tell a black brick from a blue one. But, for now, let's continue to use the expression *reading colors*, now that you know what's really behind the reading.

Line Following

Probably the most widespread usage of the light sensor is to make the robot read lines or marks on the floor where it moves. This is a way to provide artificial landmarks your robot can rely on to navigate its environment. The simplest case is *line following*. The setup for this project is very simple, which is one of the reasons it's so popular. Despite its apparent simplicity, this task deserves a lot of attention and requires careful design and programming. We will discuss this topic in greater detail in Part II; for now, though, we want to bring your attention to what happens when the light sensor "reads" a black line on a light floor.

When the sensor is on the floor, it returns, let's say, 70 percent, while on the black line, it returns 30 percent. If you move it slowly from the floor to the line or vice versa, you notice that the readings don't leap all of a sudden from one to the other, they go through a series of intermediate values. This happens because the sensor doesn't read a single point, but a small area in front of it. So when the sensor is exactly over the borderline, it reads half the floor and half the black strip, returning an intermediate result.

Is this feature useful? Well, sometimes it is, sometimes it's not. When dealing with line following in particular, it is *very* useful. In fact, you can (and should) program your robot to follow the "gray" area along the borderline rather than the actual black line. This way when the robot needs to correct its course, it knows which direction to turn: If it reads too "dark," it should turn toward the "light" region, and vice versa.

Designing & Planning…

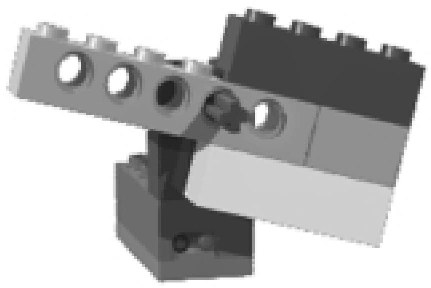
Calibrating Readings

Sometimes you can't know in advance what actual values your sensor is going to read. Suppose you're going to attend a line following contest: You cannot be sure of the values your sensor will return for the floor and the black line. In this case, and as a good general practice, it is better not to write the expected values as constants in your program, but allow your robot to read them by itself through a simple calibration procedure. Staying with the line following example, you can dedicate a free input port to a touch sensor to be manually pressed when you put your robot on the floor and then on the black line, so it can store the maximum and minimum readings. Or you can program the robot to perform a short exploration tour to uncover those limits itself.

When you need to navigate a more complex area, one, for example, that includes regions of three different colors, things get more difficult. Imagine a pad divided into three fields: white, black, and gray. How can you tell the gray area from the borderline between the white and the black? You can't, not from a single reading, anyway. You must take into consideration other factors, like previous readings, or you can make your robot turn in place to make it gather more information and understand where it is. To handle a situation like this, your software is required to become much more sophisticated.

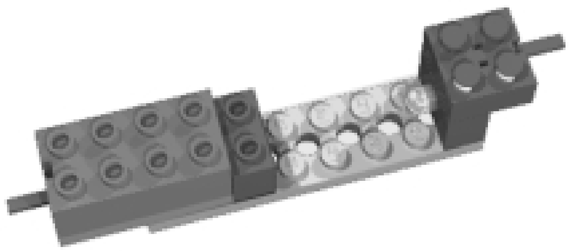
The light sensor is such a versatile device that you can imagine many other ways to employ it. You can build a form of proportional control by placing a multicolor movable block of LEGO parts in front of it. Figure 4.6 shows an example of this kind. When you push or pull the upper side of the beam, the sensor reads different light intensities.

Figure 4.6 An Analogue Control with a Light Sensor



Combining the light sensor with a lamp brick (not included in the MINDSTORMS kit) you get a photoelectric cell (Figure 4.7); your robot can detect when something interrupts the beam from the lamp to the sensor. Notice that we placed a 1 x 2 one-hole beam in front of the light sensor to reduce the possible interference from ambient light.

Figure 4.7 A Photoelectric Cell



Proximity Detection

You can also use the light sensor as a sort of radar to detect obstacles *before* your robot hits them. This is called *proximity detection*. The technique is based on a property we have already discussed and explored: that the light sensor can be used to measure relative distances based on reflected light. Suppose your robot is going straight, with a light sensor pointing ahead of it. Suppose also that your robot moves in a dark room, with no other sources of light except the emitting red LED of the sensor. While moving forward, the robot continuously reads the sensor. If the readings tend to increase rapidly, you can deduce that the robot is going toward something. There's nothing you can tell about the *nature* of the obstacle and its distance, but if nothing else moves inside the room, you are sure the robot is getting closer to the obstacle. Great! We now have a system to *avoid* obstacles instead of being limited to detecting them through collisions.

NOTE

The red LED in the sensor emits visible light, while the IR LED in the RCX emits invisible light!

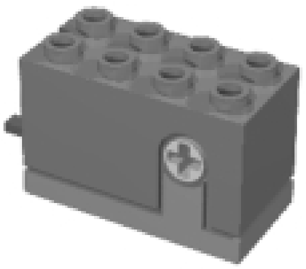
Unfortunately, this technique doesn't work very well in a room with any source of light, because your program cannot tell the difference between its red light reflected back, or any other change in the ambient light. You would need a stronger independent source on the robot to provide a better reference. Thankfully, you just happen to have one! The RCX has an IR LED to send messages to the tower or to another RCX. Sending a message uses the IR LED in the RCX to encode the bits in a format that can be received by the tower. We don't care about the contents of the message; we just want the light. Infrared light, though not visible to the human eye, is of the very same nature as visible light, and the LEGO light sensor happens to be very sensitive to it.

So you now have all the elements to use proximity detection in your programs. Send an IR message and immediately read the light sensor. You had better average some readings to minimize the effect of external sources (we'll discuss this trick in Chapter 12). If you notice a significant increase between two subsequent groups of readings, for example, ten percent, your robot is very likely headed towards an obstacle.

Rotation Sensor

The third LEGO sensor we'll examine is the *rotation sensor* (Figure 4.8). It's a pity this piece of hardware is not included in the MINDSTORMS kit, its versatility being second only to the light sensor. However, there is one included in the 3801 Ultimate Accessory Set, together with a touch sensor, a lamp brick, the remote control, and a few other additional parts.

Figure 4.8 The Rotation Sensor



Bricks & Chips…

How the Rotation Sensor Works

The rotation sensor returns four possible values that correspond to four states, let's call them A, B, C, and D. For every complete turn, it passes through the four states four times—that's why we get 16 counts per turn. Turning the sensor clockwise, it will read the sequence ABCDA..., while turning it counterclockwise will result in the sequence ADCBA.… The RCX polls the sensor frequently, and when it detects that the state has changed, it can not only deduce that the sensor has turned, but also tell in which direction it has turned. For example, transitions from A to B, or from D to A, increment the counter by one unit, whereas transitions from D to C, or from A to D, decrement it by one unit.

The rotation sensor, as its name suggests, detects rotations. Its body has a hole that easily fits a LEGO axle. When connected to the RCX, this sensor counts a unit for every sixteenth of a turn the axle makes. Turning in one direction, the count increases, while turning in the other, the count decreases. This count is *relative* to the starting position of the sensor. When you initialize the sensor, its count is set to zero, and you can reset it again in the code, if necessary.

By counting rotations, you can easily measure position and speed. When connected to the wheels of your robot (or to some gearing that moves them), you can deduce the traveled distance from the number of turns and the circumference of the wheel. Then you can convert the distance into speed, if you want, dividing it by the elapsed time. In fact, the basic equation for distance is:

distance = speed x time

from which you get:

speed = distance / time

If you connect the rotation sensor to any axle between the motor and the wheel, you must remember to apply the proper ratio to the count you read. Let's do an example along with the math together. In your robot, the motor is connected to the main wheels with a 1:3 ratio. The rotation sensor is directly connected to the motor, so it shares the same 1:3 ratio with the wheel, meaning that every three turns of the rotation sensor, the wheels make one turn. Every rotation of the sensor counts 16 units, so 16 x 3 = 48 increments, which correspond to a single turn of the wheel. Now, to calculate the traveled distance we need to know the circumference of the wheel. Luckily, most of the LEGO wheels have their diameter marked on the tire. We had chosen the largest spoked wheel, which is 81.6cm in diameter (LEGO uses the metric system), thus its circumference is 81.6 x $\pi \approx$ 81.6 x 3.14 = 256.22cm. At this point, you have all the elements: the distance traveled by the wheel results from the increment in the rotation counter divided by 48 and then multiplied by 256. Let's summarize the steps. Calling R the resolution of the rotation sensor (the counts per turn) and G, the gear ratio between it and the wheel, we define I as the increment in the rotation count that corresponds to a turn of the wheel:

I = G x R

In our example G is 3, while R is always 16 for LEGO rotation sensors. Thus, we get:

I = 3 x 16 = 48

On each turn, the wheel covers a distance equal to its circumference, C. You can obtain this from its diameter D by using the formula:

$C = D \times \pi$

Which, in our case, means (with some rounding):

$C = 81.6 \times 3.14 = 256.22$

The final step is about converting the reading of the sensor, S, into the traveled distance, T, with the equation:

$T = S \times C / I$

If your sensor reads, for example, 296, you can calculate the corresponding distance:

$T = 296 \times 256.22 / 48 = 1580$

The distance, T, results with the same unit you use to express the diameter of the wheel.

Actually doing this math in your program, even it is nothing more than a division and a multiplication, requires some care (something we will discuss later in Chapter 12).

Controlling your wheels with rotation sensors provides a different way to detect obstacles, a sort of indirect detection. The principle is quite simple: If the motors are on, but the wheels don't rotate, it means your robot got blocked by an obstacle. This technique is very simple to implement, and very effective; the only requirement being that the driving wheels don't slip on the floor (or don't slip too much), otherwise you cannot detect the obstacle. You can avoid this possible problem by connecting the sensor to an idle wheel, one not powered by a motor but instead dragged by the motion across the floor: If it stops while you're powering the driving wheels, you know your motor has stalled.

There are many other situations where the rotation sensor can prove its value, mainly by way of controlling the position of an arm, head, or other movable parts. Unfortunately, the RCX has some problems in detecting precisely any count when the speed is too low or too high. Actually, this is not the fault of the RCX itself, but its *firmware*, which misses some counts if the speed is outside a specific range. Steve Baker proved through an experiment that 50 to 300 rpm is a safe range, with no counts missed between those values. However, in ranges under 12 rpm or over 1400 rpm, the firmware will *surely* miss some counts. The areas between 12 rpm and 50 rpm, and between 300 rpm and 1400 rpm, are in a gray area where your RCX *might* miss some counts.

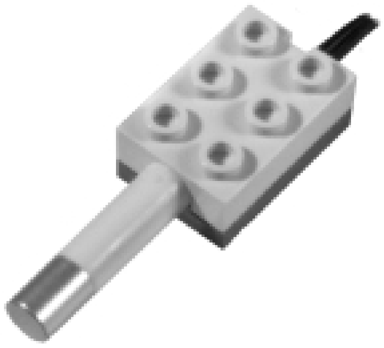
This is a small problem, if you consider that you can often gear your sensor up or down to put it in the proper range.

Temperature Sensor

This is the last sensor of the LEGO MINDSTORMS line. It's an optional sensor, not supplied with the MINDSTORMS kit, but it's easy to get through the LEGO online shop or through their Shop-At-Home service. Let's just say that it's a sensor you can definitely live without, even though it can support some funny projects, like a robot that warns you if your drink is getting too warm or too cold.

There are no movable parts, just a small aluminum cylinder that protrudes from the body of the sensor (Figure 4.9). Depending on how you configure it in your code, you can get the temperature values returned in Celsius or Fahrenheit degrees. It can detect temperatures in the range -20°C to 70°C (-4°F to 158°F), but is very slow in changing from one value to the next, so it's not the best device to use if you're looking to detect sudden changes in the temperature.

Figure 4.9 The Temperature Sensor



NOTE

LEGO sensors come in two families: *active* and *passive* sensors. Passive simply means they don't require any electric supply to work. The touch and temperature sensors belong to the passive class, while the light and rotation sensors are members of the active class.

In case you're wondering how active sensors can be powered through the same wire on which the output returns to the RCX, the answer is that a control circuit cycles between supplying power (for about 3 ms) and reading the value (about 0.1 ms).

The equation used to convert raw values from this sensor into temperatures (in C°) is the following:

C° = (785 – raw value) / 8

Celsius degrees translates into Fahrenheit according to the formula:

F° = C° x 9 / 5 + 32

Sensor Tips and Tricks

Sooner or later you will probably find yourself without the proper sensor for a particular project. For instance, you need three touch sensors, but have only two. Or you need a rotation sensor, but don't have any at all. Is there anything you can do? There's no way to turn any sensor into a light sensor or a temperature sensor, but touch and rotation sensors are at some level replaceable.

Another problem we do battle with every time we build with the RCX is the limited number of ports. Later in the book we'll explore some non-LEGO solutions to this problem, but for now we'll talk about some simple cases where you can connect two or more sensors to the same input port.

In the following sections, you'll find some common and well-tested tips that can help.

Emulating a Touch Sensor

Turning a light sensor into a touch sensor is an easy task; you already know the solution. Basically, you build something similar to what we showed in Figure 4.6. In its default state, the sensor has a LEGO brick just in front of it. The pressure on a lever (or beam, axle, plate, and so on) moves a brick of a different color in front of the light sensor, where your software detects the change. Use a belt to keep your assembly in its default position. Try and protect the light sensor as much as possible from external interferences.

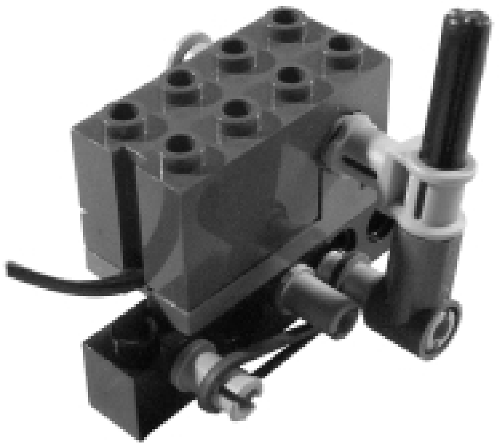
Emulating a touch sensor with a rotation sensor is also doable by building a small actuator that rotates the sensor at least a sixteenth of a turn when touched. One of the many possible approaches is shown in Figure 4.10.

Emulating a Rotation Sensor

There's a long list of possible alternatives to the rotation sensor. All the suggested methods are based upon counting single impulses generated by a rotating part.

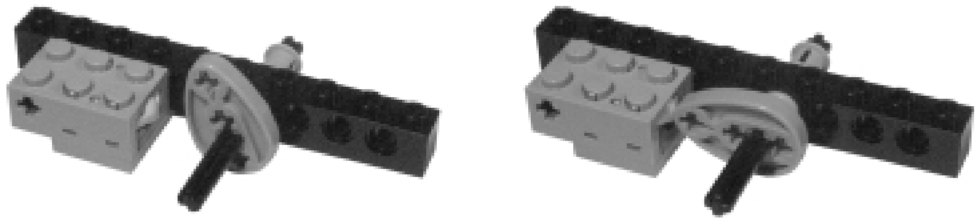
They all work well, but usually they don't detect the direction of rotation. In many cases this is not a problem, because when coupled with a motor you *know* which direction your sensor is moving.

Figure 4.10 Emulating a Touch Sensor with a Rotation Sensor



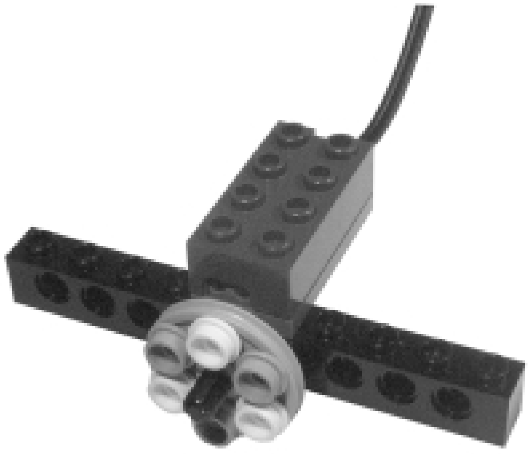
The assembly pictured in Figure 4.11 shows an axle with a cam that closes a touch sensor. This is the principle, use either a cam or any other suitable part that, while rotating, periodically pushes the touch sensor. Counting only a single tick per turn, this sensor has a very low resolution. You can increase it by making the sensor close more than once per turn, or simply by gearing the sensor up a bit until you get the required accuracy.

Figure 4.11 Emulating a Rotation Sensor with a Touch Sensor



Making a rotation sensor out of a light sensor is not very different: build some kind of rotating disk with sectors of different color, and count the transitions from one color to the other (Figure 4.12). The general tip for the light sensor applies to this case, too: Try and insulate the light sensor from external light sources as much as possible.

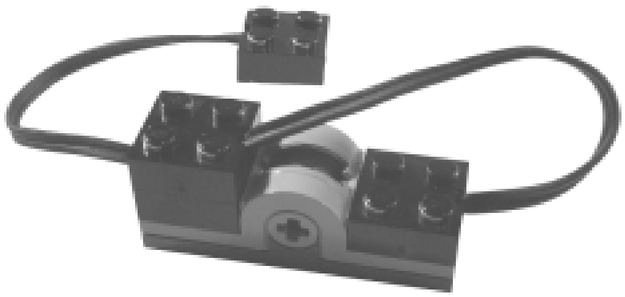
Figure 4.12 Emulating a Rotation Sensor with a Light Sensor



There are two LEGO electric devices that, though not being actual sensors, can be successfully employed to emulate a rotation sensor. None of them is included in the MINDSTORMS kit, but they're not hard to find.

The first is the polarity switch we introduced in Chapter 3. Connect it as shown in Figure 4.13, and configure it as a touch sensor. With every turn, it closes the circuit twice.

Figure 4.13 Emulating a Rotation Sensor with a Polarity Switch

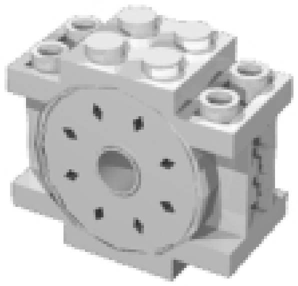


WARNING

When using a polarity switch to emulate a rotation sensor, due to higher friction, the polarity switch cannot rotate as freely as a true rotation sensor.

The second is the Fiber-Optic System (FOS) device (Figure 4.14). Designed to be mainly a decorative item, this unit, when powered, emits a red light, and by rotating it, you can address the light to one of eight possible small holes. Despite its original purpose, it works surprisingly well as a rotation sensor. Try and connect it directly to an input port of your RCX and configure it as a light sensor; rotate it slowly while viewing its values on the display. They swing from about 70 percent down to 2 percent, then back to 70 percent. You can count sixteen transitions per turn. Thus, the resolution is the same as the original rotation sensor. It has a very low friction quotient, too, resulting in an ideal substitute.

Figure 4.14 The Fiber Optic System Unit



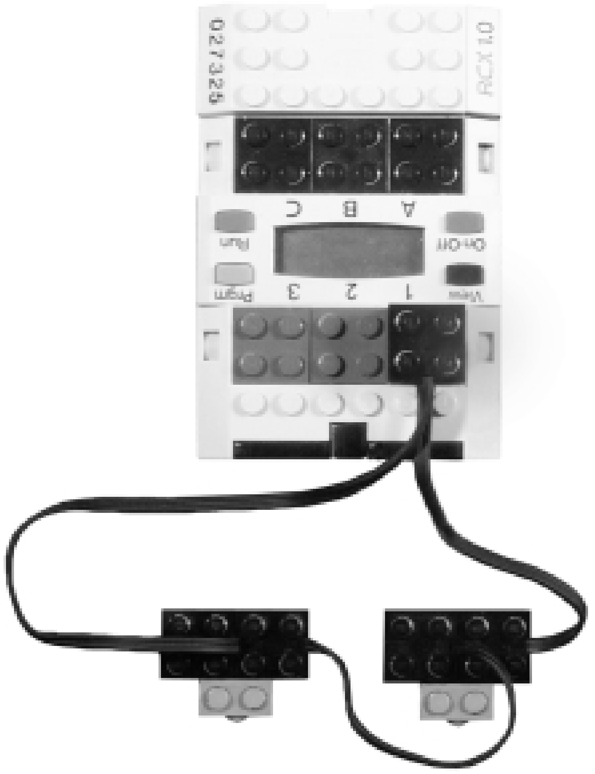
Connecting Multiple Sensors to the Same Port

In some particular cases, connecting multiple sensors to the same port is doable and safe for your devices. Touch sensors, for example, are easy to combine together in an OR configuration, meaning that if any in a group of them gets pressed, you read an *on* state. This is very easy to achieve, simply stacking all the connectors that come from the sensors on the same port. You cannot tell which one was pressed, but there are indeed situations where you can deduce this information from other known facts. For example, say you have a robot with front and rear bumpers. You can connect them to two switches wired to the same port.

When a bumper closes, your program knows if the robot was going forward or backward, thus it can properly interpret the information and behave accordingly. In another example, perhaps your moving robot has a lift arm that requires a limit switch to stop at a specific position. If your robot is stationary when it activates the arm, you can safely use the same port for that limit switch and for a switch wired to a bumper.

The scheme to combine two sensors in an AND configuration is a bit more complex. Tom Schumm came up with the solution shown in Figure 4.15. It works well, provided you connect the wires exactly as shown in the diagram. You get an *on* state only when both sensors are pressed at the same time. The scheme can be extended for an AND configuration using more than two sensors, though it's hard to imagine a situation where you might need such a combination.

Figure 4.15 Connecting Two Touch Sensors in an AND Configuration

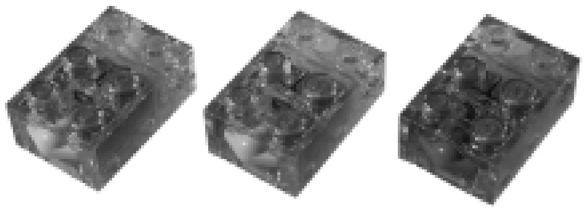


The other sensors (light, rotation, and temperature), don't go together well when used on the same port. If you combine any of them with others of the same or different kind, you will get unpredictable or useless results. There's only one very significant exception to this rule: You can connect a light sensor and a touch sensor to the same port (when configured for a light sensor). This trick, suggested by Brian Stormont, works because the light sensor actually never reads more than about 90 percent, but when the touch sensor gets pressed, the reading jumps to 100 percent, allowing your code to detect the event. The only drawback is that you cannot read the light intensity while the touch sensor is closed. But there are many situations where the touch receives only short impulses, and by applying this trick, you conserve one of your input ports.

Other Sensors

There are other kinds of sensors in the LEGO universe, but we won't discuss them in great detail because they are either difficult to acquire or not very useful. The Cybermaster set (code 8482) includes three touch sensors that are very similar to the MINDSTORMS variety, but that come in three flavors recognizable by the different colors of the buttons (see Figure 4.16). Their transparent casing allows you to see the internal mechanism, which feature internal resistors of different values. For this reason, in terms of raw values, they return different individual readings. This means you can wire them to the same port, and by reading the resulting raw value, determine which one was pressed.

Figure 4.16 The Cybermaster Touch Sensors



The LEGO DACTA line of products includes other sensors designed to survey weather conditions (like humidity) or other specific quantities. They are of no general use, and tend to be very expensive.

Many people have developed their own designs when building custom sensors, and some of them are quite useful if you're open to adding nonoriginal parts to your system. We'll return to this topic in Chapter 9.

If you want to learn more about how LEGO (and non-LEGO) sensors work, don't miss the reference material in Appendix A, and be sure to check out Michael Gasperi's site as well. He is an authority in this field, having discovered many functional details himself, and so displays them in his Web site along with useful information collected from other people.

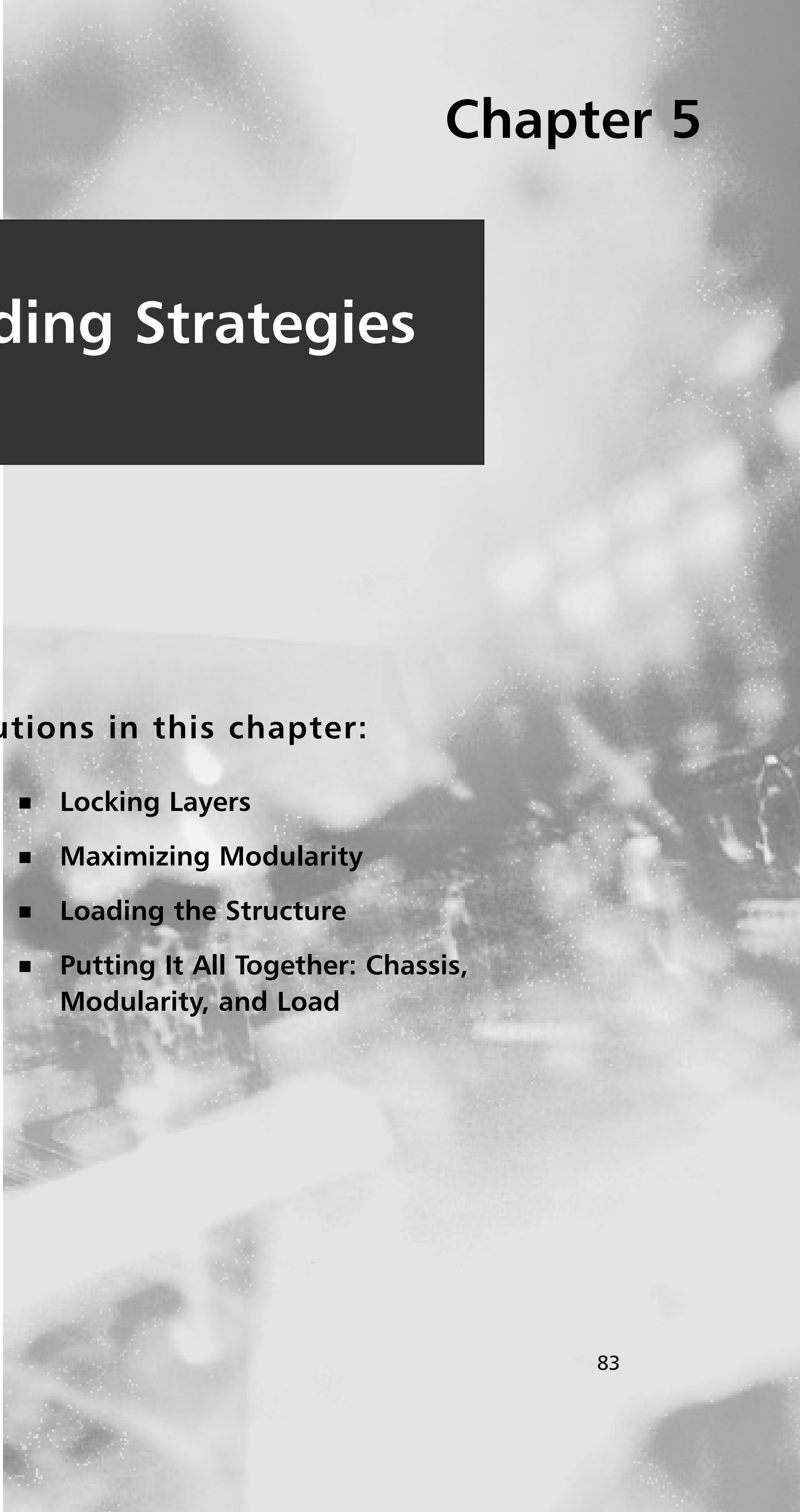
Summary

In this chapter, we've introduced you to the world of sensors, four basic types in particular: touch, light, rotation, and temperature. Their basic behavior is easy to understand, but here you've discovered that if you want to get the very most out of them, you must study them in greater detail. The touch sensor, for example, seems to be a simple device, but with some clever work on your part, it can become an important tool for counting clicks, or can make a good bumper.

You were also introduced to the light sensor, a small piece of incredibly versatile hardware, which can act as a substitute for both the touch and rotation sensors with minimal effort. Together with the IR LED of the RCX, it makes proximity detection possible, a technique that allows your robot to avoid obstacles before it physically touches them.

The rotation sensor will be your partner in the most sophisticated project of this book. Now you know how it works, and also how to replace it in case you don't have one.

Only the temperature sensor received very little attention. It's the Cinderella of this chapter, basically because it has very limited applications. Nevertheless, it will have its moment of glory at the end of Part II.



Chapter 5

Building Strategies

Solutions in this chapter:

- **Locking Layers**
- **Maximizing Modularity**
- **Loading the Structure**
- **Putting It All Together: Chassis, Modularity, and Load**

Introduction

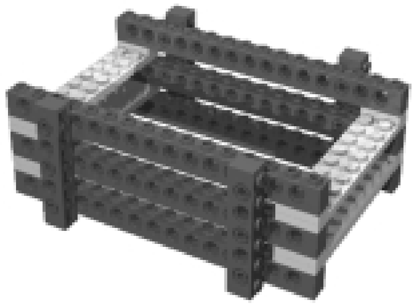
Having discussed motors and sensors, and geometry and gearings, it's now time to put all these elements together and start building something more complex. We stress the fact that robotics should involve your own creativity, so we won't give you any general rule or style guide, simply because there aren't any. What you'll find in this short chapter are some tips meant to make your life easier if you want to design robust and modular robots.

Locking Layers

Recall the standard grid we discussed in Chapter 1. We showed how it leads to easy interlocking between horizontal and vertical beams. The sequence was: 1 beam, 2 plates, 1 beam, 2 plates…

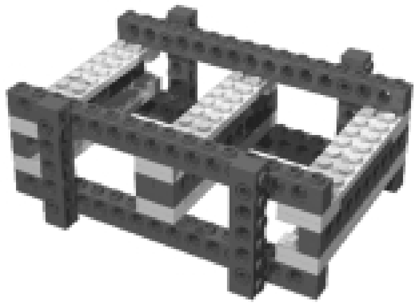
You can take advantage of the plate layer between the beams to connect two groups of stacked beams, thus getting a very simple chassis like the one in Figure 5.1. If you actually build it, you can see how, despite its simplicity, it results in a very solid assembly. This also proves what we asserted in Chapter 1 regarding the importance of locking layers of horizontal beams with vertical beams. For instance, if you remove the four 1 x 6 vertical beams, the structure becomes very easy to take apart.

Figure 5.1 A Simple Chassis



You're not compelled to place all the beams in one direction and the plates in another. Actually, you are likely to need beams in both directions, and Figure 5.2 shows a very robust way to mount them, locked in the intermediate layer of our example structure.

Figure 5.2 Alternating Plates and Beams



NOTE

Remember to use the *black pegs* (or *pins*) when connecting beams. They fit in the holes with much more friction than the gray ones, because they are meant to block beams. The *gray pegs*, on the other hand, were designed for building movable connections, like levers and arms.

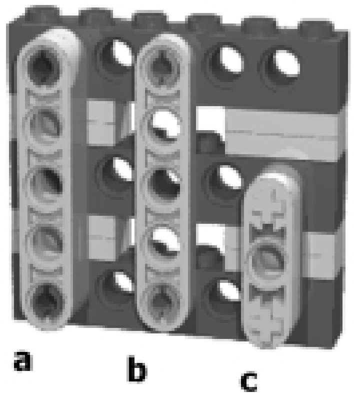
Sometimes you want to block your layers with something that stays *inside* the height of the horizontal beams, maybe because you have other plates or beams above or below them. The full beams we've used up to this point extend slightly above and below the structure. The *liftarms* help you in such cases, as shown in the three examples of Figure 5.3:

Liftarm a	Two coupled 1 x 5 liftarms with standard black pegs
Liftarm b	A single 1 x 5 liftarm and .75 dark gray pegs
Liftarm c	Two 1 x 3 liftarms with axle-pegs

NOTE

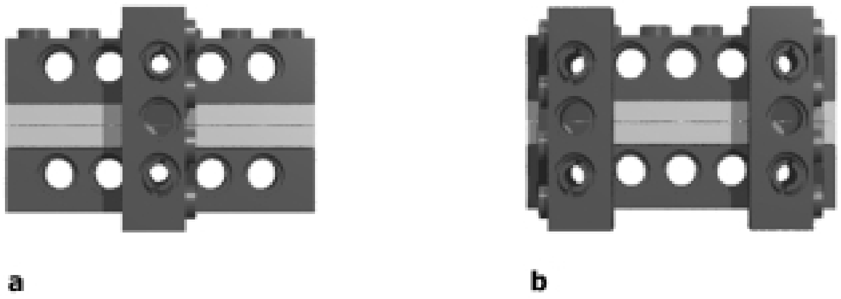
Naming all the individual LEGO parts is not an easy task. Some people call a *half-beam* what we refer to as a *liftarm*, because it has half the thickness of a beam. Due to this, we chose to use the terminology defined in a widely accepted source: the LUGNET LEGO Parts Reference (see Appendix A for the URL to the site).

Figure 5.3 Using Liftarms to Lock Beams



Despite our insistence on the importance of locking beams, there's no need to go beyond the minimum required to keep your assembly together. When the horizontal beams are short, a single vertical beam is usually enough. The example a in Figure 5.4 is better than its b counterpart, because it reaches the same result with fewer parts and less weight. Weight is, actually, a very important factor to keep under control, especially when dealing with mobile robots. The greater the weight, the lower the performance, due to the inertia caused by the mass and because of the resulting friction the main wheel axles must endure.

Figure 5.4 One Vertical Beam Is Sometimes Enough



Bricks & Chips…

What Is Inertia?

In physics, *inertia* is the tendency objects have to resist changes when in states of motion or rest. Objects at rest tend to stay at rest, while objects in motion tend to stay in motion, moving with the same direction and speed. All objects have this tendency, but some more so than others: chiefly because inertia depends on *mass* (quantity of matter). A good example of how mass affects inertia comes from something with which most people have a direct experience: shopping carts! When the cart is empty, you can easily start and stop it, or change its direction, with minimum effort. The more stuff you put inside, however, the more strength is required to maneuver it. Why? Because its objective mass, and thus its inertia, has increased. Similarly, the greater the mass of your robot, the more force is required from its motors when accelerating or braking.

Maximizing Modularity

While building your robot, you will likely have to dismantle and rebuild it, or parts of it at least, many times. This isn't like following someone's detailed instructions; it's more of a trial and error process. Unless you're a very experienced builder and are blessed with clear ideas, your design will develop in both your mind and your hands at the same time.

For this reason, it's best to make your model as easy to take apart as possible, or, to term it more appropriately, your robot should be *modular* in construct. Building in a modular fashion also gives you the opportunity to reuse components in other projects, without having to rebuild common subsystems that already work. This is not always possible, because when you want something really compact, you have to trade away some modularity in favor of tighter integration. Nevertheless, it's a good general building practice, especially when constructing very large robots.

The same principle applies to your most important components: motors, sensors, and obviously the RCX itself. If you are becoming obsessed by LEGO robotics, you will probably buy some extra parts to expand your building possibilities. You might have the resources to start more than one project at a time, or to not be forced to dismantle your last robot when building a new one—however

RCXs, motors, and sensors are definitely not cheap, so your best option is to install them in a way that makes them easy to remove without having to break your robot down into single parts.

NOTE

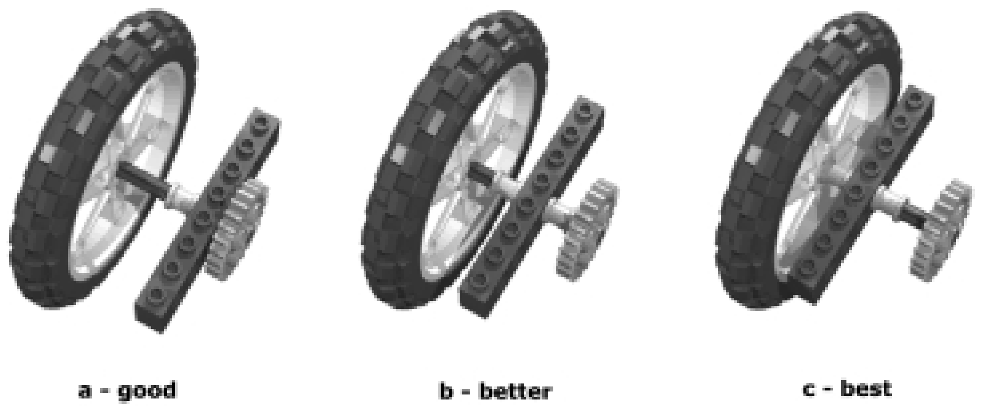
One good reason to make your RCX easily detachable is that you must be able to change batteries when necessary. The most common solution is to keep the RCX at the very top of your robot—this way you can also easily access the push buttons and read the display.

Loading the Structure

Even the most minimal configuration of a mobile robot has to carry a load of about 300g (11 oz): the weight of one RCX (with batteries) and two motors. Adding cables, sensors, and other structural parts, can easily push you up to about 500g (18 oz). Should you worry about this mass? Is its position relevant?

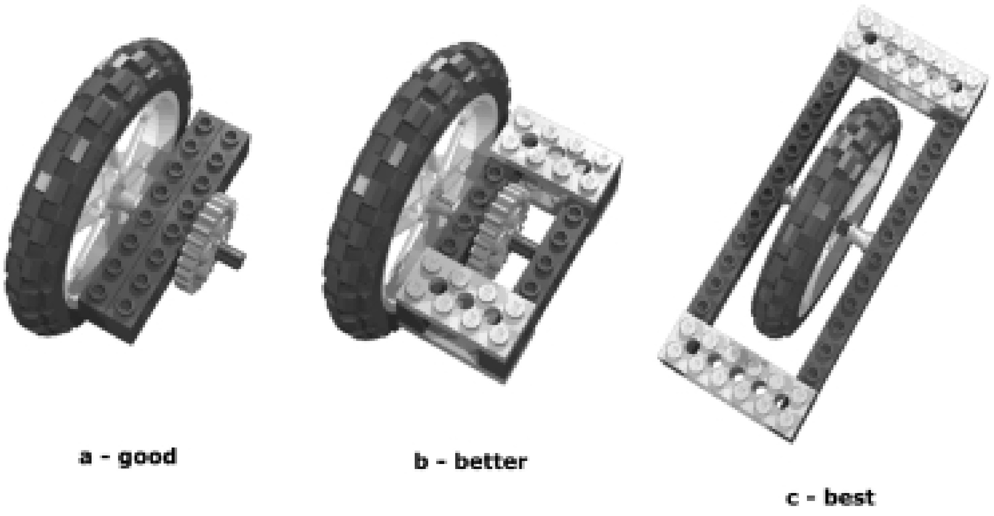
The first factor you need to consider is friction. You should take all possible precautions to minimize it. This is especially true where the structure attaches to the wheels, because it is there that you transfer all the weight to the wheels by way of the axles. The wheel acts as a lever: the greater the distance from its support, the greater the resulting force on the axle. Such forces tend to bend axles, twist beams, and produce plenty of friction between the axle and the beam itself. For this reason, it's important you keep your wheel as close as possible to its supporting beam. Figure 5.5 shows three examples: a being the worst case, with c the best.

Figure 5.5 Keep a Wheel as Close as Possible to Its Supporting Beam



We suggest you also support the load-bearing axles with more than a single beam whenever possible. The three examples shown in Figure 5.6 are better than those in Figure 5.5, with 5.6c being the best among all the solutions shown so far. The use of two supports, one on either side of the wheel, like on a bicycle, avoids any lever-effect created by the axle on the support, thus reducing the friction to a minimum.

Figure 5.6 Two Supporting Beams Are Better than One



The position of the RCX has a strong influence on the behavior of mobile robots. It's actually the shape and weight of the whole robot that determines how it reacts to motion, but the RCX (with batteries) is by far the heaviest element and thus the most relevant to balancing load. To explain *why* balancing load is important, we must recall the concept of *inertia*. We explained earlier in the chapter that any mass tends to resist a change in motion. In some cases, to resist *acceleration*. The greater the mass, the greater the force needed to achieve a given variation in speed.

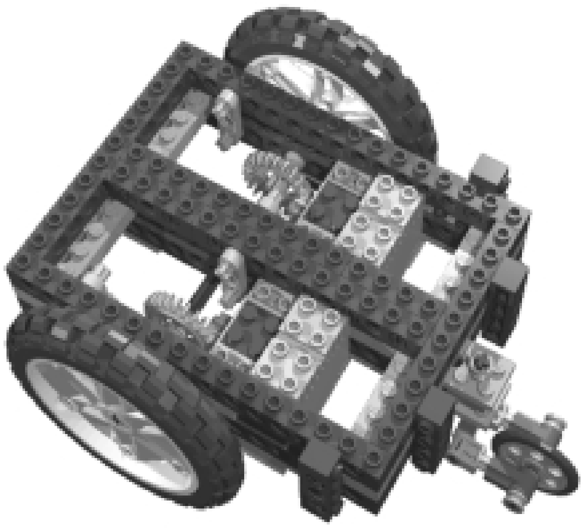
The Acrobot model shown in the MINDSTORMS Constructopedia works under this same principle. If you have already built and tried it, did you wonder why it turns upside down instead of moving forward? This happens because the inertia of the robot keeps it in its present condition—which is stationary. Once power is supplied to the motor, the wheels try to convert that power into motion, accelerating the robot. But the inertia is so great that the force resorts to the path with least resistance, turning the body of the robot instead of the wheels. After having turned upside down, the robot has the undriven wheels in front of it, preventing it from turning again, and now can't do anything other than accelerate.

You probably don't want your robots to behave like Acrobot. More likely, you're looking for stable robots that don't lose contact with the ground. You can use gravity to counteract this unwanted effect, putting most of the weight further from the driving axles. There's no need for complex calculations, simply experiment with your robot, running a simple program that starts, stops, reverses, and turns the robot to see what happens. Place the RCX in various positions until you're satisfied with the result.

Putting It All Together: Chassis, Modularity, and Load

The following example summarizes all the concepts discussed so far in this chapter. Using only parts from the MINDSTORMS kit, we built the chassis shown in Figure 5.7. Its apparent simplicity actually conceals some trickiness. Let's explore this together.

Figure 5.7 A Complete Platform

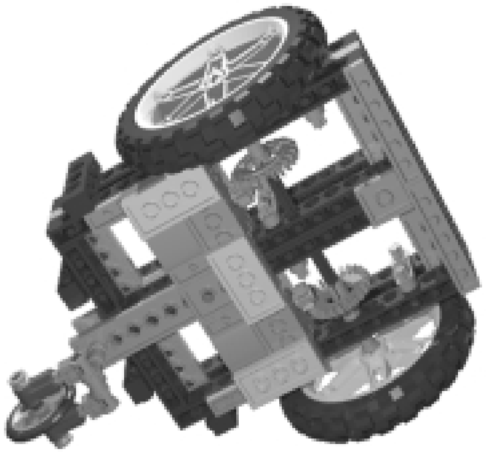


It's built like a sandwich, with two layers of beams that contain a level of plates. It's robust, because vertical beams lock the layers together. Notice that for the inner part of the robot, we used 1 x 3 liftarms instead of 1 x 4 beams. This way the top results in a smooth surface where one can easily place the RCX or other components.

The load-bearing axles are two #8 axles that support both the outer and inner beams (#8 means that the axle is 8 studs long), while the wheels are as close as possible to their supports.

The motors have been mounted with the 1 x 2 plates with rail, as explained in Chapter 3 (look back to Figure 3.4). They are kept in place by two 2 x 4 plates on their bottom (Figure 5.8), but by removing those plates you can quickly and easily take out the motors without altering the structure (Figure 5.9).

Figure 5.8 Bottom View



You can also remove the pivoting wheel and the two main wheels in a matter of seconds to reuse them for another project (Figure 5.10). We should mention here that the pivoting wheel is quite special, since it's what makes a two-wheeled robot stable and capable of smooth turns. The technique of making a good pivoting wheel has its own design challenges, of course, which we'll explore in Chapter 8.

The truth is that if you own only the Robotic Invention System, you probably won't have enough parts to build another robot unless you dismantle the whole structure. If you have more LEGO TECHNIC parts, however, you can leave your platform intact and reuse wheels and motors in a new project.

Figure 5.9 Removing the Motors

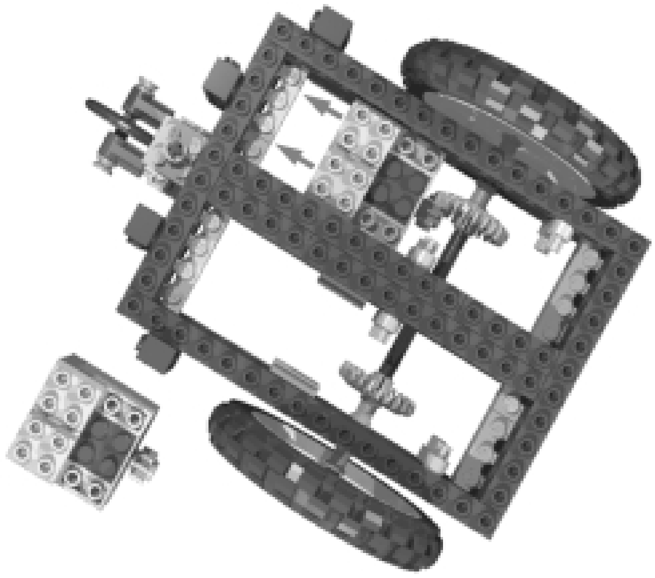
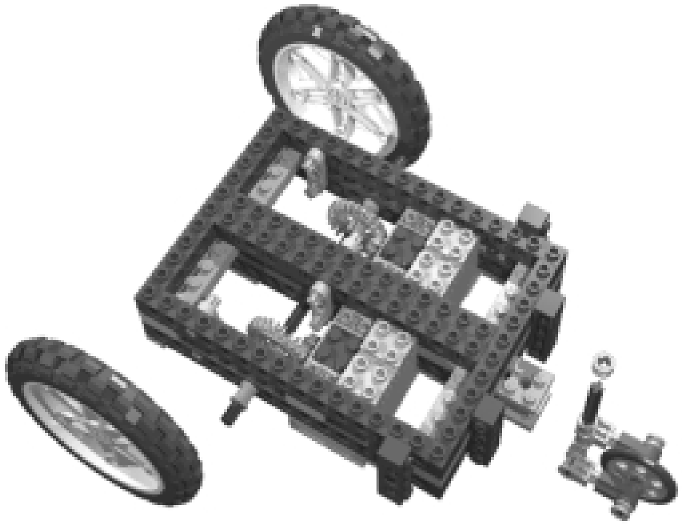


Figure 5.10 …and the Wheels

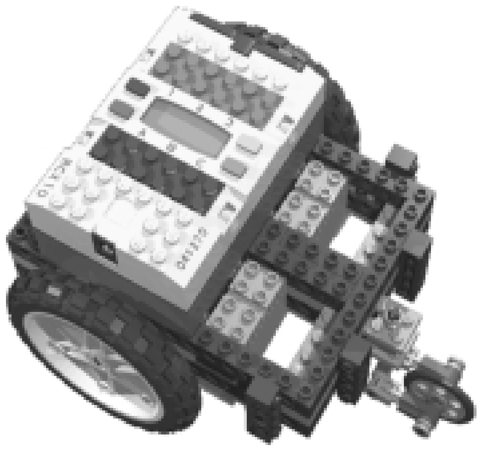


Now we can experiment with load and inertia. If you have the LEGO remote control, you don't need to write any code. If not, we suggest you write a very short program that moves and turns the robot. You don't need anything more complex than the following pseudo-code example, which will drive your robot briefly forward then backward, and make it turn in place:

```
start left & right motors forward
wait 2 seconds
stop left & right motors
wait 2 seconds
start left & right motors reverse
wait 2 seconds
stop left & right motors
wait 2 seconds
start left motors forward
start right motors reverse
wait 2 seconds
stop left & right motors
```

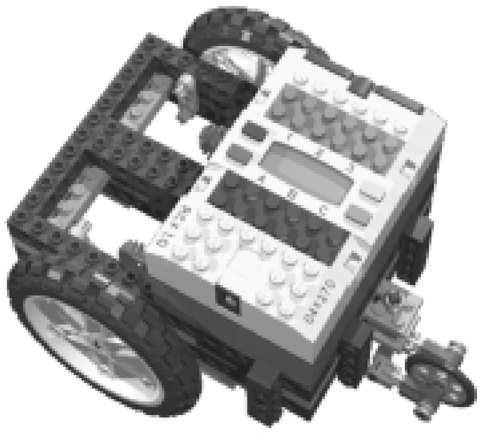
Place your RCX in different locations and test what happens. When it is just over the main wheel axles (Figure 5.11), the robots tend to behave like the Acrobot and overturn easily.

Figure 5.11 Poor Positioning of the Load RCX Makes This Robot Unstable



As you move the RCX toward the pivoting wheel, the robot becomes more stable (Figure 5.12). It still jumps a bit on sudden starts and stops, but it doesn't flip over anymore.

Figure 5.12 Better Positioning Improves Stability



Summary

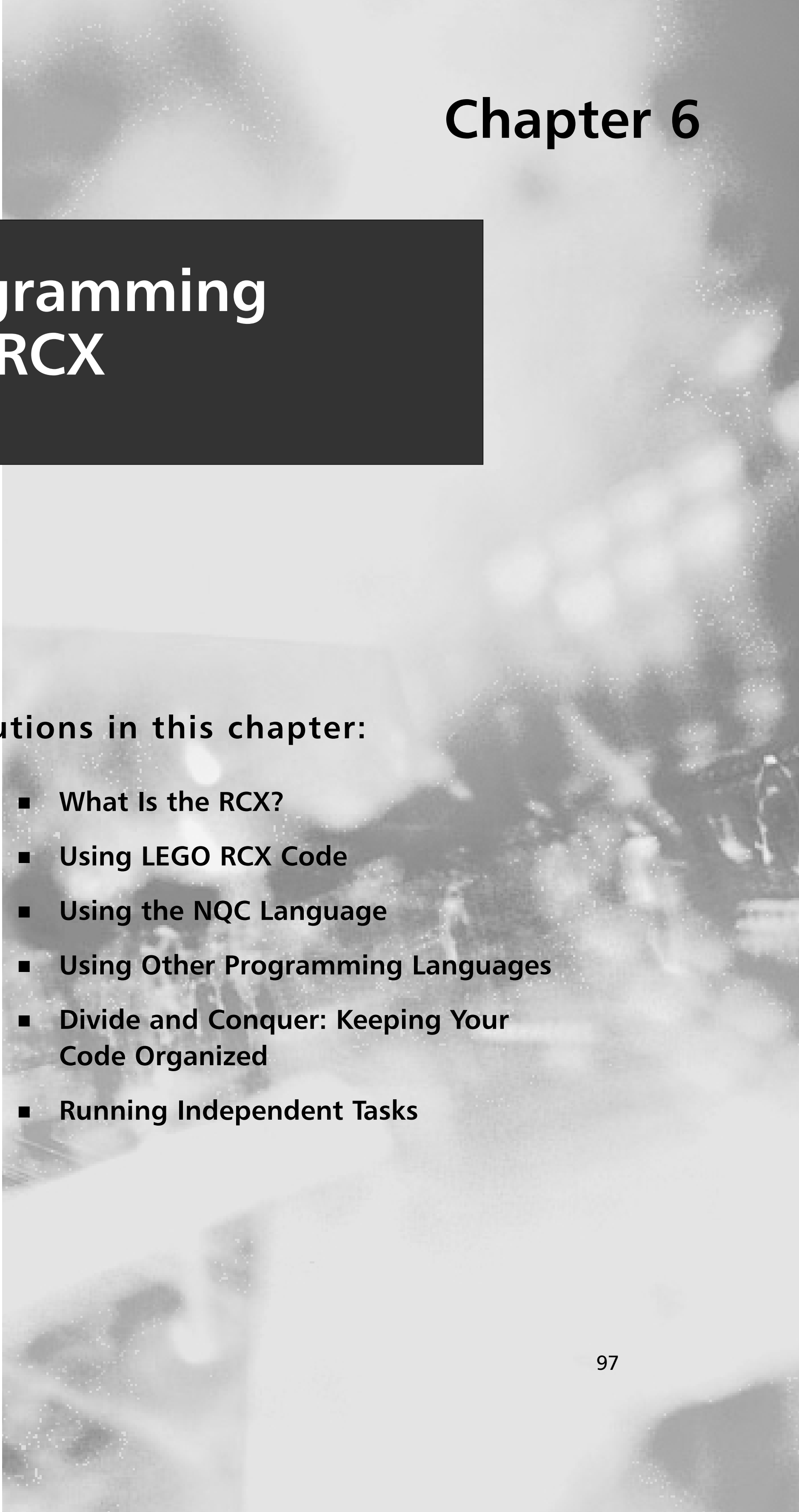
The content of this chapter may be summarized in three words: layering, modularity, and balancing. These are the ingredients for optimal structural results.

Thinking of your robot in terms of *layers* will help you in building solid, well-organized structures. Recall the lessons you learned in Chapter 1 about layering beams and plates and bracing them with vertical beams to get a solid but lightweight structure. A robust chassis comes more from a good design than from using a large number of parts.

Modularity can save you time, allowing you to reuse components for other projects. This is especially important when it comes to the "noble" parts of your MINDSTORMS system—the sensors, motors and, obviously, the RCX—because they are more difficult and expensive to replicate. You should put this concept into operation not only for single parts, but for whole subsystems (for example, a pivoting wheel), which you can transfer from one robot to another.

Balancing is the key to stable vehicles. Keep the overall mass of your mobile robots as low as possible to reduce inertia and its poor effects on stability. Experiment with different placements of the load, mainly in regards to the RCX, to optimize your robot's response to both acceleration and deceleration. We will look more deeply into this matter in Chapter 15, when we learn how to build walking robots (where management of balance is a strict necessity).

Unfortunately, these goals are not always reachable; sometimes other factors force you to compromise. Compactness, for example, doesn't mesh well with modularity. Certain imposed shapes, like those used in the movie-inspired droids of Chapter 18, can force you to bypass some of the rules stated here. We aren't saying they can't be violated. Use them as a guide, but feel free to abandon the main road whenever your imagination tells you to do so.



Chapter 6

Programming the RCX

Solutions in this chapter:

- What Is the RCX?
- Using LEGO RCX Code
- Using the NQC Language
- Using Other Programming Languages
- Divide and Conquer: Keeping Your Code Organized
- Running Independent Tasks

Introduction

As we explained in the Introduction, this book is not about programming—there are already many good resources about programming languages and techniques, and about programming the RCX in particular. However, the nature of robotics (often called *mechatronics*) is such that it combines the disciplines of mechanics, electronics, and software, meaning you cannot discuss a robot's mechanics without getting into the software that controls the electronics that drives the machine. Similarly, you cannot write the program without having a general blueprint of the robot itself in your mind. This applies to the robots of this book as well. Even though we are going to talk mainly about building techniques, some projects have such a strong relationship between hardware and software that explaining the first while ignoring the latter will result in a relatively poor description. For these reasons, we cannot simply skip the topic, we need to lay the foundations that allow you to understand the few code examples contained in the book.

In the previous chapters, we mentioned the RCX many times, having assumed that you are familiar with the documentation included in the MINDSTORMS kit and know what the RCX is. The time has come to have a closer look at its features and discover how to get the most from it. We will describe its architecture and then give you a taste of the broad range of languages and programming environments available, from which you can choose your favorite. Our focus will be on two of them in particular: RCX Code, the graphic programming system supplied with the kit, and NQC, the most widespread independent language for the RCX.

The last sections of the chapter provide a complete code example, which is meant to help explain how to write well-organized code that is easy to understand and maintain, and is designed to familiarize you with the programming structures you'll find later in the book.

What Is the RCX?

The RCX is a computer. You are used to seeing computers that have a keyboard, a mouse, and a monitor—devices created to allow human users to interface with their computers—but the RCX hasn't got any of those features. Its only gates to the external world are a small display, three input ports, three output ports, four push-buttons, and an infrared (IR) serial communication interface. The RCX is actually more similar to industrial computers created to control machinery than it is to your normal desktop computer. So, how can you program it if it hasn't any user interface? You write a program on your PC, then transfer it to the RCX with the help of the IR tower (a device designed to work as a link between the PC and the RCX), and, finally, the RCX executes it.

To understand how the RCX works, imagine a structure made of multiple layers. At the very bottom is the processor, an Hitachi H8300, which executes the machine code instructions. The processor cooperates with additional components that convert signals from the ports into digital data, using chips that provide memory for data and program storage. Just as with most computers, the memory of the RCX is made up of two types: read-only memory (ROM) and random access memory (RAM). The content of the ROM cannot be altered or cancelled in any way, since it is permanently written on the chips, while the data in the RAM can be replaced or modified. The RAM requires a continuous power supply in order to retain its content. When the supply breaks, everything gets erased.

Above the processor and circuit layer you find the ROM code. When you unpack your brand new RCX, there's already some code stored in its internal ROM that's aimed at providing some basic functionality to the RCX: input ports signal conversion, display and output ports control, and IR communication. If you are familiar with the architecture of a personal computer, you can compare this ROM code to the basic input/output system (BIOS), the low-level machine code which is in charge of booting the computer at startup and interfacing with the peripherals.

An RCX with just the ROM code is as useless as a personal computer with just the BIOS. On top of the ROM code layer the RCX runs the *firmware*, which, to continue with our comparison to computers, is its *operating system*. The term *firmware* denotes a kind of software the user normally doesn't alter or change in any way; it's part of the system and provides standard functionality, as operating systems do. In RCX, the firmware is not burned into the system like the ROM code, rather it is stored in the internal RAM, and you download it from your PC using the infrared interface. The LEGO firmware was copied to your PC during the installation of the MINDSTORMS CD-ROM, and transferred to your RCX by the setup process.

The firmware is not the final layer of the system: on top of it there's your own code and data. They will be stored in the same RAM where the firmware is, but from a logical standpoint they are considered to be placed at a higher level. As we explained earlier, you write your code on the PC, then send it to the RCX through the infrared interface. The MINDSTORMS software on the PC side, called *RCX Code*, translates your program (made of graphical code blocks) into a compact form called *bytecode*. The RCX receives this bytecode via the IR interface and stores it in its RAM. When you press the **Run** button, the firmware starts *interpreting* the bytecode and converting its instructions into actions.

WARNING

Because the firmware is stored in RAM, it will vanish if your RCX remains without power for more than a few seconds, and you will have to reload it before using your RCX again. When you power off your RCX, the RAM remains supplied just to keep the firmware in existence, and this is the reason why the RCX will slowly drain the batteries even when switched off. If you plan not to use it for more than a few days, we suggest you remove the batteries to preserve them. Remember that when you need your RCX again, you will have to reload the firmware.

Let's summarize the process from the top to the bottom level:

- You write your program using RCX Code, the MINDSTORMS software on the PC side.
- RCX Code automatically translates your program into a compact format called *bytecode*.
- Using the IR link between the PC—via the IR tower—to the RCX, you transfer the bytecode version of your program to the RAM of the RCX.
- The firmware interprets your bytecode and converts it into machine code instructions, calling the ROM code routines to perform standard system operations.
- The RCX processor executes the machine code.

Most of these steps are hidden to the user, who simply prepares the program on the PC, downloads it to the RCX, presses the **Run** button, and watches the program execute.

A Small Family of Programmable Bricks

The RCX belongs to a small LEGO family of *programmable bricks*. The first to appear on the scene was the Cybermaster, a unit that incorporates two motors, three input ports, and one output port. It shares with the MINDSTORMS devices the ability to be programmed from a PC, with which it communicates through the "tower," which in this case is based on radio frequency instead of infrared transmission. But the similarities end here, and the Cybermaster has more limitations than the RCX:

- Its three input ports work with passive sensors only.
- The firmware is in ROM instead of RAM. This means that it's not possible to upgrade it to a newer version.
- The RAM is much smaller than the one in the RCX and can host only very short programs.

The *Scout*, contained in the Robotics Discovery Set, is programmable from the PC with the same IR tower of the RCX (not included in the set), but features a larger display that allows some limited programming, or better said, it allows you to choose from among various predefined behaviors. It features two output ports, two input ports (passive sensors only), and one embedded light sensor. Like for the Cybermaster, the firmware is in ROM and cannot be upgraded or modified.

Using LEGO RCX Code

RCX Code is the graphical programming tool that LEGO supplies to program the RCX. If you have installed the MINDSTORMS CD-ROM, followed the lessons, and tried some projects, you're probably already familiar with it.

RCX Code has been targeted to kids and adults with no programming experience, and for this reason it is very easy to use. You write a program simply by dragging and connecting *code blocks* into a sequence of instructions, more or less like using actual LEGO bricks.

There are different kinds of code blocks that correspond to different functions: You can control motors, watch sensors, introduce delays, play sounds, and direct the flow of your code according to the state of sensors, timers, and counters. RCX Code also provides a simple way to organize your code into *subroutines*, groups of instructions that you can call from your main program as if they were a single code block.

When you think your code is ready to be tested, you download it to the RCX through the IR tower. The RCX has five *program slots* that can host five independent programs. When downloading the code, you choose which slot to download to, and with the **Prgm** push-button on your RCX, you select which program to execute.

The intuitiveness of RCX Code makes it the ideal companion for inexperienced users, but it has some major drawbacks:

- Its set of instructions is very limited, and doesn't disclose all the power your RCX is capable of. Sooner or later you will start desiring a more powerful language.
- Its graphical interface is not suitable for large programs. The sequence of code blocks, though very intuitive for small programs, becomes hard to follow when you have tenths or hundredths of them.

For these reasons, you'll find that RCX Code is a barrier to the development of complex projects.

Using the NQC Language

The LEGO firmware is a solid, well-tested software that provides a rather complete functionality. The surprising thing is that it actually offers many more possibilities than what the RCX Code discloses to us. It's like having a car whose motor is capable of 100 mph but with an accelerator pedal that allows you to reach no more than 50 mph. The power is there, but the interface doesn't allow you to get at it. This fact drove some independent developers to create new programming environments able to get the most from the LEGO firmware, providing access to those features that RCX Code conceals. All of them share the same approach, which consists of making a new interface on the PC side that's able to generate bytecode and transfer it to the RCX.

Developed and maintained by Dave Baum, the language called Not Quite C (NQC) has achieved enormous popularity among MINDSTORMS fans and is by far the most widespread of this category. NQC is based on C-like syntax; if you're not a programmer, or if you have no experience with C, don't be frightened by this. NQC has a very smooth learning curve, and comes with a lot of documentation and tutorials. The success of NQC has come about for many reasons:

- It's based on the original LEGO firmware, thus taking advantage of its ability to produce very reliable code, and at the same time freeing all of RCX Code's hidden power. Even from its very first releases NQC has proven to be rock solid.
- Dave Baum puts a lot of effort into maintaining it, continuously adding new features and acknowledging new opportunities offered by the

LEGO firmware. NQC supported the new RCX 2 firmware version well before it was officially released in any LEGO product.

- It is multiplatform, both on the host side (it runs on PC, Mac, and Linux machines) and on the target side (it supports all the LEGO programmable bricks: RCX, Scout, Cybermaster).
- It is self-contained. To use NQC you don't need any other tools than a simple text editor (Windows Notepad is enough). The installation procedure is as easy as copying a file.
- There are many documents and tutorials, in many different languages, that help new users understand all the details.
- The NQC compiler is a command-line tool, with no user interface, but people have developed nice integrated development environments that encapsulate NQC inside a productive system that includes editors, tools, diagnostics, data logging, and other utilities, as well as, most importantly, the Bricx Command Center.
- NQC is free software released under the Mozilla Public License (MPL).

Some of the projects discussed in this book actually require that you go beyond the limits imposed by RCX Code. This is the main reason why we chose NQC to illustrate the few programming examples. NQC also has the advantage that, being a textual language, it allows for a very compact representation that better suits the format of a book.

Using Other Programming Languages

The fact that LEGO placed the firmware of the RCX in the RAM left the system open to other languages that follow a more radical approach. Instead of substituting the software that produces bytecode on the PC side, they *replaced* the firmware on the RCX. It's important to note that installing any of these alternative environments doesn't entail any risk at all for your RCX. You can always return to your original system.

All the work that has been done in this direction heavily relies on Kekoa Proudfoot's pioneering hacking of the RCX. Kekoa patiently disassembled the LEGO firmware and documented all the routines and their calls, thus laying the foundations for the subsequent alternative firmware versions.

Using legOS

In 1999, Markus Noga started The legOS Project, the first attempt to write a replacement firmware for the RCX. Noga's goal was to bypass all the limitations of the bytecode interpreter to run the code directly on the Hitachi H8300 processor of the RCX. A legOS program is a collection of system management routines that you link to your own C or C++ code and load to the RCX in place of the firmware.

What was initially an individual effort turned into a collective open source project under the Mozilla Public License. The legOS Project is now managed by Luis Villa and Paolo Masetti and maintained by a team of a dozen developers.

The installation is not always straightforward, especially for Windows machines. You need to be a programming expert, because what you have to deal with here is true C, not the simplified and friendly NQC version. You have to manage cross-compilers and Unix emulators if you don't run a Unix-like machine, so legOS is definitely not for everyone. But for this price it unleashes the full power of your RCX up to its last bit. You get full control of any resource and any device, can use any C construct and structure, and can address any single byte of memory. Plus, your code runs at an astonishing speed when compared to the interpreted bytecode.

Using pbForth

The pbForth language (the name stands for *programmable brick FORTH*) is the result of Ralph Hempel's experience in designing and programming embedded systems, a field where FORTH is particularly well suited. Conceived in the sixties, the FORTH language has a strong tradition in robotics, automation, and scientific applications. More than a language, FORTH is an interactive environment. The traditional concepts of editing source files, compiling, linking, and so on, don't translate very well to FORTH; it's mainly a stand-alone system.

Ralph Hempel's implementations make no exception to this rule. You download the pbForth kernel to your RCX, and from that moment on you *dialog* with it using a simple terminal emulator. For this reason, pbForth is very portable and very easy to install on any platform.

If you haven't any experience with FORTH, it will probably seem a bit strange to you in the beginning. The language is based on the *postfix notation*, also called reverse polish notation (RPN), which requires you to write the operator after the operands.

If you decide to give pbForth a try, you will discover the benefits of an extensible system that naturally leads you to program in terms of layers. You might find it challenging to learn, but it's a productive—and fun—tool with which you can write compact and efficient code.

Using leJOS

Jose Solorzano started the TinyVM Project, a small Java footprint for the RCX. TinyVM was designed to be as compact as possible, and for this reason lacked much of the extended functionality typical of Java systems. Over the foundation of TinyVM, Jose and other developers designed leJOS, a fully functional Java implementation that includes floating point support, mathematical functions, multiprogram downloading, and much more. LeJOS is an Open Source project and, like legOS, is under continuous development.

leJOS is the newcomer on the scene of MINDSTORMS programming, but we foresee a great future for it. It's complete, portable (currently to PC and Unix-like machines), very easy to install, fast, efficient, and based upon a widespread language. There are also some visual interfaces under development that will make this system even more attractive to potential users.

Using Other Programming Tools and Environments

We know we didn't cover all the available programming tools for the RCX. There are others, like Gordon's Brick Programmer, or Brick Command, that follow the same solution of NQC and convert a textual program into bytecode. There are also a few more replacements for the firmware, like QC or TinyVM. And, finally, some other tools, like ADA for the RCX, that translate source code into NQC code. They are good tools, solid and well-tested, but we choose to describe the most representative and widespread in each class. We recommend you look at Appendix A for further information about the software we introduced here and about other possible choices; the list is so long we are sure you'll find the tool that fits your needs. In the same appendix, you will find some links to other tools that, though not intended for programming, can help you monitor your RCX, transfer data to the PC, graph the status of the input ports, and more.

Divide and Conquer: Keeping Your Code Organized

Up to this point the few programming examples you met were written in a sort of pseudo-code very close to plain natural language. The use of pseudo-code allows the programmer to "play computer" and understand what the program does, but to complete the projects of the book, some of which are a bit complex, you need a real environment to run and test the code with. We chose to write all the examples using NQC because it combines power with compactness, it's easy to install and learn, and has become a widespread standard among thousands of MINDSTORMS programmers. In the following example, we will describe some of the most important features of NQC, but we strongly recommend you read the documentation available from its official Web site, listed in Appendix A. Even if you don't choose NQC, we're sure you can easily translate our examples into your favorite programming language.

What we said in Chapter 5 about keeping your construction designs modular applies to programming as well. Organizing the code into logical sub units is a good programming practice that will often help you in the debugging process. Unless your robot is designed for a very simple task, try to split its code into blocks that correspond to the different situations it's expected to manage and to the actions it should perform. The Latin motto "divide et impera" applies well to programs: the more you divide the code into small sections, the better you can control and understand the program's behavior.

We will use an example to clarify this concept and introduce other tips: Say your robot has been designed to follow a black line, detect small obstacles with a bumper and remove them from its path by pushing the obstacles away with some kind of arm. As we explained earlier, it's impossible to write a program without having a precise idea of how the robot is designed and what it is expected to do. For the example we are going to illustrate, we made the following assumptions about the robot and the environment:

- The line is darker than the floor.
- The robot will follow the left border of the line (e.g., It turns right to go toward the line, left to go away from line).
- Output ports A and C control the left and right drive wheels respectively.
- Output port B operates the arm.

- Input port 1 is attached to a touch sensor connected to the bumper. It closes (goes from 0 to 1) when the bumper is pressed.
- Input port 2 is attached to a face-down light sensor that reads the line.

Here is the initial code you should write:

```
int floor,line;

task Main()
{
  Initialize();
  Calibrate();
  Go_Straight();

  while(true)
  {
    Check_Bumper();
    Follow_Line();
  }
}
```

The main level of your program is quite simple, because at this point you're not concerned with what **Go_Straight** or the other subroutines mean in terms of actions, you're only concerned with the logic that connects the different situations. You are deciding the rules that affect the general behavior of the robot and you don't want to enter into the details of *how* it can actually go straight. This result is achieved by encapsulating the instructions that make your robot go straight into a *subroutine*, a small unit which "knows" what the robot requires in order to go straight. This approach has another important advantage: Your code will be more general because it doesn't depend on the architecture of the robot. For example, for one specific robot "go straight" will mean switching motors A and C on in the forward direction, while for another it might mean switching on motor B in the reverse direction. When you want to adapt the program to a different architecture, you simply change the implementation details contained in the low-level subroutines, without having to intervene on the logic flow.

Let's come back to your main task to examine it in deeper detail. The first instruction is actually placed before the beginning of the task: It declares that you are going to use two *variables* named *floor* and *line* and intended to contain integer numbers. A variable is like a box with a name written on it: You can place something inside, a specific number—that is, you can *assign* a value to the variable. Or you can watch what's inside the box, *reading* the variable. At this stage, you are neither assigning nor reading the variables, you are simply declaring that you need two of them. In other words, you are asking NQC to prepare two boxes with the names just mentioned.

When the user presses the **Run** button on the RCX, the main task begins. After it has completed initialization and calibration procedures, the program starts the robot in straight motion, then it enters an endless loop where the program continuously manages its two tasks: removing obstacles and following the line. The **while(true)** statement repeats all the instructions delimited by the open and close brace forever. In your case, it will execute the Check_Bumper subroutine, then the Follow_line, then the Check_Bumper again in a continuous loop that only the user can interrupt using the **Run** button.

Everything is clear and simple, as it should be. Now let's have a look at what happens at a lower level in our subroutines.

Any program will typically include an *initialization* section, where you set the motor power, configure the sensors, reset timers and counters and initialize variables. This is not required when you use RCX Code, because it automatically configures the input ports for you. NQC, like the other textual environments, requires that you explicitly declare what kind of sensor you connect to each port:

```
void Initialize()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  SetSensor(SENSOR_2,SENSOR_LIGHT);
}
```

The word **void** is what tells NQC that you are describing a subroutine, and it's followed by the name you choose for it. The **SetSensor** statements are used to configure input port 1 for a touch sensor and input port 2 for a light sensor.

The *calibration* routine is designed to inform your robot of the actual light readings it should expect on its path. We discussed this topic briefly in Chapter 4, explaining that keeping your program independent from particular cases is a good general programming practice. In this example, it means you should not write the light sensor thresholds into the code, but rather give your robot the possibility to read them from the environment, and this is what you have declared the *floor* and *line* variables for.

```
void Calibrate()
{
  WaitBumperPress();
  floor=SENSOR_2;
  WaitBumperPress();
  line=SENSOR_2;
  WaitBumperPress();
}

void Wait_Bumper_Press()
{
  PlaySound(SOUND_DOUBLE_BEEP);
  while (SENSOR_1==0);  // wait for bumper press
  while (SENSOR_1==1);  // wait for bumper release
}
```

This code shows that in some situations you can recycle a sensor and use it for more than a single purpose: during the calibration process, the bumper is used as a trigger to tell the robot that it's time to read a value. It also shows that subroutines can be *nested*. In other words, you can make a subroutine call another subroutine. In this particular case, the **WaitBumperPress** is a small service subroutine that produces a beep and then waits until the bumper switch gets pressed and released.

When you run the program, the calibration procedure begins and informs you with a beep that it waits for the first reading. You place your robot with the light sensor on the floor, far from the line, and push the bumper. The program reads the light sensor and stores that value as a typical "floor" value in the *floor* variable. Then it beeps again while waiting to read the line. You place the robot with the sensor just over the line and push the bumper again, making it detect the "line" light value and store it in the *line* variable. The robot finally beeps again, meaning the calibration process has finished and that the next push on the bumper will put it in motion.

This sort of pre-run phase is quite useful in many other situations, such as when you need to prepare the robot for operations by either reading some environmental variable or resetting mechanisms that might have been left in an unknown state by previous executions.

The **Check_Bumper** procedure is in charge of testing whether the robot has hit an obstacle, and if so, how it should react:

```
void Check_Bumper()
{
  if (SENSOR_1==1)
  {
    Stop();
    Remove_Obstacle();
    Go_Straight();
  }
}
```

It checks the bumper, and, if found closed, stops the robot, calls the **Remove_Obstacle** subroutine to clear the path and then resumes motion. Testing the bumper is as simple as checking if SENSOR_1 has become equal to 1, which means that the touch sensor connected to port 1 has been pressed. You notice that we apply here the same concepts used at the main level: encapsulating details into routines at a lower level.

The **Follow_Line** routine is what keeps your robot close to the line edge—let's say the left edge. If the light sensors read too much of the "floor" value, it turns right toward the line. If, on the contrary, it reads too much of the "line" value, it turns left, away from the line. (See Chapter 4 for a discussion of this method.)

```
void Follow_Line()
{
  #define SENSITIVITY 5
  if (SENSOR_2<=floor+SENSITIVITY)      // reading too "floor"
    Turn_Right();
  else if (SENSOR_2>=line-SENSITIVITY)  // reading too "line"
    Turn_Left();
  else
    Go_Straight();
}
```

The method used in this subroutine deserves some explanation. First of all, the word **#define** tells NQC that the following word denotes a *constant*; for the sake of simplicity, you can consider a constant to be like a variable whose value cannot be changed by the program. In this particular case, your program defines the constant *SENSITIVITY* with the value 5. This value is used together with the *floor* and *line* variables to decide what the robot should do. An example with actual numbers can make the things clearer: suppose the **Calibrate** routine placed the value 55 in the *floor* variable and the value 75 in the *line* variable. The program tests if SENSOR_1 is less than or equal to *floor* + *SENSITIVITY*, which results in 55 + 5 = 60, to decide if the robot has to turn right toward the line. Similarly, it tests if SENSOR_1 is greater than or equal to *floor* – *SENSITIVITY*, which corresponds to 75 – 5 = 70, and if this is the case, it makes the robot turn left, away from the line. While the readings remain greater than 60 and lower than 70, the robot goes straight. You can change the value of *SENSITIVITY* to make your robot more or less reactive to readings: An increase will narrow the range of values that allow the robot to go straight, thus your robot will make more corrections in order to remain close to the edge of the line.

The code you wrote so far is rather general and could work for a broad class of robots. Now the time has come to write the part of the program that depends on the physical architecture of your robot.

The **Go_Straight** routine will be very straightforward in most cases. You know from the initial assumptions that the robot has two side wheels (or tracks) driven by two independent motors. In Chapter 8, we will explore this configuration, called *differential drive*, in greater detail. For the moment, let's stick to the fact that if both the motors go forward, the robot goes forward and straight. If one of the motors stops, the robot turns toward the side of the stationary wheel. This knowledge is enough to write the following routines, which control motion:

```
void Go_Straight()
{
  OnFwd(OUT_A+OUT_C);
}

void Stop()
{
  Off(OUT_A+OUT_C);
}

void Turn_Left()
```

```
{
  Off(OUT_A);
  OnFwd(OUT_C);
}

void Turn_Right()
{
  Off(OUT_C);
  OnFwd(OUT_A);
}
```

Designing & Planning…

Benefits of Designing Modular Code

If you follow the principles illustrated in this chapter when writing a modular and well-structured code, your program will result in greater readability, reusability, and testability:

- **Readability** The program is organized into small sections that are easy to comprehend with just a quick glance. This means that your program will be easier to maintain, and more easily understood by the friends with whom you share it.
- **Reusability** Separating the logic of the program from the instruction related to the physical structure of the robot, you make your code more flexible and reusable for different architectures. The general principle is: the upper levels of the code reflect *what* the robot does, while the lower ones reflect *how* the robot does it.
- **Testability** A nice side effect of well-structured code is that it speeds up your testing procedures, segmenting possible problems into small portions of code. Remove (or comment out) the call to **Follow_Line** from inside the repeat block in the main task: Your robot should simply go straight until it hits an obstacle, then activate the arm and remove it. Conversely, you can remove the call to **Check_Bumper** to turn your robot into a simple line follower!

There's one last routine left: **Remove_Obstacle**. Let's say your robot features a very simple arm that works with a single motor and only requires a timed activation:

```
void Remove_Obstacle()
{
  OnFwd(OUT_B);
  Wait(200);
  OnRev(OUT_B);
  Wait(200);
  Off(OUT_B);
}
```

The statement **Wait(200)** makes the program wait for 200 hundredths of a second, or two seconds. This parameter depends on the time your mechanism needs to remove the obstacle, and it is once again related to the physical structure of the robot.

Your program is now finished and ready to be tested. We hope this example made you realize the benefits of a modular and well-structured code.

Running Independent Tasks

All the tools you can choose from to program your RCX support some form of *multitasking*, that is, they support two or more independent tasks that run at the same time. This is not particularly evident when you use RCX Code, but it's a well-documented feature in all the alternative environments.

Multitasking can be helpful in many situations and it's often a tempting approach, but you should use it with a lot of care because it will not always make your life easier. Let's go back for a moment to our previous example: would multitasking have been a good choice? Didn't your robot have two different tasks to manage: line following and obstacle detection? Well, it did, but they were mutually exclusive—after all, your robot was not following the line *while* it removed the obstacle. In cases like this, and in many others, your robot is asked to perform different activities *one at a time* more often than it is asked to perform different activities *at the same time*. Using multitasking, you would have made your code more complex, because of the additional instructions needed to synchronize the tasks. When the **Remove_Obstacle** task stops the robot, it should communicate the **Follow_Line** task to suspend line following, and communicate again when it can be resumed.

In designing a multitasking application, you are required to move from a sequential, step-by-step flow to an *event-driven* scheme, which usually requires additional work to keep the processes coordinated. While sequential programming is like following a recipe to cook something, you can compare multitasking to preparing two or more recipes at the same time. This is quite a common practice in any kitchen, but requires some experience to manage the allocation of resources (stoves, oven, mixer, blender...), respond to the events (something's ready to be taken out of the oven) and coordinate the operations so the tasks don't conflict with each other. You have to think in terms of *priorities*: Which dish should you put in the oven first? Programming independent tasks implies the same concerns: You must handle the situations where two tasks want to control the same motor or play two different sounds. The RCX is well-equipped to manage resource allocation and to support event-driven programs, and NQC gives you full access to these features. However, most of the effort is still on your shoulders: no tool makes up for the disadvantages inherent in a bad design.

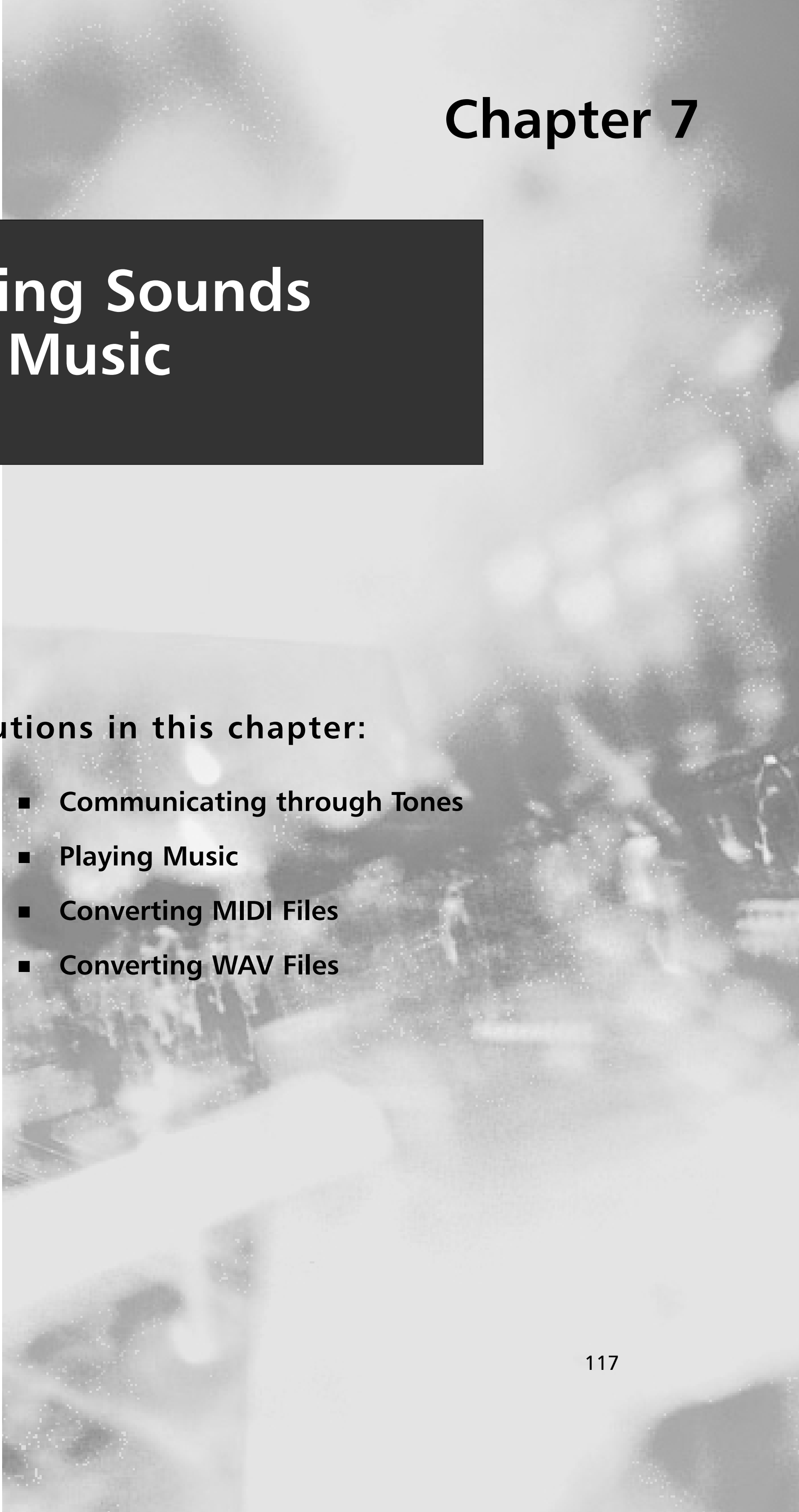
In our experience with LEGO robotics, there are few actual situations where multitasking is absolutely necessary, or even useful. Our suggestion is that you approach it only when your robot performs some really independent activities, like playing background music while navigating a room, or responding to messages while looking for a light source.

Summary

In this chapter, you took some first steps on your path to programming LEGO robots. We started describing the RCX, the LEGO programmable unit that's the core of your robots, to unveil some of its secrets. You discovered how its architecture can be easily understood in terms of layers: your program, its translation into bytecode, the interpreter in the firmware, and the processor which executes the operations.

To create your program on a PC, you can choose from many available tools; we briefly described RCX Code, the original LEGO graphic programming environment, and NQC, the most widely accepted independent language for the RCX. We also reviewed a few other environments—legOS, pbFORTH, leJOS—which follow a more radical approach to the goal of getting the most from the RCX: replacing its firmware.

The second part of the chapter does for programming what the previous chapter did for building: it establishes some guidelines. Oddly enough, the two arenas share a lot, since layered architecture and modularity principles apply just as much to the body of the robot as they do to its brain—with the notable difference that sometimes you have good reason not to follow those principles in the hardware. In other words, there is no excuse for badly organized software! We used a short but complete program written in NQC to put these principles into practice, showing how they can improve the readability, reusability, and testability of your code.



Chapter 7

Playing Sounds and Music

Solutions in this chapter:

- Communicating through Tones
- Playing Music
- Converting MIDI Files
- Converting WAV Files

Introduction

The RCX features an internal speaker and the hardware necessary to drive it, thus making your robot able to produce sounds. Do not underutilize this feature! It not only offers you a fun way to give your robots a more defined personality, but gives you a simple communication protocol which will help in testing and debugging your programs.

This is why we decided to devote a book chapter to playing sounds and music with the RCX, even though the topic is more related to programming than to building techniques. However, as we explained in Chapter 6, when you are dealing with robotics, the two matters are seldom separable. For some of the robots described in the second part of the book, sounds are an important component in their interface with the external world; for others, sounds are an interesting addition that enriches their behavior.

If you are not familiar with musical terminology or audio file formats, you might find topics in this chapter a bit complex. But the prize is worth the effort, because the techniques explained here open exciting new opportunities in your robot world. You will discover how to use simple tones, how to write melodies, even how to convert digital audio files into sound effects that can be incorporated into your program!

Communicating through Tones

As we explained in the introduction, the RCX features an internal speaker. There is little evidence of it on the outside: The RCX 1.5 has two very small slits on the sides with the LEGO logo stamped on it, from which the sound emanates. The sound system of the RCX is designed to be accessed from your program; you are not allowed to alter the volume of the speaker, which is predefined, but you have full control over the frequency (pitch) and the duration of the notes. The language Not Quite C (NQC), which we will be using in our examples, includes two basic instructions on how to produce sounds, called **PlaySound** and **PlayTone**. Through the **PlaySound** command, the RCX can output one of six predefined sound patterns, such as a short click, a double beep, or a short sequence of tones:

```
PlaySound(SOUND_CLICK);
PlaySound(SOUND_DOUBLE_BEEP);
PlaySound(SOUND_UP);
PlaySound(SOUND_DOWN);
```

The **PlayTone** command plays a single note of a given pitch (in Hertz) and duration (in hundredths of a second). The following statement plays a tone of 262 Hertz for half a second:

```
PlayTone(262,50);
```

The RCX is capable of reproducing any frequency from 31 Hertz to more than 16,000 Hertz; however, you will usually limit yourself to the frequencies which correspond to the musical notes (see the table in Appendix C). All the programming languages built over the LEGO firmware offer this same feature, while most of the others include some kind of more or less sophisticated control over sound.

Sounds are the most immediate way your RCX has to inform you about a specific situation. There is, of course, the *display*, but it's not always in sight, especially when your robot is running across the room! There's also the *datalog*, the feature that allows your PC to read values accumulated in a special memory area in the RCX, but to use it you must be sitting in front of your computer the whole time. Sounds, on the other hand, can be emitted by the robot without interrupting any other activities, and can be heard by you even if the robot is out of sight or far away.

Through simple sound patterns you can make your robot inform you that an operation has ended, something has gone wrong, its batteries are low, and much more. It can acknowledge the push of a button, or tell you it's waiting for specific input from you, as in the case of the **Calibration** routine described in Chapter 6. At the 1999 Mindfest gathering of MINDSTORMS fans and professionals at the Massachusetts Institute of Technology (MIT), we built a Tic-Tac-Toe-playing robot—a version of which you'll see in Chapter 20—that used different musical themes to inform its human opponent about the result of the game.

Playing Music

Sometimes a sound pattern can give your creatures a specific character. Could you imagine a *silent* reproduction of the famous R2-D2 droid from the Star Wars saga?

Music can enrich the personality of your robot even more then tone sequences. A wrestling robot probably appears more resolute if, while facing its opponents, it plays Wagner's "Ride of the Valkyries" rather than a Chopin piano sonata or nothing at all. Our LEGO reproduction of Johnny Five from the movie Short Circuit—described in Chapter 18—plays the Saturday Night Fever theme song while dancing—but if you switch off the soundtrack, it becomes simply a robot that moves around swinging its arms and head.

Playing music requires that you patiently code every single note into your program. LEGO RCX Code is not a suitable tool for melodies longer than just a few notes, but with other textual languages, like NQC, you can write and store very long songs.

Every note in the song requires two attributes: *pitch* and *duration*—the first expressed by a frequency and the second by a time. You must introduce delays between the notes to let the CPU wait out the note's duration before playing the next note.

```
PlayTone(440,50);
Wait(50);
PlayTone(220,100);
Wait(100);
```

In this example, the RCX plays an A (440 Hertz) that's half a second long, waits for the note to finish, then plays another A (220 Hertz) one octave below the previous note for one second.

The RCX is limited to playing a single note at a time, thus we say it's a *monophonic* device. There's no chance to play chords, which require two or more notes played at the same time, but you can adjust note timing to get various effects. In our previous example, the duration of the first note filled the entire interval before the second note, thus producing a *legato* effect. You can just as easily get a *staccato* effect—shortening the duration of the note inside the interval produced by the **Wait** statement—by introducing a pause with no sound between the two notes:

```
PlayTone(440,10);
Wait(50);
PlayTone(220,100);
Wait(100);
```

Coding a melody by hand is a long and tedious task. What happens if when you're finished you discover that the execution is faster or slower than what you intended? Unfortunately, you'd have to go back and change all the time intervals. A better approach takes advantage of a feature that all textual programming environments offer: the definition of *constants*. Using constants you can make all the intervals relative to a specific duration that controls the execution speed:

```
#define BEAT 50
PlayTone(440, BEAT);
Wait(BEAT);
PlayTone(220, 2*BEAT);
Wait(2*BEAT);
```

This code behaves exactly like our first example, but you'll see that by having defined a constant, the code is clearer and easier to maintain, simply changing the value of **BEAT** to change the overall speed. We can extend the usage of constants to include note frequencies as well, making our code more readable:

```
#define BEAT 50
#define A3 220
#define A4 440
PlayTone(A3, BEAT);
Wait(BEAT);
PlayTone(A4, 2*BEAT);
Wait(2*BEAT);
```

You can also patiently define a table of constants for all the notes, so you can reuse it in many different programs:

```
#define C1  33
#define Cs1 35
#define D1  37
#define Ds1 39
//...
#define C4  262
#define Cs4 277
//...
#define B8 7902
```

We coded, for example, the D# note as **Ds** (D sharp) because most languages don't allow the use of special symbols like **#** in the names of constants and variables. Don't worry about the length of this table, because constants get resolved by the compiler and don't change the length of your actual code or the space it takes up in memory.

Creating a soundtrack for your robot is a typical example of where multi-tasking proves to be really helpful. You will typically enclose your song in a separate *task*, starting and stopping it from the main task, as required by the situation.

Converting MIDI files

By using constants, your program becomes more clear, but you don't save any time in coding your melody. You still have to write the notes one by one. The good news is that some tools can do some or all the work for you. The Bricx Command Center, for instance, lets you click notes on a virtual piano keyboard on the PC screen, generating the corresponding NQC code for you. A more complete solution comes from the conversion of standard musical files.

The Musical Instruments Digital Interface (MIDI) is a complex standard that includes communication protocols between instruments and computers, hardware connections, and storage formats. A MIDI file is a song stored in a file according to the format defined by this standard.

MIDI files have achieved incredible success among professionals, amateurs, and instrument manufacturers, and are by far the most preferred way for musicians to exchange songs. For this reason, you can easily find virtually any song you're looking for already stored in a MIDI file.

But what is a MIDI file? It is simply a sequence of notes to play, their duration, their intensity, and, of course, a code that denotes the instrument to be used. Thus a MIDI file is not an audio file. It does not contain digital music like CDs, WAV files, MP3 files or other common audio formats. Rather, it contains instructions for a player (either a human being or a machine) to reproduce the song, almost a score, to be performed by actual musicians. And, as with a real score, the result rests heavily on who actually performs it. For MIDI files, this means that the output depends on the device which renders the music: with a professional MIDI expander you can get impressive results, while execution of the notes by a low-end PC audio card will probably be very poor. What makes MIDI files so interesting to musicians is that they are easy to read and edit (with special programs) in terms of standard musical notation.

So, the key question is: Is there a way you can render MIDI files with the RCX? Though you cannot *import* them directly to the RCX, there's a very nice utility that can convert any MIDI file into the proper code: MIDI2RCX, a free conversion utility developed by Guido Truffelli. It currently runs on Windows machines only, producing either NQC or legOS code, but Truffelli plans to implement more target languages. You can find it at Truffelli's site (see Appendix A).

Before going into the details about how to use it and what it can do for you, there's another characteristic of MIDI files you must be aware of. The notes inside a MIDI file are grouped into *channels*, and each channel is assigned to the *instrument* meant to reproduce those notes. For example, channel 1 could be assigned to an Acoustic Piano, channel 2 to a Bass, channel 3 to a Nylon String Guitar, and so on. Channel 10 is always assigned to Drums, while channel 4 is usually, but not always, assigned to the melody line, that is, the notes sung by the vocalist or played by the leading instrument. As we explained earlier, the RCX has monophonic sound capabilities and cannot reproduce more than a note at a time, so you have to choose carefully the notes it plays. Before you start converting a MIDI file into code straight away, we suggest you do some exploring using a specific software to see which channel could better render the idea of the song. There are many commercial products which are capable of manipulating MIDI files in almost every possible way, but you don't actually need all the power and complexity they provide. The Internet is crammed with freeware and shareware programs perfectly suitable for the task of identifying the best single channel to be converted into instructions for the RCX. You open your MIDI file with the editor, mute all the channels except one in turn, and decide which one to use. If you feel at ease with the MIDI editor, you can cut away some notes from the selected channel, since you probably don't need the whole song, only a chunk of it, the part that contains the refrain or main theme. If you do this through editing, you will save the modified MIDI file, of course.

NOTE

You can save a lot of work if you find a MIDI file targeted to cellular phones. These typically have sound reproduction limits very similar to those of the RCX.

Now you're ready to use MIDI2RCX. It is a console application, not a graphic interface, so you need to run it from a Command Prompt window. It requires the name of the MIDI file, and two optional parameters that specify the channel to convert (it defaults to *all*) and the target language (it defaults to *legOS*). Your typical command will be something like this:

```
c:\midi2rcx>midi2rcx letitbe.mid 4 nqc
```

where *letitbe.mid* is your original MIDI file, *4* is the converted channel, and *nqc* the target language. With this command, MIDI2RCX will produce a file named letitbe.nqc containing plain NQC code ready to be compiled, downloaded to your RCX and executed, or more likely, pasted into your own program. We strongly advise you against converting all the channels: The result will be almost unrecognizable.

Converting WAV Files

Guido Truffelli also wrote a WAV2RCX application that converts WAV files into NQC or legOS instructions. Unlike MIDI files, WAV files contain digitalized audio ready to be executed. If you are familiar with graphic file formats, you can think of MIDI files like *vector* graphics, while WAV files resemble *raster* graphics.

Sequencing MIDI files on the RCX is a challenging task. Playing a WAV file, however, is a lot more challenging. As far as we know, no one has succeeded in getting very good quality. Most likely, the RCX audio hardware has limits that aren't easy to overcome.

Truffelli's program adopts a simple strategy that leads to good results with many WAV files: It splits the source into small intervals and for each of these computes the dominant frequency using an algorithm called FFT; it then converts these frequencies into RCX program statements using the same approach as MIDI2RCX. This is not enough to make your RCX speak, but works well with simple audio patterns like the ding.wav or ringing.wav files included in the Windows system. WAV2RCX is a prized tool with which you can equip your robots with sounds in the best science fiction tradition: laser guns, jump sparks, and buzzing!

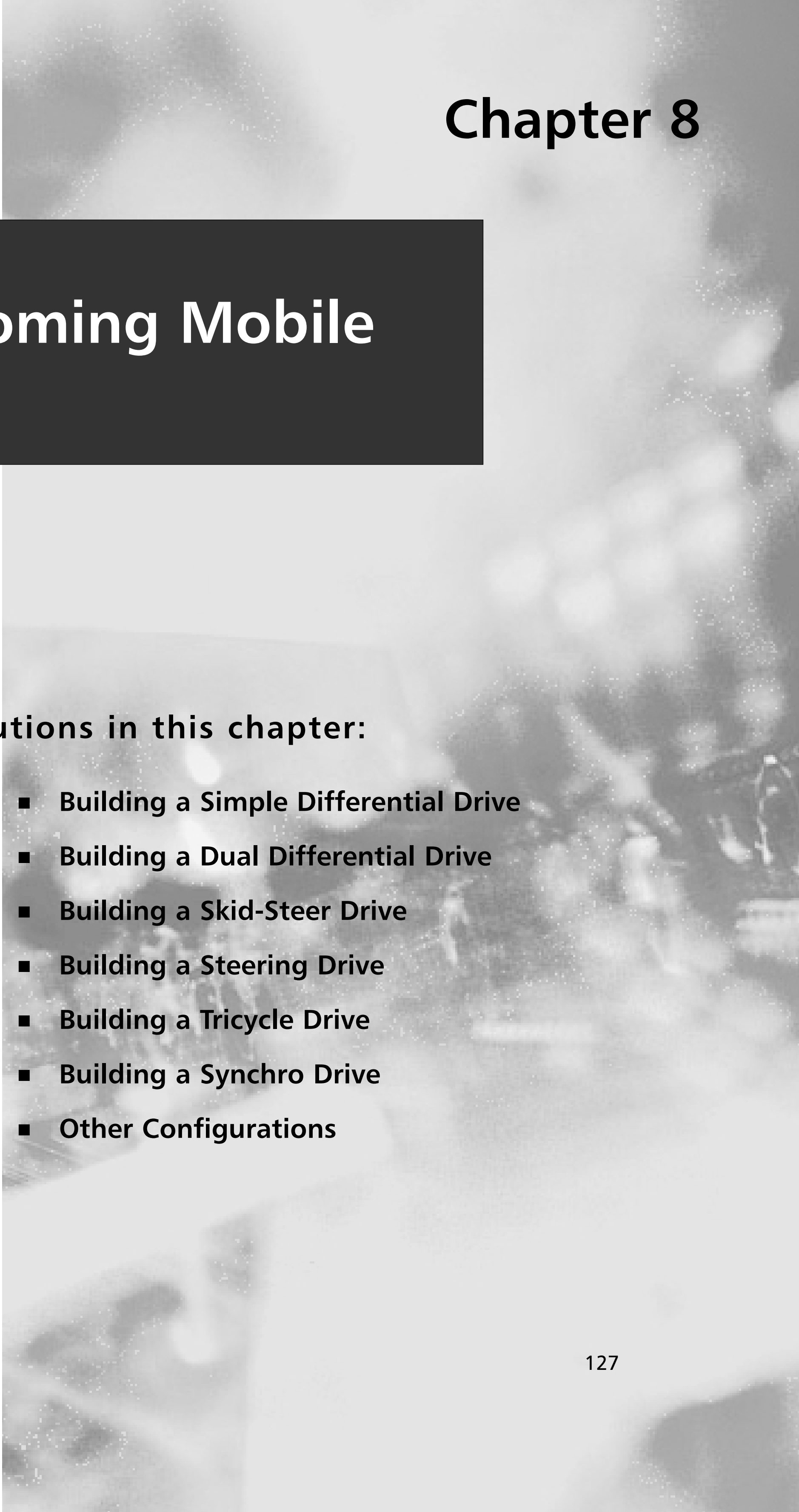
Summary

The purpose of this short journey into the sound system of the RCX was to show that, despite its strong limitations, it's still an invaluable resource. It can support you in debugging, return information in the form of sounds of different patterns or frequencies, or complete the personality of your robots.

NQC offers two commands to control the sound system: **PlaySound** to perform predefined sound patterns, and **PlayTone** to play any note of a desired pitch for the desired duration. While **PlaySound** is suitable for most user interfacing needs, **PlayTone** offers finer control and lets you create melodies.

Thanks to the work of independent developers, you can convert some of the most common digital audio formats straight into NQC instructions. Considering the hardware limitations of the RCX, MIDI files translate very well and are the ideal candidates to provide your robots with a musical soundtrack. The conversion of WAV files, on the other hand, present greater difficulties and offer poorer results; nevertheless, they can equip your robot with amazing sound effects.

More than one robot in this book relies on sounds as a relevant feature. For example, the Tic-Tac-Toe and Chess players of Chapter 20 beep to inform the user they are ready for input, and in the Flight Simulator of Chapter 24 the sound system is entrusted with an essential part of the simulation: reproducing the noise of the engine. Other robots, which can work without sound, would benefit a great deal from some sound effects—good examples of this are the animals and droids of Chapters 17 and 18. In Chapter 21, we will take a different approach, offering ideas about how to build robots capable of playing instruments themselves!



Chapter 8

Becoming Mobile

Solutions in this chapter:

- **Building a Simple Differential Drive**
- **Building a Dual Differential Drive**
- **Building a Skid-Steer Drive**
- **Building a Steering Drive**
- **Building a Tricycle Drive**
- **Building a Synchro Drive**
- **Other Configurations**

Introduction

Most robots are designed with some kind of mobility in mind. Motion makes your creatures animated and "alive," and offers a limitless number of interesting, fun, and challenging projects with which to test your creativity and skills. Most mobile robots belong to one of two categories: *wheeled* robots or *legged* robots. Though legs provide an effective way to move on rough terrains, wheels are generally much more efficient on smooth surfaces.

In this chapter, we will survey the most common wheeled mobility configurations, discussing some of their pros and cons. Please bear in mind that the chassis shown in the following examples are designed to highlight the details of gearings and connections, and for this reason, many of them need some reinforcement to be used in actual robots.

Building a Simple Differential Drive

If you have built some of the robots described in the LEGO Constructopedia, or put together the test platform outlined in Chapter 5, you're already familiar with the *differential drive* architecture. It has so many advantages, particularly in its simplicity, that it's by far the most often used configuration for LEGO mobile robots.

A differential drive is made of two parallel drive wheels on either side of the robot, powered separately, with one or more casters (pivoting wheels) which help support the weight but that have no active role (Figure 8.1). Note that it is called a differential drive because the robot motion vector results from two independent components (it's of no relation to the differential *gear*, which isn't used in this configuration).

When both the drive wheels turn in the same direction at the same speed, the robot goes straight. If the wheels rotate at the same speed but in opposite directions, the robot turns in place, pivoting around the midpoint of the line that connects the drive wheels. Table 8.1 shows the behavior of a differential drive robot according to the direction of its wheels (assuming that when it's in motion they run at the same speed).

Figure 8.1 A Simple Differential Drive

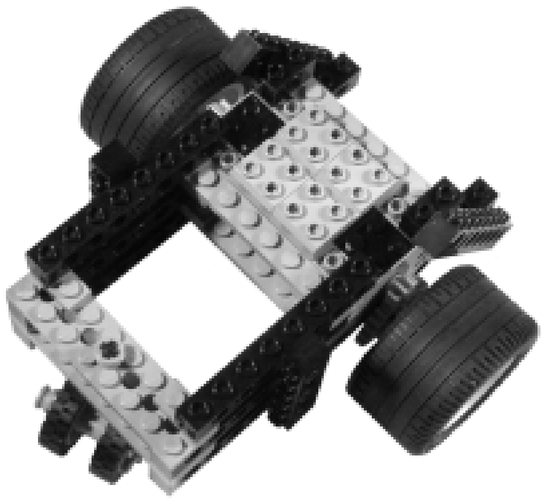


Table 8.1 Behavior of a Differential Drive Robot According to the Direction of Its Wheels

Left Wheel	Right Wheel	Robot
Stationary	Stationary	Rests stationary
Stationary	Forward	Turns counterclockwise pivoting around the left wheel
Stationary	Backward	Turns clockwise pivoting around the left wheel
Forward	Stationary	Turns clockwise pivoting around the right wheel
Forward	Forward	Goes forward
Forward	Backward	Spins clockwise in place
Backward	Stationary	Turns counterclockwise pivoting around the right wheel
Backward	Forward	Spins counterclockwise in place
Backward	Backward	Goes backward

At different combinations of speed and direction, the robot makes turns of any possible radius. This maneuverability, the capability to turn in place in particular, makes the differential drive the ideal candidate for a broad class of projects.

Add to this the fact that it is very easy to implement, and you can understand why more than 50 percent of all mobile LEGO robots belong to this category.

If *tracking* the robot position is one of your goals, again the differential drive is a good candidate, requiring very simple math. (We'll discuss this later in the book.)

There's only one real drawback to this architecture: It's not easy to get your robot to move in a perfectly straight line. Because no two motors have exactly the same efficiency, you will always have one wheel turning a bit faster than the other, thus making your robot turn slightly left or right. In some projects, this isn't a problem, particularly those programmed for continuous route correction, like following a line or finding a path through a maze. But when you want your robot to simply go straight in an open space, this problem can be really frustrating.

Keeping a Straight Path

There are many ways to maintain a straight path when using a simple differential drive. The easiest approach involves reducing the effect by choosing two motors with similar speeds. If you have more than two motors, try finding a combination with the closest matching speeds. This won't guarantee your robot actually goes straight, but it can reduce the problem to a tolerable level. We have a friend who measured the speed of his motors under a small load, and wrote the actual rpm on the bottom of each one with a permanent marker to be able to combine them with satisfactory performance.

A second simple way involves adjusting the speed via software. As described in Chapter 3, your program can control the power of each motor. You can trim the power level of the faster motor until you get an acceptable result. The problem with this approach is that when the load changes (when the robot runs on different terrains), the power levels required to maintain speed will change.

Using Sensors to Go Straight

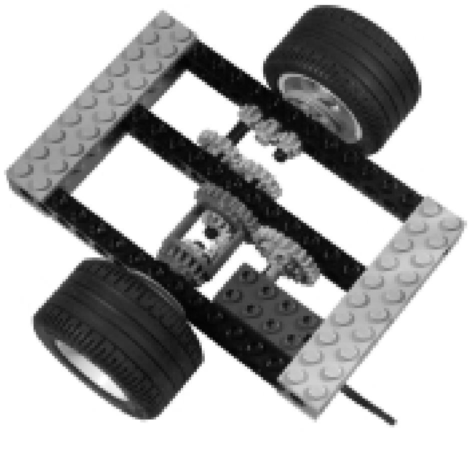
A more sophisticated approach that has several positive side effects requires you to introduce a feedback mechanism into your system, thus controlling each wheel with sensors and adjusting their speed according to the readings. This is what most of the "real life" differential drives do. You can attach to each drive wheel an encoder that counts rotations, and then control the power level in your software to compensate for the difference in the number of turns. The LEGO rotation sensor is ideal for this task. Connect one to each wheel and measure the difference in counts, then stop or slow down the faster of the two for a while to keep the counts equal. One positive side effect is that you can use the same sensors to detect obstacles utilizing the technique described in Chapter 4. If a motor is on but the wheel doesn't rotate, you can deduce your robot is stuck against something. Another benefit is that you can use the rotation sensors to perform turns of a precise angle. Finally, they provide the basic equipment to make your robot compute its position using a technique called *odometry* which we'll discuss later in Chapter 13.

Using Gears to Go Straight

If you have only one rotation sensor, there's a little trick you can use to control the *difference* in speed between the drive wheels instead of the *actual* speed of the wheels. Recall our discussion of the differential gear in Chapter 4. You can use it to add and subtract. If you connect the drive wheels with a differential so that one wheel enters the differential with a direction that's inverted with respect to the other, the body of the differential itself should stay still when the wheels rotate at the same speed.

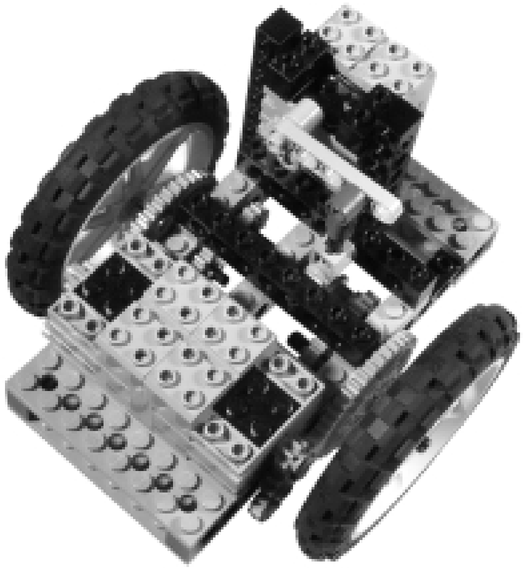
If there is any difference in speed, the differential gear rotates and its direction tells you which wheel is turning faster. Figure 8.2 shows a possible setup (a bit tricky, isn't it?). We strongly suggest you build this chassis even if you don't have a rotation sensor, because the mechanism is instructive and fascinating by itself. We omitted the motors and any reinforcing beams to keep the picture as clear as possible, but in your implementation you should add two motors, each one acting on its wheel like in a standard differential drive. The purpose of the geartrain on the right is to reverse the rotation direction of the axle that enters the differential gear, at the same time keeping the original gear ratio. The rotation sensor, meanwhile, connects to the body of the differential gear to detect whether it turns.

Figure 8.2 Monitoring the Difference in Right and Left Wheel Speed with a Single Rotation Sensor



A more radical solution is to lock the wheels together when you need to go straight. This system is very effective, making your robot go perfectly straight, but it requires a third motor to activate the locking system as well as some additional gearing, which makes the solution less than compact. Figure 8.3 shows an example of a locking mechanism that requires special parts: a dark gray 16t gear with clutch, a transmission driving ring, and a transmission changeover catch, which combine in a sort of clutch mechanism (Figure 8.4). That special gear has a circular hole instead of the standard cross-shaped hole, thus it rotates freely on the axle. The driving ring should then be mounted on an axle joiner. When you push the driving ring into the gear (with the help of the changeover catch), the gear becomes solid with the axle.

Figure 8.3 A Lockable Differential Drive



You can also use the setup shown in Figure 8.2, inserting a motor in place of the rotation sensor. Recall from Chapter 4 that a motor works as an electric brake, too: In its *off* state, it opposes motion, while in the *float* state it is still not powered but free to turn. In this solution, you will not power this motor, but rather operate it as an electric brake for the body of the differential. When you brake the motor in *off* state, the differential hardly turns, making your robot go straight. On the other side, with the motor in *float* state, the differential can rotate and the robot is able to turn. Table 8.2 summarizes some of the possible combinations: The rule is that when the left and right motor run with different directions, the differential gear lock motor must be in float state.

Figure 8.4 The 16t Gear with Clutch, the Transmission Driving Ring, and the Transmission Changeover Catch

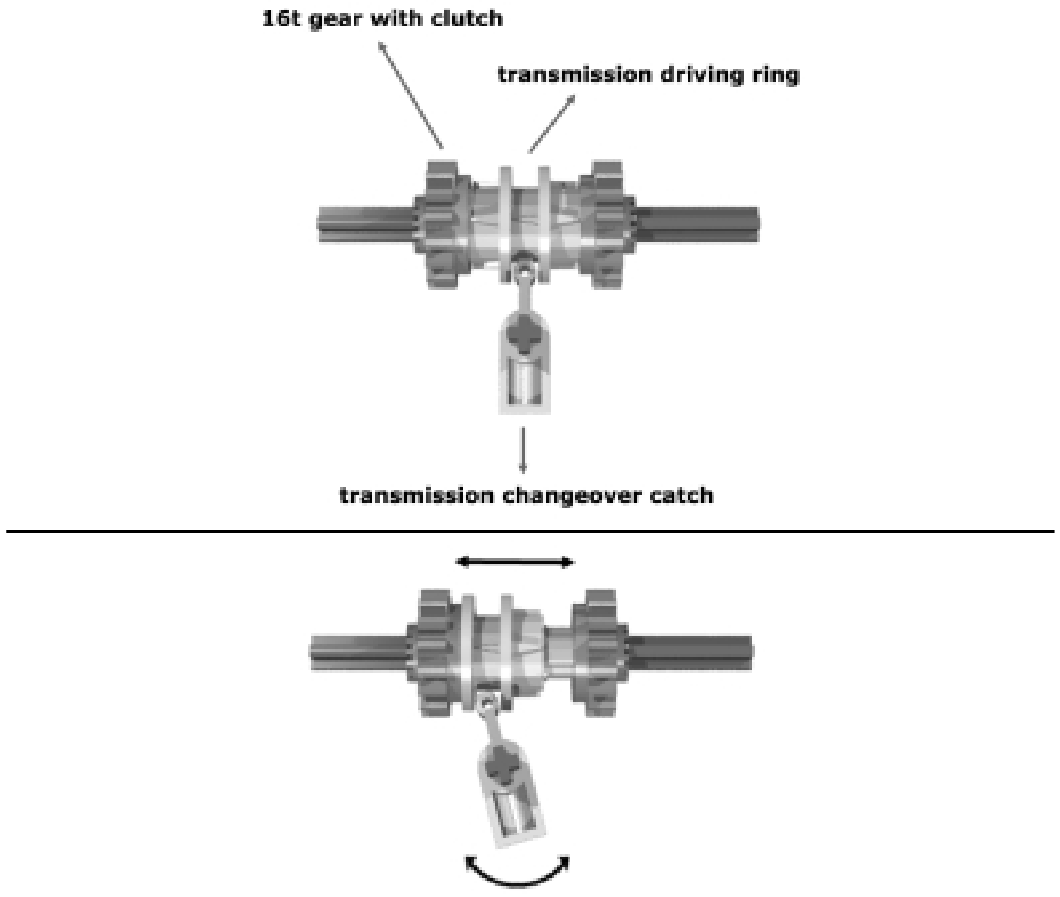


Table 8.2 How to Control a Differential Drive Robot Provided with Electric Differential Gear Lock

Left Wheel Motor	Right Wheel Motor	Differential Gear Lock Motor	Robot
Off	Off	Off	Rests stationary
Forward	Forward	Off	Goes straight forward
Forward	Reverse	Float	Spins clockwise in place

Continued

Table 8.2 Continued

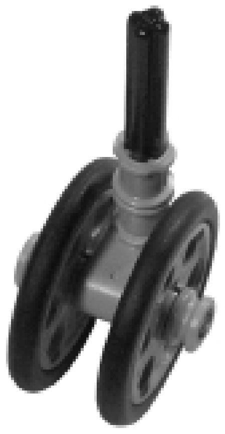
Left Wheel Motor	Right Wheel Motor	Differential Gear Lock Motor	Robot
Reverse	Forward	Float	Spins counterclockwise in place
Reverse	Reverse	Off	Goes straight backward

Consider that even in float mode the motor has significant mechanical resistance, so the robot will not turn as quickly and the drive motors will be under more stress when turning.

Using Casters to Go Straight

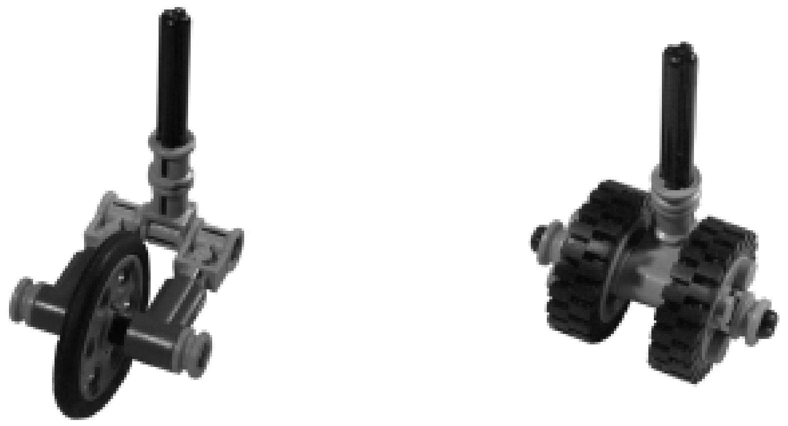
Casters are another key factor in getting your differential drive moving and turning smoothly. Most often, though, they are not given enough consideration. The LEGO Constructopedia suggests the caster shown in Figure 8.5, but we will take the liberty of saying that it is a poorly designed caster. It uses two wheels coupled on the same axle. You already know from Chapter 2, however, that this configuration doesn't allow the wheels to turn independently. Keep the assembly gently but firmly pressed on a table, and try to rotate it in a tight turn—it doesn't turn very well, does it? In fact, unless you let one of the wheels skid, it doesn't turn at all.

Figure 8.5 The Coupled Caster from Constructopedia



The casters shown in Figure 8.6 get much better results. The one on the left uses a single wheel, thus avoiding the problem entirely. The one on the right, which is more solid, uses two free wheels that allow the caster to turn in place without friction or slippage problems. The difference is in the wheel hubs. In the assembly on the left, the axle turns with the wheel, while the one on the right has the wheels spinning on the axle.

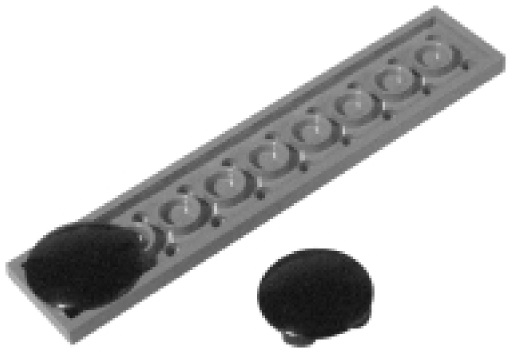
Figure 8.6 Casters Designed to Avoid Skidding



The choice of using one or more casters depends on what task the robot is designed for. A single caster is enough for most applications, but two casters at the front and rear of the robot are a better option when stability is important.

In some cases, as with a simple robot of limited weight that has a smooth surface on which to navigate, you can substitute the caster with *inverted round tiles* or other parts that provide limited friction when contacting the floor (Figure 8.7).

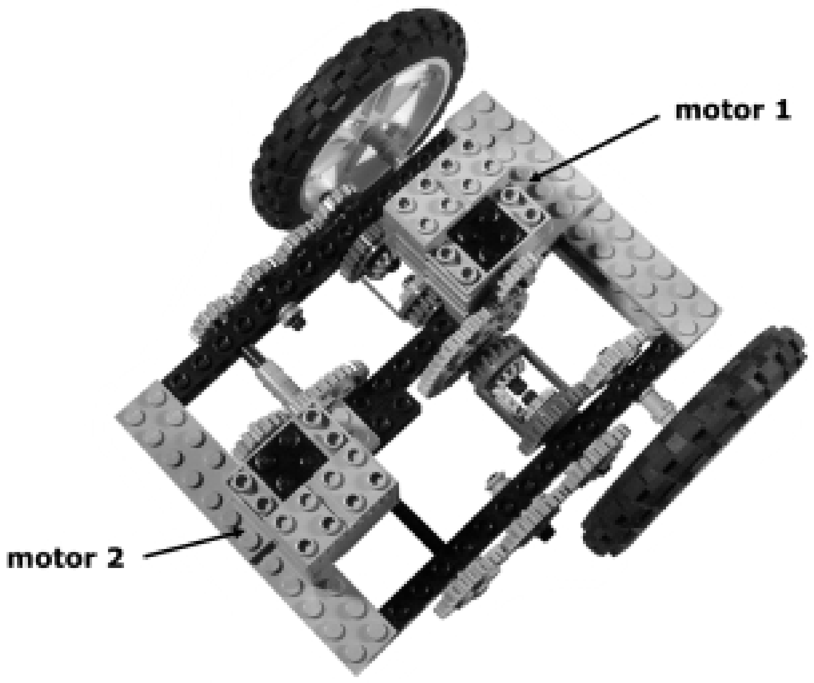
Figure 8.7 Inverted Round Tiles Can Replace Casters



Building a Dual Differential Drive

A *dual differential drive* is an improvement on the simple differential drive. It is designed to mechanically solve the problem of following a straight path, and uses only two motors (see Figure 8.8). Its gearing setup is a bit complex, and relies again on the differential gear—two of them to be precise (see Chapter 9 about getting supplementary parts).

Figure 8.8 A Dual Differential Drive

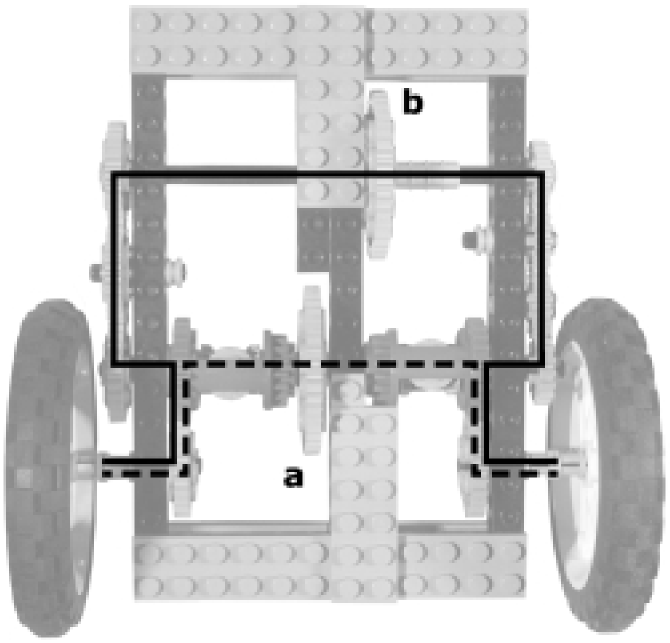


The dual differential drive inverts the common use of the differential gear. Normally, the wheels are connected to the *axles* coming out of the differential gear, while in this case, the wheels are connected to the *body* of two differential gears. In Chapter 4, we explained that a differential gear can be used to mechanically add or subtract two independent motions; to do this, use the axles coming out of the differential gear as *input*, and the body of the differential gear will move according to the result of their algebraic sum (a sum that takes direction into account).

In this setup, both motors provide one input to the two differential gears. The trick is that one of the motors rotates the input axles of the two differentials in the same direction, while the other is geared to rotate the other input axles in opposite directions. To operate a dual differential drive, you will normally use just one of the motors, keeping the other braked.

In Figure 8.9, you see the same assembly as in Figure 8.8, but without motors. When motor 1 rotates the 40t gear A, and motor 2 keeps B braked, motion gets transmitted along the dotted line path in the picture, the two differentials rotate in sync and the robot goes straight. On the other hand, keeping motor 1 off and consequently A braked, and operating motor 2 to rotate B will make the motion transfer along the solid line and the differentials rotate at the same speed, but in opposite directions. The result is that the robot spins perfectly in place.

Figure 8.9 The Dual Differential Drive Dissected



Thus, you would normally use a single motor at a time, one for going straight, the other for turning. Nothing bad happens if you power both motors—depending on their direction. One of the differentials will receive two opposing inputs, nullifying them and remaining stationary, while the other adds two inputs, doubling the resulting speed, in which case the robot pivots around the stationary wheel, exactly like a simple differential drive does when one of its wheels moves and the other rests.

A very nice feature of the dual differential drive is that with a single rotation sensor you can precisely monitor any kind of movement of your robot. Couple the sensor to one of the wheels (it doesn't matter which one). When the robot goes straight, you can use the sensor to measure the traveled distance, and when the robot turns in place, the sensor measures the change in heading.

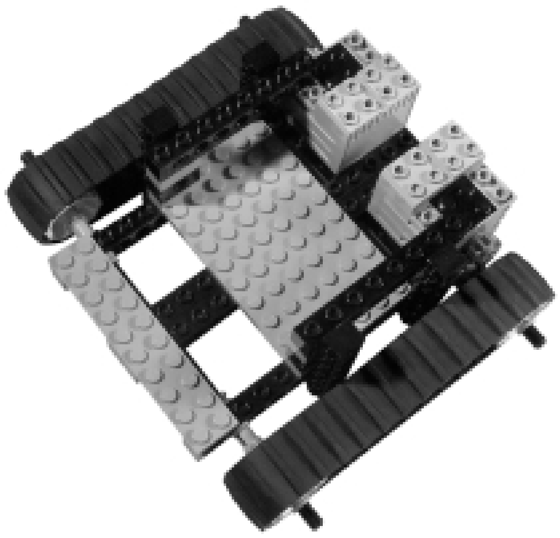
Of course, remember we said earlier that there are no free lunches in mechanics. In other words, this ingenious configuration has its drawbacks. The first, obviously, is its complexity. We deliberately built our example flat on a plane to keep all the connections easy to understand; however, you can build more compact versions by stacking some of the gearing (it will still require all those gear wheels, maybe just a couple less). The complex gearing leads to the second side effect: our nemesis *friction*. To make matters worse in this case, you have just a single motor to fight it!

Building a Skid-Steer Drive

A *skid-steer drive* is a variation of the differential drive. It's normally used with tracked vehicles, but sometimes with 4- or 6-wheel platforms as well. For tracked vehicles, this drive is the only possible driving scheme. Good examples of skid-steer drives in real life are excavators, tanks, and a few high-end lawnmowers.

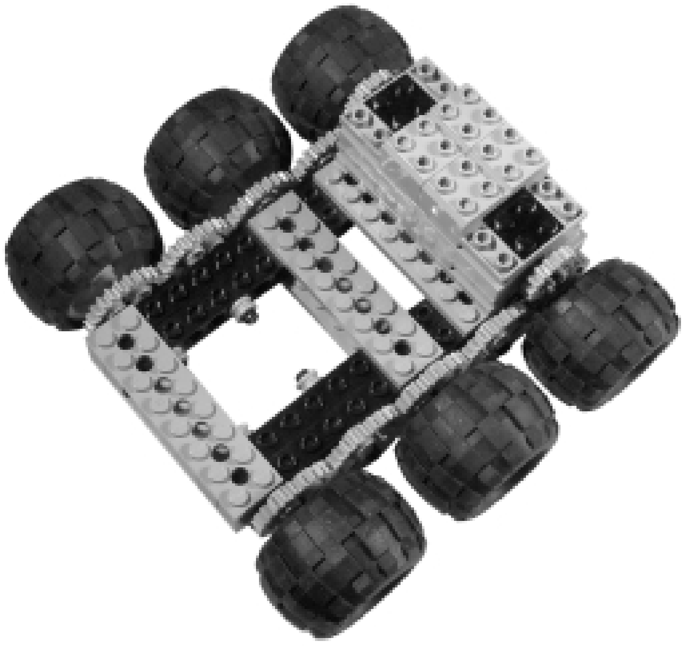
Figure 8.10 shows a simple tracked skid-steer drive. Each track is powered by its independent motor, that mounts an 8t gear and meshes a 24t gear connected to the track wheel. The front track wheels need not be powered.

Figure 8.10 A Tracked Skid-Steer Drive



A wheeled skid-steer drive requires a trickier setup. You must transmit the power to all the wheels, otherwise your platform won't turn smoothly, or might not even turn at all. The model shown in Figure 8.11 uses a row of five meshed 24t gears for each side, all of them receiving power from two motors like in the tracked version. Every wheel axle mounts its gear, and they are interleaved with idler gears that serve the purpose of transferring motion from one wheel to the other. If you do have enough 24t gears, you can mix them with 24t crown gears, which are exactly the same size. The balloon tires in the picture come from supplementary sets.

Figure 8.11 A Wheeled Skid-Steer Drive



Tracked robots are easy to build and fun to see in action, thus placing them among the favorites of many builders. Just as with differential drives, when the tracks go the same direction, the robot goes forward; differences in their speeds or directions make the robot turn; in-place steering is possible, too. Skid-steer drives also share with differential drives the same difficulties in getting them to move in a straight line.

Here is where the similarities end, and some peculiarities of skid-steer emerge:

- Tracks have a better grip than wheels do on rough floors and terrains, but this is *not* true on smooth surfaces.
- Tracks introduce more friction which uses up some of the power supplied by the motors.
- The unavoidable skidding intrinsic in the nature of these vehicles makes them absolutely unsuitable for applications where you need to determine the position by utilizing the motion of the robot.

Building a Steering Drive

A *steering drive* is the standard configuration used in cars and most other vehicles that features two front steering wheels and two fixed rear wheels. Thankfully, it's suitable for robots too. You can drive either the rear or the front wheels, or all four of them, but the first is by far the easiest solution to implement with LEGO parts, so this is what we'll cover here. Though less versatile than differential drives, and impossible to steer in place or in very tight turns, this configuration has many advantages: It's very easy to drive straight, and very stable on rough terrain.

When building a steering drive robot from the basic MINDSTORMS equipment, you have only one motor to power the drive wheels, because you need the other to steer the front wheels. Thus your steering drive robot will have about half the power of a differential drive one, which can benefit from both motors during straight motion.

In Figures 8.12 and 8.13 you see two simple steering platforms. Apart from implementation details, these two models share the same construction principles. For instance, the rear wheels are connected to the driving motor through a differential gear. As explained in Chapter 2, you cannot avoid the differential if you want your vehicle to turn. A second motor steers the front wheels, providing your robot with a way to change direction. Notice that we used a belt to drive the steering mechanism, taking advantage of its implicit torque-limiting transmission to avoid any damage to the mechanism or the motor if the motor remains on after the steering mechanism has reached one of its limits. You would probably add a sensor to detect the steering position, allowing your robot to control its direction. A single touch sensor is the bare minimum needed—make it close when the steering is centered, so you can use timing to steer the wheels and utilize the sensor to center them back after the turn (Chapter 14 contains an example of this technique).

Figure 8.12 A MINDSTORMS-only Steering Drive

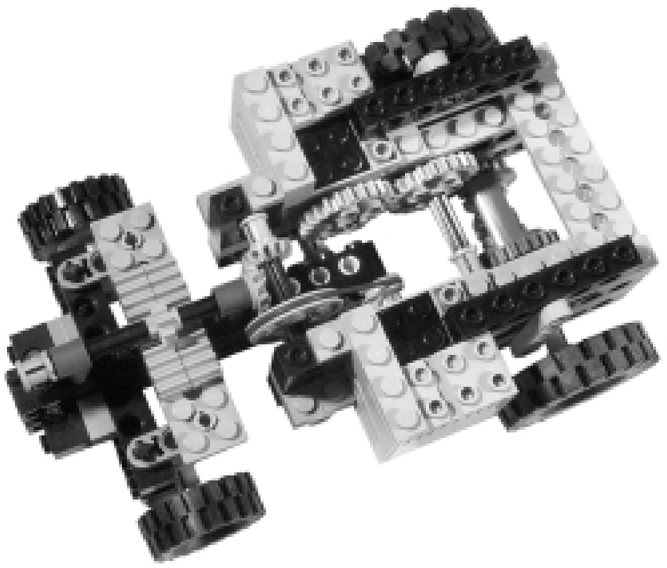
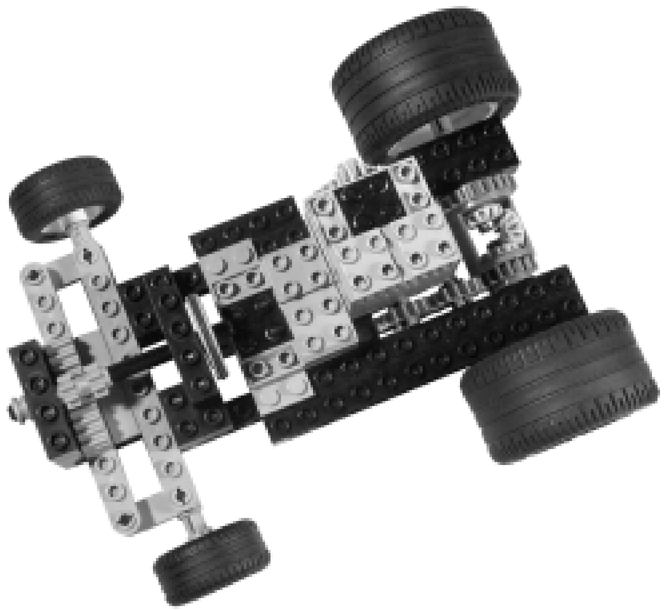


Figure 8.13 Another Steering Drive

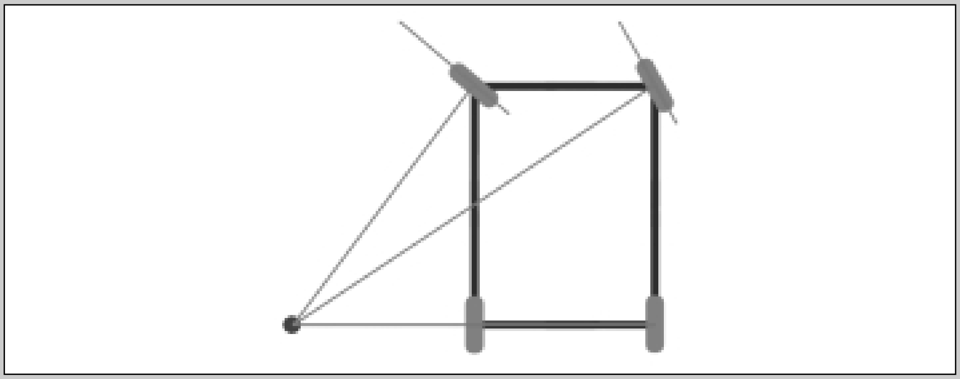


Designing & Planning…

Using Ackerman Steering for Smooth Turns

True-life steering vehicles implement a more sophisticated scheme called *Ackerman steering* (from the name of the person who first studied it). In our simple design, the steering wheels turn at the same angle, but this is not entirely correct—during turns, the inner wheel goes along a tighter bend than the outer one. During large radius turns, the difference is small and its effect negligible. In tight turns, however, the effect becomes quite noticeable, causing one of the steering wheels to skid. Ackerman's steering system is designed to compensate for the different turning angle of the inside wheel, thus eliminating any skidding. The theory says that the vehicle turns smoothly when the "lines" extended from every wheel axle meet and revolve around one common point (Figure 8.14).

Figure 8.14 Ackerman Steering Scheme: The Inner Wheel Turns More than the Outer One



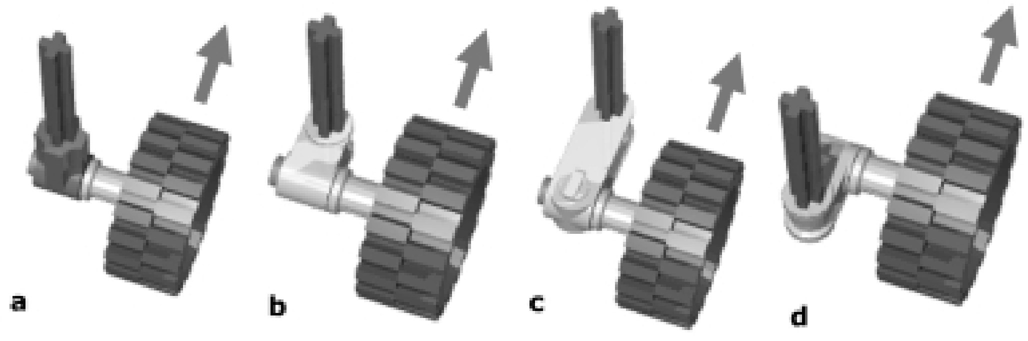
Building an Ackerman scheme with LEGO is definitely possible. Chapter 14 incorporates the prototype of a front-wheel drive that features the Ackerman correction.

Both models employ a *rack and pinion* steering mechanism where an 8t gear (the *pinion*) meshes with a special plate with teeth, a sort of "unrolled gear" (the *rack*). The difference between the chassis in Figure 8.12 and the one in Figure 8.13 is that we built the latter using extra parts that make our life easier: three 1 x 10

TECHNIC plates, two steering arms, and two tiles. These components are designed to be combined together, creating a very simple steering mechanism used in many LEGO TECHNIC car and truck models. In the model presented in Figure 8.12, built only from MINDSTORMS parts, we had to use a 2 x 8 plate, instead of the 1 x 10 ones, and replace the steering arms with a home-made version. The whole front section of the vehicle has been built with the beams oriented studs-front, to provide the necessary support for the wheels and the steering mechanism, but mostly to provide a smooth surface (the side of the beam) which the rack can slide over (you will find more information about this setup in Chapter 14).

When you build the steering assembly, you can move the wheel behind its pivoting axle for self-centering steering (an advisable property in many situations). In version a in Figure 8.15, you see a wheel mounted just below the pivoting axle, which does not effect the steering. If you mount the wheel behind its steering column, friction causes the dynamic forward motion of the car to push the wheels toward the rear, resulting in a self-centering action. Look at the design of a shopping cart, and you will see that the actual wheel contact area is behind the pivoting axis. The more you move the wheel behind the pivoting axis, like in versions b and c, the more self-centering you get. Don't ever mount the wheel in front of the pivoting axle, like in version d. This will make your steering unstable. In fact, the wheel will tend to go toward the rear, causing your car to turn spontaneously.

Figure 8.15 Moving the Wheel from the Pivoting Axle

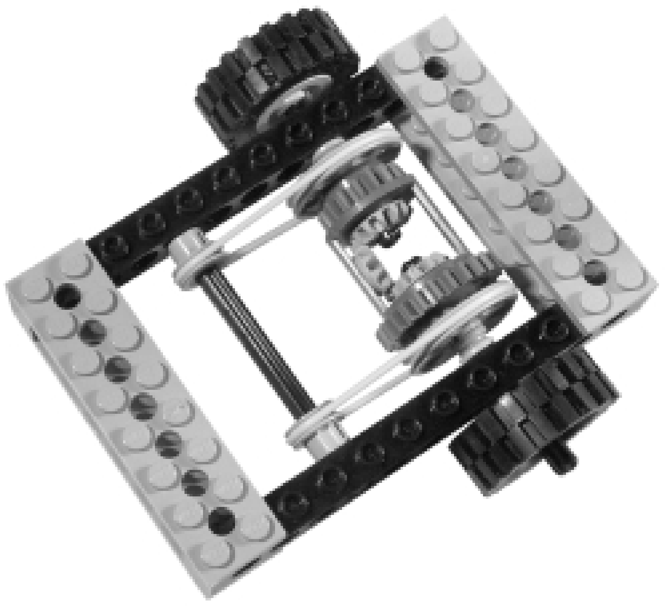


We encourage you to experiment with these concepts, building a simple chassis and exploring the properties of the various assemblies shown in Figure 8.15.

The steering drive is a suitable configuration for rough terrains, since it's very stable on its four wheels. You can improve the grip of the wheels on the ground by using some kind of suspension. It's very important that none of the drive wheels permanently lose contact with the ground, otherwise the differential would find the path of least resistance and transfer all the power to that wheel, resulting in the wheel spinning and your robot becoming immobilized.

A *limited slip differential* can help reduce this problem (see Figure 8.16) by connecting the wheel axles to a common supplementary axle through pulleys and belts. The belts tend to keep the driven axles rotating at the same speed, but during turns they slip a bit on their pulleys, allowing the wheel to adjust their speeds. Should a wheel lose contact with the ground, the belts will still be able to transfer a good portion of power to the other wheel.

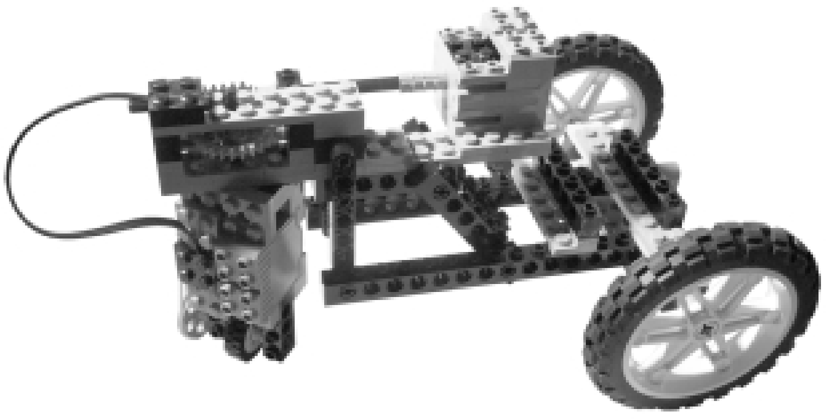
Figure 8.16 A Limited Slip Differential



Building a Tricycle Drive

A *tricycle drive* configuration involves a front wheel that drives and steers and is matched with two passive independent rear wheels which provide stability (Figure 8.17). The peculiarity of this configuration lies in the fact that the front wheel is both powered and steering, giving the robot a high grade of mobility.

Figure 8.17 A Tricycle Drive



You might think that driving the rear wheels instead of the front one would give you the same results, but this is true only for a limited range of steering angles. In fact, like in a steering drive, when narrowing the turn radius, you ultimately reach a point where the rear wheels can no longer convert power into motion. The maximum turning angle that a steering vehicle can reach is when the inner wheel is stationary and the outer one draws a circle around that point. A front-wheel driven tricycle, on the other hand, can manage any steering angle, even when the wheel is perpendicular to the direction of motion of the rear wheels.

Ideally, the driven wheel can rotate 360° to point in any possible direction. This means you should build a system with no constraints on a full turn (an example of this architecture is the mechanism used to drive bumper cars at amusement parks). Our example in Figure 8.14 is capable of rotating the steering a full 360°, but cannot make more than a single 360° rotation due to the wire that connects the motor to the RCX.

In practical applications, a 180° turn is enough to allow the robot any possible movement, because any angle in the range of 180° to 360° is equivalent to an angle in the range of 0° to 180° with the motion reversed. In other words, 210° with the motor in forward motion corresponds to 30° (210 – 180 = 30) with the motor in reverse. As with the steering drive, you will probably use a sensor to detect the position of the steering.

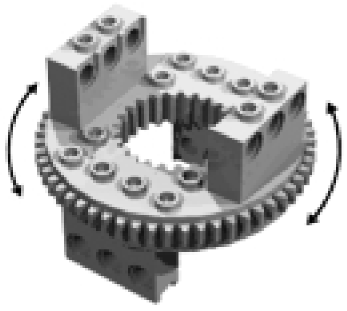
Building a Synchro Drive

A *synchro drive* uses three or more wheels, all of them driven and steering. They all turn together in sync, always remaining parallel, thus the robot changes its direction of motion without changing its orientation.

Synchro drives are quite challenging to build with LEGO parts. Until a few years ago, there was general agreement that it should have been possible, yet nobody had succeeded in the undertaking. Now the barrier has been broken, and if you navigate the Internet, you can find many well-designed LEGO synchro drives.

To make a full 360° synchro drive and avoid any limitations in its turning ability, the key point is to transfer motion along the pivoting axle of each wheel. The simplest approach requires a special part called the *turntable*, a large round rotating platform usually employed in LEGO models to support revolving cranes or excavators (Figure 8.18).

Figure 8.18 The LEGO Turntable



You can attach the wheel to one side, and drive it with an axle that passes through the hole in the center of the turntable. In Figure 8.19, you can see an example of this technique. Notice that the turntable is upside down, because the wheel must be connected to the part of the turntable that gets rotated by the external gear. Because of this, the robot will result in an entirely, or at least partly, studs-down design!

We want our synchro drive robot to be able to change direction in place without moving. To this aim, the two assemblies in Figure 8.19 and 8.20 are similar, but not interchangeable. With the driving axle blocked, the lower part of the turntable should turn smoothly in place—in Figure 8.19 it does, but in Figure 8.20 it doesn't. This happens because the wheel in Figure 8.20 is not centered below the pivoting axle, and so when it changes its direction it has to travel some distance. The gearing in Figure 8.19 makes the wheel rotate in the proper direction, the one that complies with the turn, while the gearing in Figure 8.20 makes the wheel oppose the turn. We realize this is a subtle difference, and we invite you once again to learn by experience, building the two versions by yourself and verifying how they work.

Figure 8.19 A Possible Wheel Assembly for a Synchro Drive

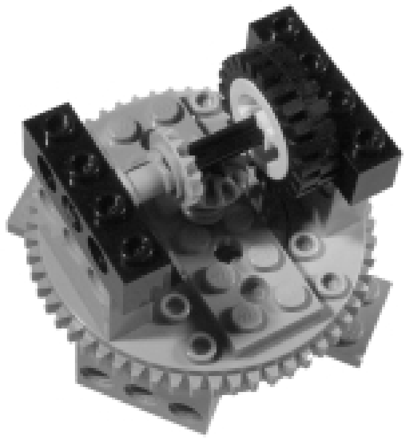
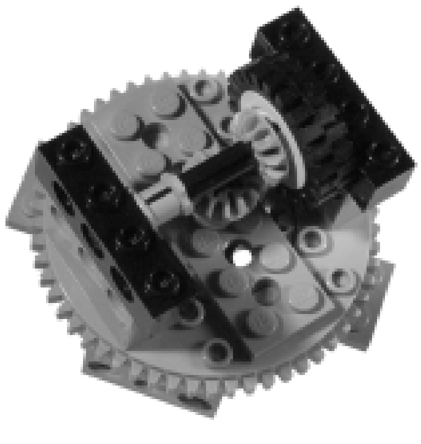


Figure 8.20 Incorrect Version of the Wheel Assembly



To build a complete synchro drive, you need at least three of these turntables. Then you have to connect them so that one motor can drive all the axles at the same time, while another can turn all the wheels in sync.

In Figure 8.21 you see the bottom view of a four-wheeled synchro drive. Notice that we linked the turntables with 8t gears so they all turn together. Powering any one of those 8t is enough to make the robot change direction.

Figure 8.21 A Complete Synchro Drive (Bottom View)

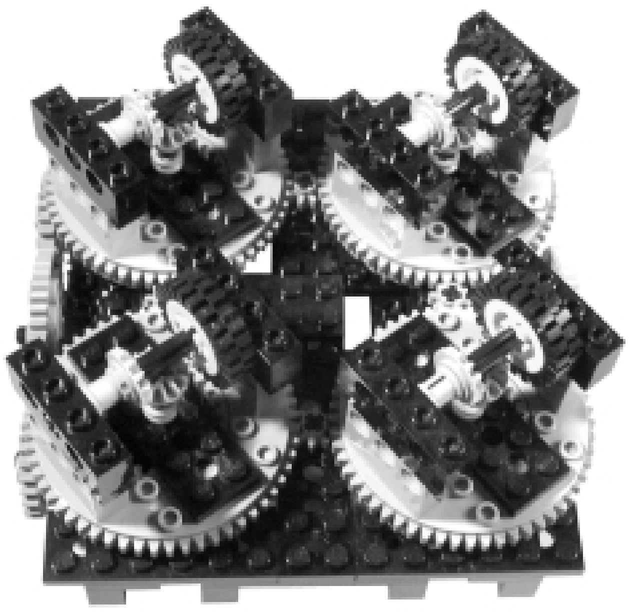


Figure 8.22 shows the top view of the same platform: the large 40t gear (a) drives the wheels through four pairs of bevel gears, while the other 40t (b) is in charge of turning the wheels. To complete this synchro, you have to add two motors to power a and b, possibly using an 8t gear to get a ratio which is capable of reducing the friction introduced by all that gearing.

Synchro drives are quite amazing to see in action, and yours will be no exception. But if you expect it to navigate the room detecting obstacles, your challenge isn't quite over yet: You still have to manage bumpers. In a synchro drive, the concept of "front" and "rear" has no meaning, since the robot can travel using any of its sides as a front. Consequently, you have to place bumpers all around it. As you learned in Chapter 4, if your robot has four sides, you are not compelled to use four sensors for four ports (which your RCX doesn't have). You can connect four touch sensors to the same port, using an OR configuration by which any sensor that gets closed puts the RCX into an "on" state. Or you could simply use a single omni-directional sensor like the one shown in Figure 8.23; the touch sensor is normally closed, but opens whenever the upper axle departs from its default position (kept by the rubber bands). Surround your robot with a ring of tubes or axles, connect the ring to the omni-directional sensor, and that's it!

Figure 8.22 A Complete Synchro Drive (Top View)

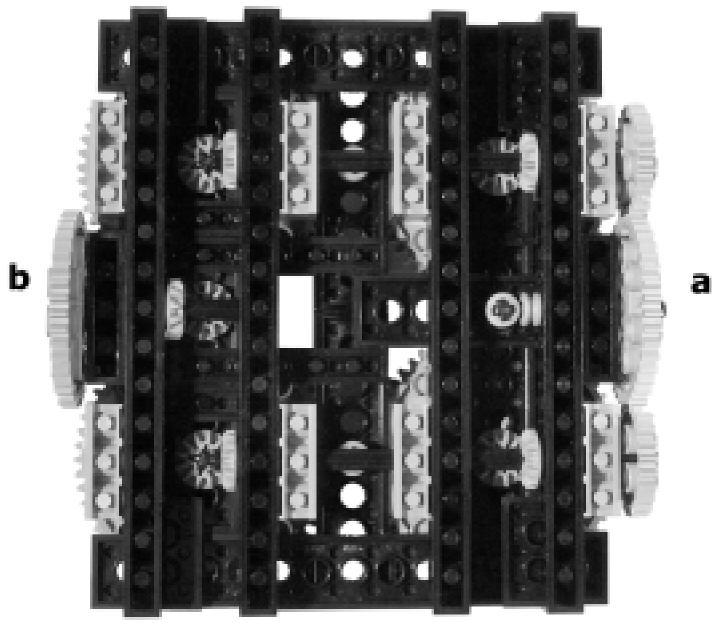
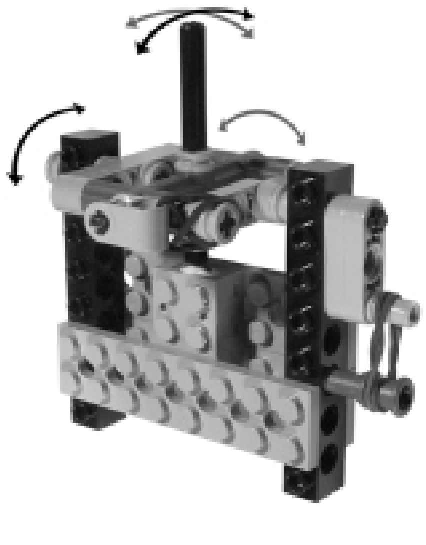


Figure 8.23 An Omni-Directional Touch Sensor



Other Configurations

Our roundup doesn't cover all the possible mobile configurations. There are other more sophisticated or specialized types:

- **Multi-Degree-of-Freedom (MDOF) vehicles** MDOF vehicles have three or more wheels, or groups of wheels, both independently turned and driven. Imagine a synchro drive where each wheel can change its speed and direction with no connection to the others: such a robot would be able to behave like a differential drive, a steering drive, or a synchro drive just by controlling its configuration from the software. Though interesting to study and very versatile in their use, they are also extremely difficult to build and control. In fact, not all of their possible configurations result in a coordinated motion!
- **Articulated Drive** This is very similar to the steering drive, but instead of steering the wheels, it steers a whole section of the vehicle. The front wheels always remain parallel to the front part of the chassis, and the same applies to the rear wheels in regards to the rear portion of the chassis. Nevertheless, the two sections connect through an articulation point that lets them pivot in the middle. This configuration is common in wheeled excavators and other construction equipment.
- **Pivot Drive** Keith Kotay defines a *pivot drive* as a configuration made of a chassis with non-pivoting wheels with a platform in the middle that can be lowered or raised. When the platform is up, the robot moves perfectly straight on its wheels. When it requires turning, it stops and lowers the platform until the wheels don't touch the ground anymore. At this point it rotates the platform to change its heading, then raises the platform again and resumes a straight motion.
- **Tri-Star Wheel Drive** The Tri-Star configuration has been designed for high-mobility, all-terrain vehicles. Each "wheel" is actually an equilateral triangle with wheels in each vertex; the vehicle features three of them for a total of twelve wheels. The wheels turn, and the triangles can also turn like larger wheels. During normal motion, two wheels of each triangle touch the ground, but when a wheel sticks against an obstacle, a complex gearing system transfers motion to the triangular structure, which turns and places its upper wheel past the obstacle. As complicated to build as it is interesting!

- **Killough Platform** Developed by Francois Pin and Stephen Killough, the official name of this mechanical configuration is Omnidirectional Holonomic Platform (OHP). *Holonomy* is the capability of a system to move toward any given direction while simultaneously rotating. While conventional wheeled vehicles aren't holonomic at all, this platform allows for unprecedented mobility. Seen from the top, a Killough drive shows three wheels placed at the vertices of an equilateral triangle. Each "wheel" is a sort of sphere made of actual wheels combined together and used in a quite unconventional way: on their side!

We hope we've made you curious about these configurations, and invite you to find out more about them using the reference material provided in Appendix A. All of them can be built from LEGO parts, and give you further challenges for when the standard configurations shown in this chapter have become old hat.

Summary

This chapter has been quite dense, but we hope we've been able to help you in choosing a drive configuration. When building a mobile robot, different architectures are relevant to its resulting shape, and most importantly, to its performance.

The differential drive is simple and versatile, but can't go straight. The steering drive, meanwhile, goes straight but cannot turn in place. The dual differential drive can do both, but it's more cumbersome and complex to build. Robotics is like cooking: there are many recipes for the same dish, but to be successful you still must know the ingredients well and use them in the right proportions. Of course, don't forget to add the most important ingredient of all: your creativity.

Chapter 9

Expanding Your Options with Kits and Creative Solutions

Solutions in this chapter:

- Acquiring More Parts
- Creating Custom Components
- Creative Solutions When More RCX Ports Are Needed

Introduction

If, by now, you are caught up in robotics, you may feel a bit constrained by the limitations of the MINDSTORMS kit.You want more.What do you perceive most limiting: the number and range of parts, or the fact your RCX has only three input and three output ports? Maybe you would like to use new kinds of sensors, or servo-motors. And why not try out some pneumatic devices?

If the MINDSTORMS was your first LEGO set, you will be pleased to see that there are many additional parts to boost and support your creativity. If MINDSTORMS is an addition to your large collection of LEGO TECHNIC sets, you already know what parts the line includes and probably already have them—but there is a also wealth of compatible non-LEGO custom parts and kits you may never have dreamed of: infrared and ultrasonic proximity detectors, compasses, sound frequency decoders, magnetic switches, and voice recognition units, just to mention a few. In this chapter, we will explore some options for expanding your designs and plans, surveying the most important additions, providing you with information about where and how you can get them, and describing also the most significant non-LEGO custom devices.

Extra parts are not the only way to expand your project ideas. Some mechanical tricks can also help you in getting the most from the limited number of output ports offered by the RCX.You will learn how a single motor can power two or more mechanisms, and how you can apply this trick to some of the mobile configurations we described in Chapter 8.

Acquiring More Parts

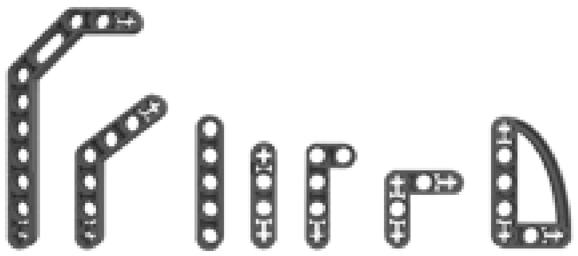
Describing all the components that make up the LEGO world would be a tremendously difficult task.The vast LEGO system includes tens of thousands of different parts, belonging to different themes, but all are easily integrated with each other.That's the beauty of LEGO:You can always find a new use for something that might have been built with a completely different purpose in mind. Whether it be towns, trains, or pirates, any or all of the LEGO themes might add something useful to your set of equipment. Of course, when it comes to building *robotics*, the natural choice is the LEGO TECHNIC line.

Created in 1977 to introduce older children to the world of mechanics and motors, the TECHNIC line developed into a complete system that includes many specialized parts.You are already familiar with the almost 140 varieties found in the MINDSTORMS kit, organized into some of the classes previously mentioned—beams, plates, axles, liftarms, gears, and so on.

Introducing Some Specialized Components

Many other TECHNIC parts come in a broader variety than shown in the MINDSTORMS kit. Liftarms, for example, are increasingly prevalent in recent TECHNIC releases (Figure 9.1). There's an evident trend in this direction, and in fact, some of the newest sets don't include traditional beams or plates at all. They're instead composed only of liftarms, axles, and connectors.

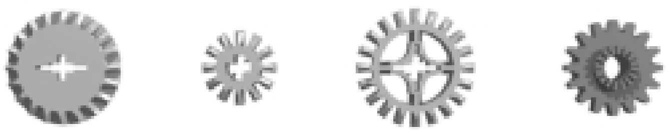
Figure 9.1 Liftarms



Liftarms have many possible uses. We showed in Chapter 5 that they can profitably replace beams to brace the layers of a structure, especially in those cases when you need a vertical lock that remains within the height of the horizontal beams because you have other plates or beams above or below them. Other common transformers include levers and arms, since their cross-shaped holes provide an ideal attachment point for axles when you need to operate them through some kind of mechanism.

In previous chapters, we covered some representatives of the new class of gears not included in the MINDSTORMS kit, like the 20t bevel gear, the 20t and 12t double-bevel gears, and the 16t gear with clutch (Figure 9.2). There isn't currently a service pack specific to gears only, and to increase your inventory, you have to buy TECHNIC models or MINDSTORMS expansion sets, which include many other parts.

Figure 9.2 Gears Not Included in the MINDSTORMS Kit

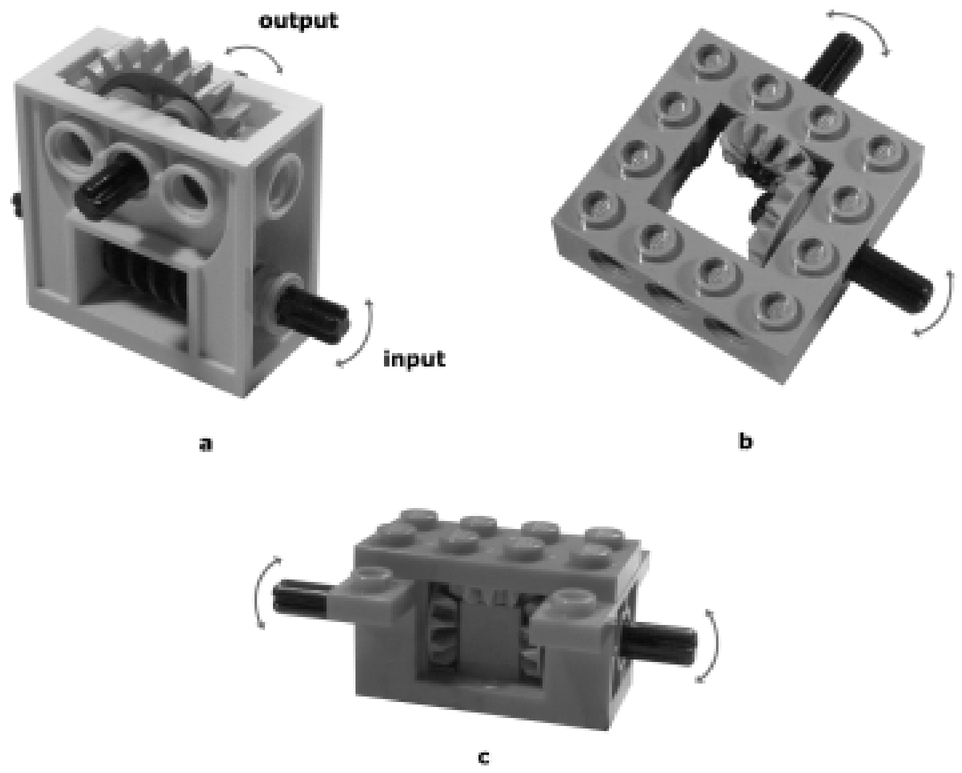


Special components called *gearboxes* help you in assembling compact and solid gearings (Figure 9.3). Version a combines a worm gear with a 24t, thus performing a 1:24 reduction. We explained in Chapter 2 that worm gears can *transfer* motion but not *receive* motion—in other words, they can turn a gear but cannot be turned by it. We also explained that this feature is of great help when you want a mechanism to rest in its current position when you've switched the power off—for example, in an arm aimed to lift weights. This gearbox is ideal for such high-torque applications, because it encloses the 24t and the worm gear into a solid body case where the gears cannot fall apart.

Version b, the newest of the three, integrates well with standard beams and provides a convenient way to change the direction of motion or to split power onto two or three axles using 12t bevel gears.

Version c comes from older TECHNIC sets and corresponds to b in the way of functionalities. It's a bit harder to integrate with other parts, but has the advantage of allowing vertical mounts for the gears.

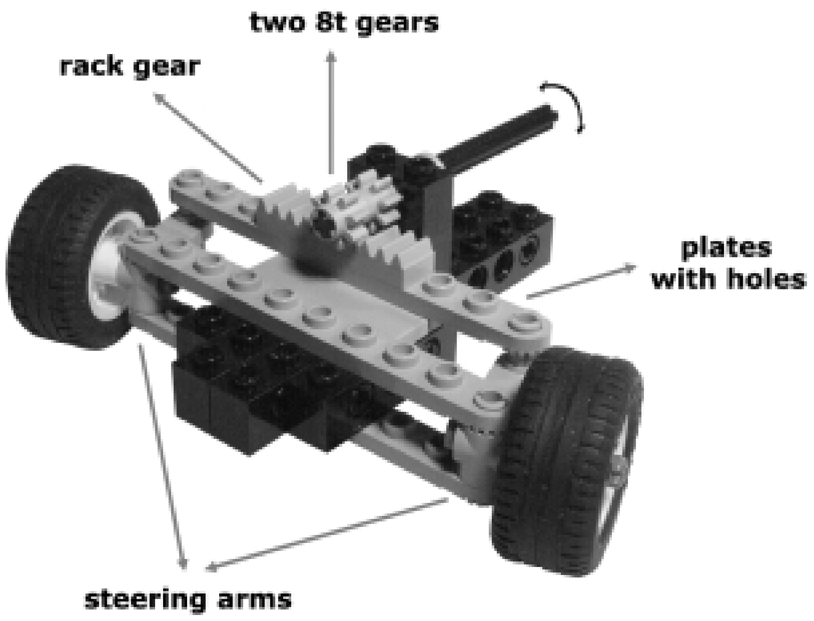
Figure 9.3 Gearboxes



Chain links, another component not included in the MINDSTORMS kit, come in two types. The first is meant for use in transmissions, as explained in Chapter 2 (*chain links*), while the second was designed to make up tracks of arbitrary length (*track links,* see Figure 2.19 in Chapter 2). They're a nice feature, although it's a pity the tracks' links don't get a better grip on most surfaces and that they come apart rather easily.

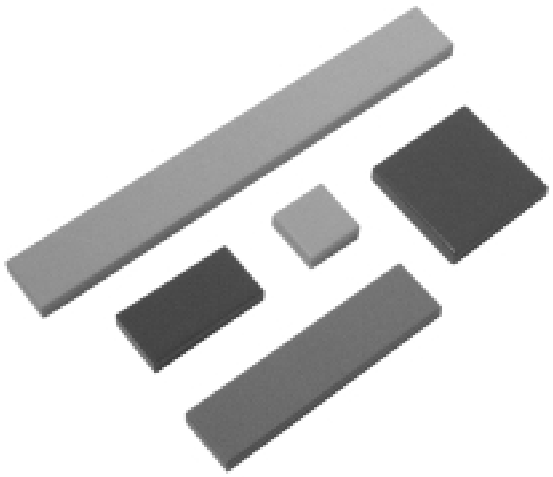
In Chapter 8, you were introduced to the turntable (see Figure 8.15 in Chapter 8), a very useful part for building rotating subassemblies, as well as some TECHNIC plates and connectors specially suited for building steering assemblies in association with the rack gear. Figure 9.4 shows how to combine three 1 x 10 TECHNIC plates, a rack gear, two 8t gears, two steering arms and some axles and connectors into a fully functional rack and pinion steering assembly. Two of the plates brace the beams of the chassis, enclosing the steering arms at their ends. The third plate connects the free ends of the steering arms and lets them pivot while remaining parallel with each other; a rack on the plate meshes with the pinion connected to the steering motor (not visible). Notice that the pinion is made up of two 8t gears; one wouldn't be enough, because while steering, the plate would move toward the stationary plates and it would lose contact with a single 8t gear pinion. Mount the wheels on the steering arms using *axles with a stud.*

Figure 9.4 TECHNIC Plates and Connectors for Building a Steering Assembly



The movable plate is supported by two 1 x 4 *tiles*, which are like plates with no studs and which provide the ideal smooth surface other parts can slide over (Figure 9.5). Racks and tiles are a very good combination for producing linear motion, and we confess we've never understood why LEGO didn't put any tiles in the MINDSTORMS kit.

Figure 9.5 Tiles Are Like Plates with No Studs



Some of the largest and most famous TECHNIC sets reproduce cars. They are a useful source for, among other things, *shock absorbers* (Figure 9.6), and large wheels (Figure 9.7). The use of shock absorbers is not limited to their traditional function, that is, keeping the wheels of a vehicle in touch with the ground on uneven or rough terrains; you can profitably employ them as springs in many kinds of mechanisms, including bumpers.

Figure 9.6 A Shock Absorber

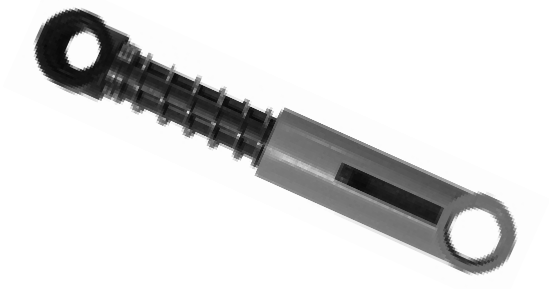
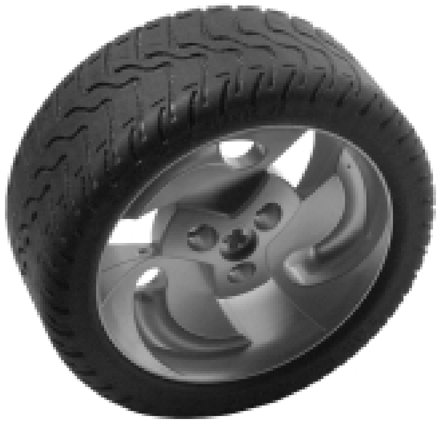
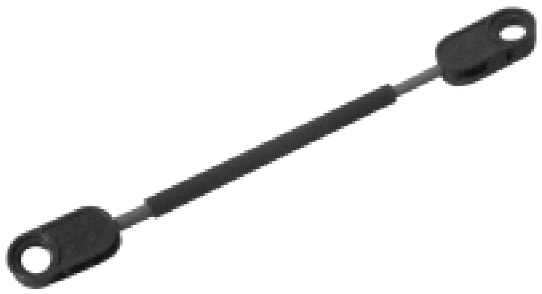


Figure 9.7 The Wheel from the 8448 Super Street Sensation



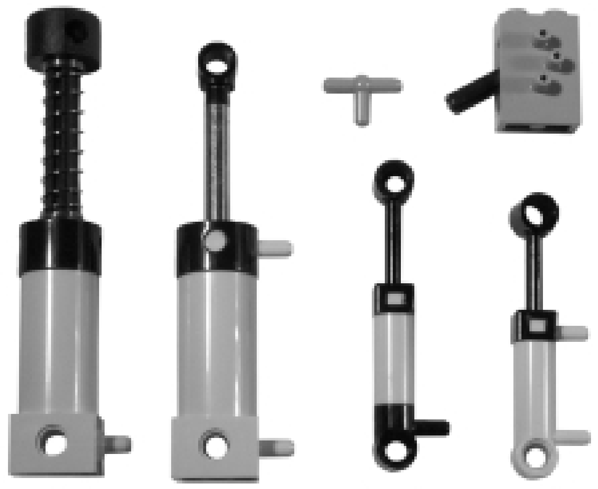
The *flex system* (Figure 9.8) allows the transference of linear motion from one point to another distant one, exactly like the wire ropes that control the accelerator and clutch in motorcycles, or the brakes on a bicycle. You probably won't need them very often, but they allow you to operate a part through a distant motor. They also prove extremely useful in making compact and lightweight mechanisms—for example, you can open/close a robotic hand at the end of a long arm that extends from a motor placed in the main body of the robot.

Figure 9.8 The Flex System



LEGO TECHNICS also features a line of *pneumatic devices*: small and large cylinders, small and large pumps, pipes, and valves (Figure 9.9). They offer so many possibilities when it comes to robotics that we decided to dedicate an entire chapter to them (Chapter 10).

Figure 9.9 Components of the Pneumatic System



Designing & Planning…

Choosing Colors

Most TECHNIC parts come in a large assortment of colors, which include the traditional LEGO colors (white, red, blue, yellow, green, black, and gray) and some more recent ones (tan, dark gray, light blue, light-green, lime, purple, orange, and brown). If you care about colors as much as we do, this is great news; a conscious use of colors can improve the appearance of robots. However, there's much more that colors can do for you: they can help in making the structure and the mechanisms of your robot more evident. In our favorite scheme, we use two colors for the body of the robot: one for plates and another for beams, liftarms, and all the other static parts. This makes the layered structure very easy to read. Then we use one or two additional colors for mobile parts to highlight their function in the robot. For example, the fact that you employ a beam as a connecting rod between two parts of a mechanism is more apparent if its color stands out against the prevalent colors of the robot.

In large and complex robots, you can use colors to identify its subsystems: one color for the mobile platform, another for the grabbing arm, a third for the rotating head, and so on for each relevant unit.

Continued

Colors help also in keeping the wiring neat when necessary. A dual RCX robot, for example, can use up to 12 input and output connections, and some of these wires are probably not so easy to trace inside the structure of the robot. Place pairs of small plates on the connectors at both ends of a wire, using a different color for each wire, and you'll have no problem understanding which port is connected to the motor and which is connected to the sensor.

Buying Additional Parts

Now that you've seen all these parts, you might wonder where you can get them. This is a very good question which, unfortunately, has no easy answer. There are general accessory sets, themed sets, expansion sets, and service packs, as well as general catalogues. Each may offer more or less than you need at one time, and price may also be a factor.

The MINDSTORMS line has many sets, but in our opinion some of them are priced a bit too high for their actual value. The 3801 Ultimate Accessory Set is a good choice, including a rotation sensor, a touch sensor, a light brick, a remote control, and other parts.

The 9732 Extreme Creatures Set contains few interesting parts for its price, but remember the Fiber Optic System unit, as explained in Chapter 4, can be used as a rotation sensor, too. The 9730 RoboSports Set is a bit more interesting, as it contains an extra motor. The most notable parts contained in the 9736 Exploration Mars Set are two gearboxes, six balloon tires, two very long cables (3m) and a bunch of beams, plates, gears, and connectors. In our opinion, these three sets are good purchases only if you find them at a reduced price.

The 9735 Robotics Discovery Set contains a unit called *Scout* that's a sort of younger brother of the RCX. Scout incorporates a light sensor, and features two output ports for motors and two input ports for sensors (passive types only: touch and temperature). It has a large display and offers some limited programmability from its console, without the need for a PC, thus offering an easy start to robotics. Despite this nice characteristic, we feel it's a bit too limited.

The two Star Wars MINDSTORMS sets, the 9748 Droid Developer Kit and the 9754 Dark Side Developer Kit contain an even more limited unit, MicroScout, that incorporates a motor and a light sensor, but has no ports. It has seven predefined programs, and can be interfaced to the Scout with an optical link to act as its third motor. Through the Scout you can also download a tiny program to the MicroScout. In our opinion, MicroScout is definitely too simple to be used for robotics, so we again suggest you buy these sets for their parts, and only if you find them for sale at a discounted price. If you really want another programmable brick, we strongly recommend a second MINDSTORMS kit, which with its RCX, two motors, three sensors, and more than 700 additional parts, in our opinion remains your best option.

LEGO also released a video camera system called 9731Vision Command. The camera connects to your PC, and a dedicated LEGO software can send IR commands to your RCX unit, through the tower, according to what happens inside the observed area. Don't dream of recognizing shapes or performing other sophisticated artificial vision tasks, since Vision Command allows only very basic reactions to changes in some predefined areas of the screen. You will discover also that the cable that links the camera to the PC is a constraint to your robot mobility. Despite these limitations, however, Vision Command opens up a world of possibilities.

MINDSTORMS expansion sets are an option, and TECHNIC sets another. Sad to say, but the current TECHNIC line does not include many expansion sets with suitable parts for robotics. Old TECHNIC sets had more beams and plates then current ones do, which, as we explained, tend to rely more and more on studless liftarms, which are useful but somewhat complicated to use. If you are so lucky as to find some discontinued TECHNIC sets, you have a good chance of it better suiting your needs. Being bound to the current production, large sets are a better purchase than small ones, having a higher ratio between functional and decorative parts. We prefer not to suggest any specific model here, as each fan has his or her own preferences; also, every year LEGO releases new sets and discontinues others.

With all that said, it is perfectly understandable that you may simply wish to buy only the specific parts you need. LEGO offers a mail service, called Shop-At-Home, from whose catalog you can order both sets and *elements packs* or *service packs*. Recently LEGO started an online service called LEGO Direct, through which you can order from your computer, pay with your credit cards, and get the parts or sets shipped to your door. LEGO Direct has been greeted with great enthusiasm by LEGO fans who see it as the promising beginning of a new era, one where everybody can order only the specific parts they need from a complete catalog. Currently, LEGO Direct offers the current line of sets and a limited choice of service packs, but the range is increasing and we all hope that it ends in a thorough and practical worldwide service.

Another useful resource is the DACTA service. DACTA is the branch of LEGO devoted to educational products, whose catalog includes a wide range of sets and supplementary kits. Though packed with a different assortment, the DACTA boxes contain the same parts used in commercial LEGO products. In all countries, the sale of the DACTA line is entrusted to companies specialized in selling educational items to institutions, though they normally sell to the public, too (for example, PITSCO in the USA and Spectrum Educational in Canada). Though not exactly cheap, the DACTA catalog includes many parts no longer available in sets or service packs, like the turntable or the track links, and many other parts that remain hard to find in large quantities, like the 40t gear and the rotation sensor.

Last but not least, LEGO fans from all over the world have formed a sort of community that has its own selling services. Some fan-run Web sites offer an impressive array of new and used parts and sets, in either mint or used condition, and most of the sellers accept credit cards and ship internationally. See Appendix A for some links to these commercial and private Internet LEGO shops.

Creating Custom Components

In the following sections, you will see that some of the proposed enhancements involve parts not supplied by the LEGO company. This applies in particular to electronics like motors and sensors.

We understand that your attitude toward non-LEGO parts could range from enthusiasm to hostility. You might see the benefit in making your own temperature sensor (spending only \$2 instead of the \$30 that the original costs), or you might be keen on the opportunity of giving your robot a voice recognition device. On the other hand, you might feel that using non-LEGO parts is a violation of the rules of the game, or you may be so fond of LEGO that you wish not to contaminate it with foreign components.

We can not, and will not, recommend one viewpoint over the other—the choice must be yours. We are personally open to some nonoriginal devices, provided that they "look like" LEGO parts. These can be cased into LEGO bricks, use standard LEGO wires and connectors, and quite closely resemble the originals. However, the use of aluminum plates, brass nuts, and bolts through LEGO beams, axles or plates cut to match a specific size, and visible chips and resistors are all unacceptable options to us. This is, again, our own choice, however.

Limiting your choices to LEGO parts has a certain appeal. It's like a common paradigm inside which you challenge yourself and other people to reach higher and higher goals. Most of the time, we build pure LEGO robots, using other devices only when we have something special in mind that we feel can really benefit from that particular hardware. Staying with original LEGO is particularly important when approaching contests and public challenges. It's a simple way to regulate what's admitted and what's not, and to be sure, too, that all competitors are pulling from identical resources.

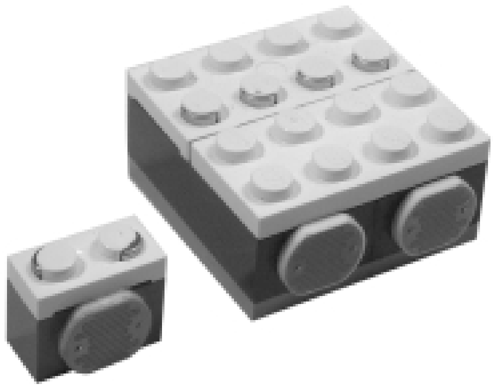
On the other side, if you're open to experimenting with non-LEGO devices, your horizons become much broader. In this section, we'll provide some examples of what can be done with them, our assumption being that you continue to use LEGO parts to build your robots, and the RCX to run them; thus, we'll discuss the use of non-LEGO sensors and motors only.

The LEGO company doesn't release much information about the internals of its electronic devices, so most of the technical details currently available to the public are based on the work of the pioneer hackers who analyzed and dissected the sensors and motors. Michael Gasperi is the person who made the strongest single contribution to this process, his Web site and book being reference points for any work in the field. Some of these custom devices are really easy to make if you can solder, or have a friend who can. In this chapter, we will show you some of *what* can be done; refer to Appendix A to find resources that teach you *how* to make this stuff, or tell you where to buy it.

Building Custom Sensors

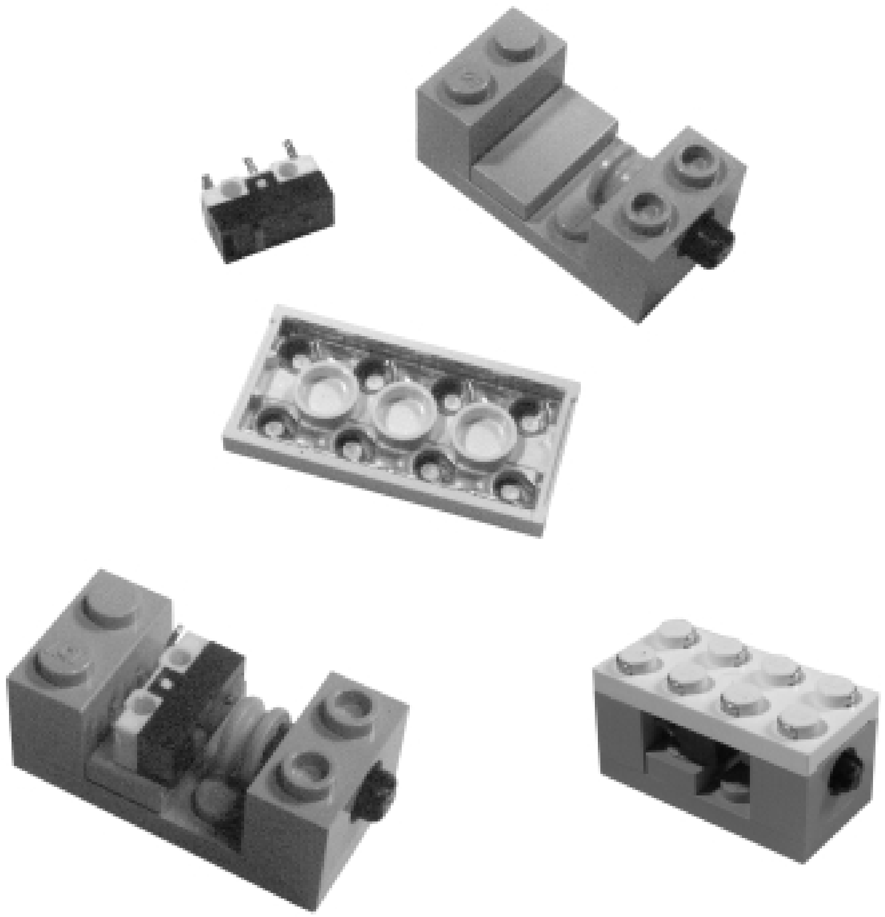
Michael Gasperi 's site explains how to build some simple custom sensors. The simplest of all is probably the passive light sensor built with a cadmium sulfide (CdS) photo-resistor and nothing more (Figure 9.10). This sensor is much better than the original LEGO light sensor in measuring ambient light, though it's a bit slow in acknowledging variations. With two CdS cells and some electronics, you can build a differential light sensor, which tells you in a single value if there's any difference in the amount of light received by the two units; this is very useful in pinpointing light sources.

Figure 9.10 Single and Double CdS Light Sensors



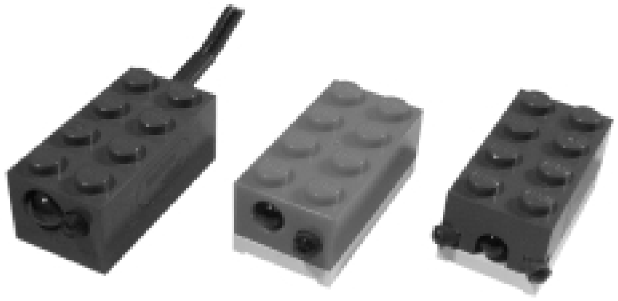
Recycling junk is an option when trying to save money. Figure 9.11 shows a touch sensor made with a switch from a computer mouse. Pulling apart a broken mouse, you will discover that there are some micro-switches connected to its push-buttons. Unsolder them from their circuit plate, solder their terminal to an electric plate, then add some parts to the case in the switch.

Figure 9.11 A Mouse Switch Recycled into a Touch Sensor



There are many people who describe in their Web sites how to make custom sensors, providing schematics and detailed instructions. Some of them also sell construction kits or finished sensors. Pete Sevcik is a good example of this latter category; his sensors are very well engineered and professionally cased into LEGO bricks. Figure 9.12 shows three of his *infrared proximity detectors* (IRPD). An IRPD is a sensor based on the IR light proximity measurement system we explained in Chapter 4, with the advantage being that you are not required to do anything in your code, just read the sensor value. IRPD sensors have an incredible range of applications. They are perfect for obstacle detection, of course, but you can use them also to make your robot follow your hand movements, to trigger the grabbing feature of a robotic hand, to find soda cans or locate your opponent during competitions. As we explained in Chapter 4, the proximity detection technique cannot measure distances, but it can tell you if an object is coming closer or entering its field of detection. The rightmost sensor in Figure 9.12 is a *dual IRPD*, able to detect an obstacle within a wider angle and tell you if it's front, left, or right with a single reading.

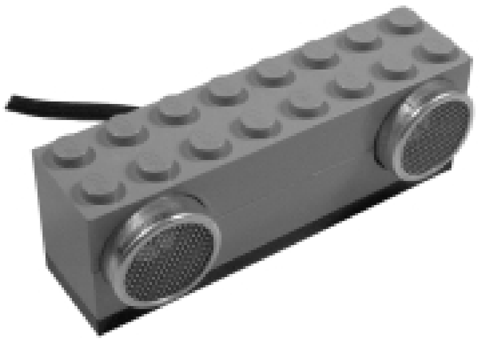
Figure 9.12 Different Kinds of Infrared Proximity Sensors



Sevick also produces a *pitch sensor*, a sophisticated sound sensor that returns a value proportional to the frequency of the incoming sound. You can thus control your robot by simply whistling or playing a flute or recorder like a modern Pied Piper. The robotic pianist of Chapter 21 represents a possible application for this sensor: It can learn a simple melody just by listening to it.

John Barnes is another person who has shown incredible creativity and competence in building custom sensors. Barnes made one of the first LEGO compatible ultrasonic sensors (Figure 9.13), a device able to measure distances evaluating the delay between the emission of a sound and its returning echo. Like a sonar, the sensor emits an ultrasonic signal (not audible), reads its echo, and returns a value that represents the distance of the closest object. The fields of application of these sensors are similar to what's described for the IRPD sensors, with the further advantage that ultrasonic sensors return an absolute distance value. This means that your robot can improve its navigation abilities, because it can not only avoid obstacles but also find the best route evaluating the distances of the surrounding objects.

Figure 9.13 An Ultrasonic Distance Sensor



Barnes has assembled many other amazing devices, including a *compass* with a resolution of 3.75° (Figure 9.14) and a *pyroelectric sensor* able to detect the presence of humans or animals by measuring the changes in ambient IR radiation (Figure 9.15). The compass sensor just looks like a pile of bricks, because there isn't any device emerging from its body, but the inside contains a small electronic compass and a circuit to convert its output into values that the RCX can interpret. Connect the compass sensor to an input port of the RCX configured for a light sensor, and it will return values in the range of 0 to 95, where 0 is North, 24 is East, 48 is South, and 72 is West. Having the RCX know which way it's pointing in order to keep going straight and having it make known angle turns makes a big difference in solving navigation problems!

Figure 9.14 A Compass Sensor

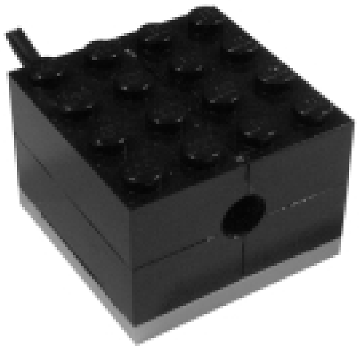


Figure 9.15 A Pyroelectric Sensor

The casing around the pyroelectric sensor has a small hole that lets its internal "eye" receive the infrared light any warm body produces. It requires some time to adapt to the ambient radiation, but afterward it can detect any change in intensity. These features make it unsuitable for mobile robots, but it's very useful in those projects where a robot must start doing something when it detects a human presence.

Probably the most astonishing of Barnes' devices is his Voice Recognition unit (Figure 9.16). After a short teaching session, you will be able to give simple one- or two-word commands to your robot like "go," "stop," "left," "take" and see your robot perform the required task. It's rather large and heavy, because it contains its own set of batteries, and, consequently, is not very easy to place in a compact robot. However, it gives reality to the dreams of robots harbored by every sci-fi fan: the ability to respond to vocal commands!

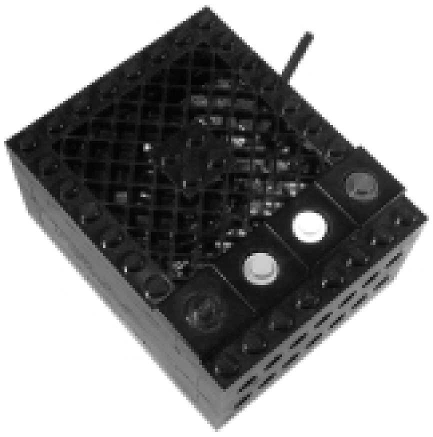
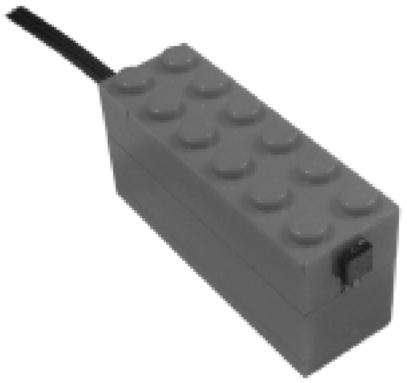


Figure 9.16 John Barnes' Voice Recognition Unit

No-contact switches are very useful tools, too. These are switches that close without the need of physical contacts with the casing of the sensor. We integrated Michael Gasperi's General Purpose Analog Interface with a *Hall-effect detector* to build a sensor capable of detecting magnetic fields (Figure 9.17). A Hall-effect detector is a small integrated circuit which returns different signals depending on whether it is in the presence of a strong magnetic field or not. Gluing a small permanent magnet on a LEGO peg, you can easily mount it on any mobile part of the robot. When the magnet comes very close to the sensor, the latter detects it.

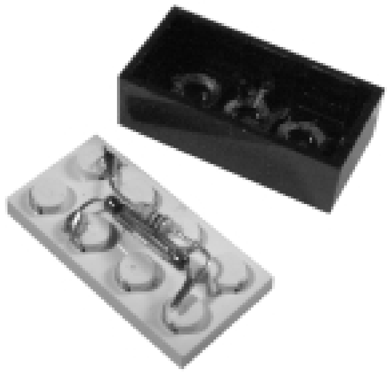
Figure 9.17 A Hall-Effect Sensor



Chris Phillips followed a simpler and more effective approach to get the same result using a cheap and easy-to-mount *Reed switch*. A Reed switch is a small bulb containing two thin metal plates very close to each other. When you put the bulb close to the source of a strong magnetic field, the metal plates touch and complete the circuit. Small permanent magnets are the ideal parts to trigger this sensor, with the same procedure we described for the Hall-effect sensor. You can also use the LEGO magnets designed to couple train cars. Detecting trains is actually what Phillips devised his sensor for, but it is suitable for many other applications: It can replace touch sensors in almost all applications, and even emulate rotation sensors if you mount the permanent magnet on a wheel that makes it pass periodically in front of the sensor.

Figure 9.18 shows a Reed bulb mounted in series with a 100K resistor over a LEGO *electric plate*, which provides an easy way to interface custom sensors to the standard LEGO 9v wiring system. The final sensor will be cased in a hollowed brick to make it look like a standard LEGO component.

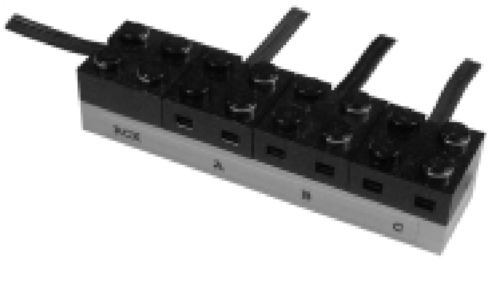
Figure 9.18 A Reed Switch Sensor before Final Assembly



Solving Port Limitations

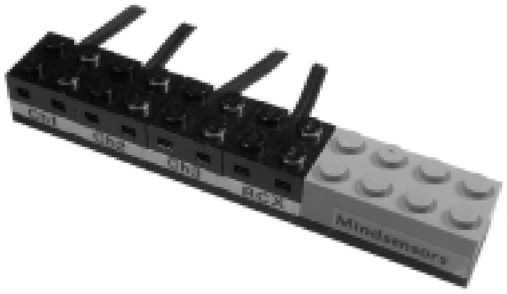
Some of the electronic devices that have appeared in the LEGO robotics community are meant to solve the endless dilemma of the limited input and output port number. The common approach involves *multiplexing*, a technique through which signals from different sources are combined into a single signal. Michael Gasperi explains how to build a very simple multiplexer that can host up to three touch sensors and return a value that the RCX decodes into their respective states (Figure 9.19). This device takes advantage of the fact that the RCX can read raw values instead of simple on/off states, and returns a unique number for any possible combination of three sensors.

Figure 9.19 A Three Touch Sensor Multiplexer



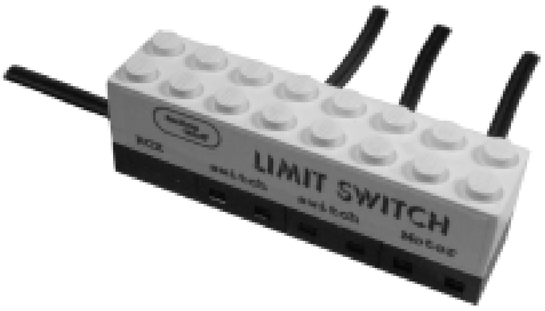
Nitin Patil designed a more complex multiplexer suitable for connecting a single input port to three active sensors, like the original light and rotation sensors, or any other custom active sensor like IRPDs, sound, and so on. Active sensors use the entire raw value range, thus this device cannot combine their signals into a single number like the three touch sensor multiplexer does. Actually Patil's device connects a single sensor at a time to the port, and requires the RCX to send a short impulse to select the desired sensor (Figure 9.20).

Figure 9.20 A Three Active Sensor Multiplexer



Pete Sevcik's Limit Switch, though not a multiplexer, allows you to save some ports by combining two touch sensors and a motor on a single output port (Figure 9.21). Until a switch closes, the motor is under normal control from the RCX. When a touch sensor gets pressed, the inner circuit prevents the motor from turning into a specific direction, thus automatically limiting the motion of a mechanical device. If your robot has a rotating head, this limit switch can make it stop at its left and right bounds using just a single port.

Figure 9.21 Pete Sevcik's Limit Switch



Output port multiplexing, though technically possible, doesn't get the same attention as input port multiplexing, thus there are few schematics and little documentation on this topic. The focus seems most on using different kinds of motors, *servo motors* in particular. Servos are typically used in radio-controlled models to steer vehicles, move ailerons, and handle other movable components. They are cheap and have high torque, thus they are ideal for some applications. Unfortunately, they expect power in a specific waveform that the RCX cannot supply. Ralph Hempel solved the puzzle creating a simple electronic interface that performs the appropriate conversion, thus revealing the power of servo motors to LEGO robotics hobbyists.

NOTE

The number of electronic expansion devices is vast, and still growing. If you are curious about these devices, we once again invite you to visit some of the Web links we provide in Appendix A.

Creative Solutions When More RCX Ports Are Needed

When you start gaining experience with LEGO robotics, and wish to build something more complex than your early robots, you will quickly find yourself facing the heavy constraints imposed by the limited number of ports the RCX has. Are three motors and three sensors too few for you? If you feel a bit frustrated, remember that you're in good company. Thousands of other MINDSTORMS fans feel the same!

In Chapter 4, we provided some tips on connecting more sensors to a single input port. We are going to describe here some tricks that, using only LEGO components, allow you to somewhat expand your motor outputs.

Start by observing that in some applications you don't need a motor turning in both directions, just one motor in one direction. Your robot can take advantage of this fact by driving two different gearings with a single motor. Figure 9.22 shows how you can achieve this using a differential gear: Its output axles mount two 24t gears that can rotate each one only in a single direction. The two 1 x 4 beams work like *ratchets*. They let the gear turn in one direction but block its teeth in the other. If you connect the motor to the body of the differential, it will drive either the right or the left axle depending on its direction.

Another setup, shown in Figure 9.23, is based on the fact that the worm gear is free to slide along the axle.

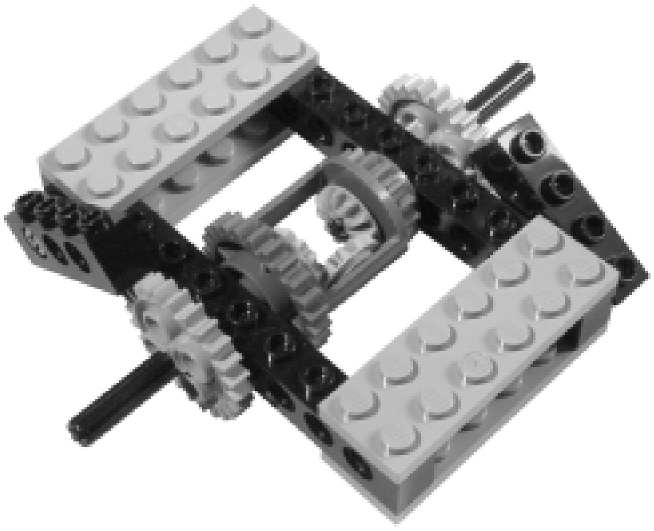


Figure 9.22 Splitting a Rotary Motion on Two Axles

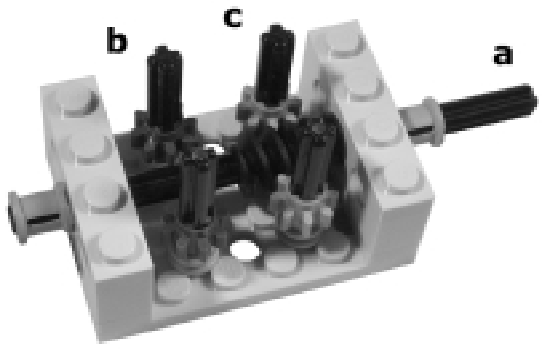
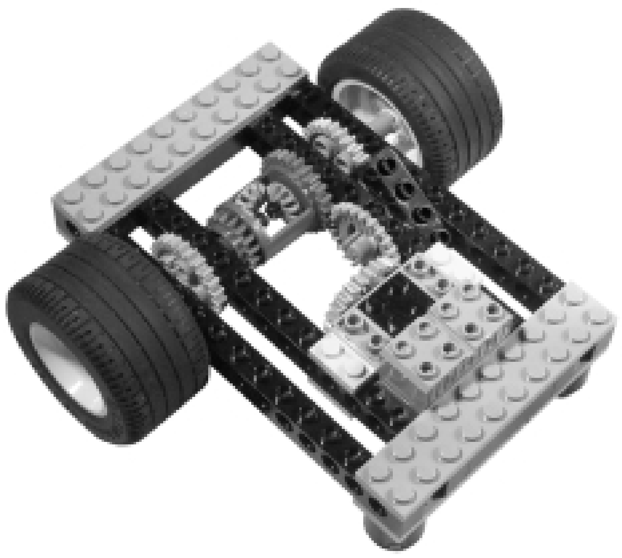


Figure 9.23 The Crawling Worm Gear

Provided that there is some friction in the output axles B and C, when axle A turns clockwise, the worm crawls left until it engages the B 8t gears and gets stopped by the beam. Turning A counterclockwise, the worm crawls right, disengaging the B gears and engaging the C pair. Thus, with a single input axle you get two pairs of outputs, each pair having one axle turning clockwise and the other counterclockwise. We invite you once again to build and test this simple assembly. It's almost unbelievable to see!

To put theory into practice, let's see how you apply these principles to the mobility configurations of Chapter 8. The differential drive is a good starting point. Can you drive two wheels with a single motor? Yes, you can—using the differential gear to split its power onto two separate outputs. Then, copying the design of Figure 9.22, add a ratchet beam that acts on one of the wheels (Figure 9.24). The motor drives both wheels through the differential when going forward, but one of them gets blocked during reverse motion, making the robot pivot around it. Simple, but limited. It's not guaranteed to go straight, and cannot spin in place. Nevertheless, it allows you to make a mobile platform that uses only one port of your RCX!

Figure 9.24 A Single Motor Differential Drive

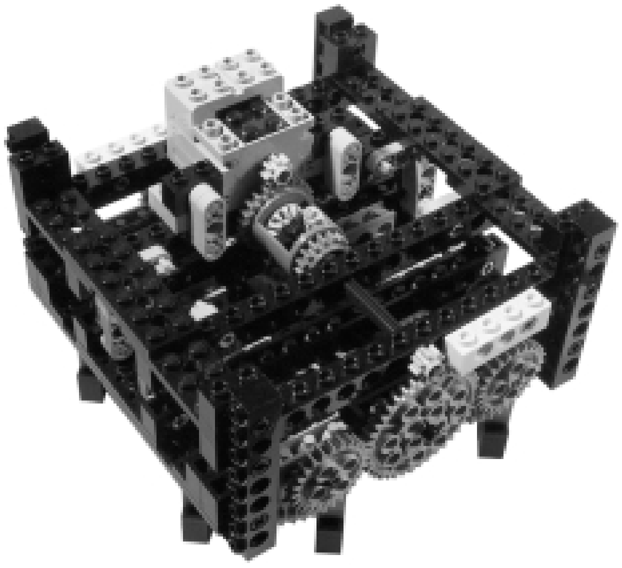


The dual differential drive shown in Chapter 8 is a good starting point for a more sophisticated solution. Its design uses one motor to drive straight and the other to change direction. You should replace these motors with a mechanism similar to that of Figure 9.22, making the output axles of its differential gear (the third of the robot!) take the place of the motor shafts. Now apply a motor to the last differential gear: In one direction it will make the robot go forward, in the other it will make the robot spin in place. It works, though we realize that the resulting gearing probably isn't the simplest thing we've ever seen!

Even in the synchro drive (Figure 9.25) you can get full motion control with a single motor. Relying on the fact that the synchro drive has the freedom to turn on its wheels at any angle, you can keep them turned in the same direction until they reach the desired position. Again, apply the scheme of Figure 9.22 and make one output axle of the differential gear operate the steering mechanism, while the other provides drive motion.

When the motor turns one way, the wheels change their orientation, and when the motor turns in the other direction, the wheels move the robot forward. Backward motion is not required, because the wheels can point to any heading and the motion reversal is performed by a 180° change in their direction. With a platform like this, you have complete control over navigation, and you still have two free output ports to drive other devices.

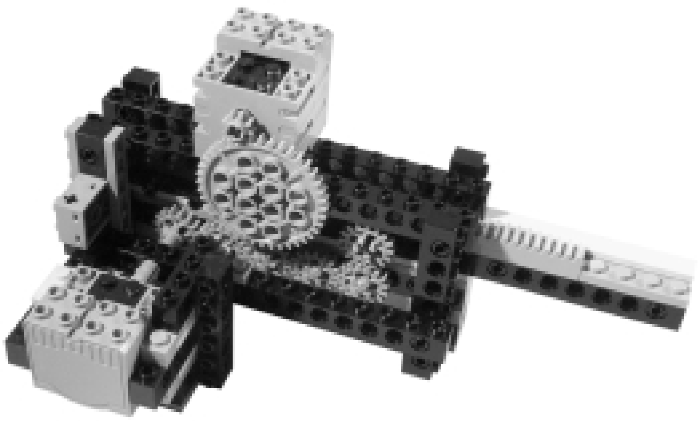
Figure 9.25 A Single Motor Synchro Drive

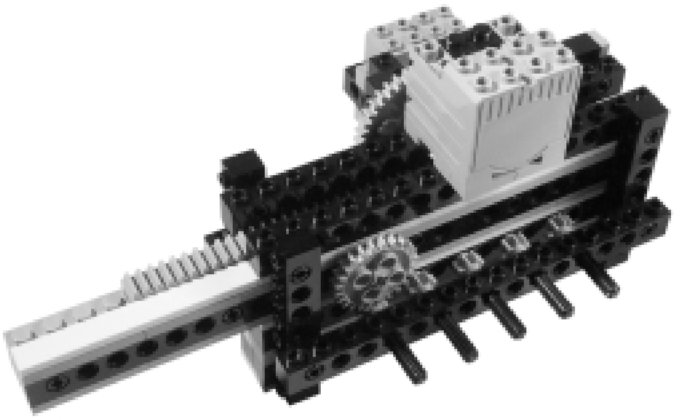


Single motor tricycle drives are possible, too, requiring a gearing very similar to that of our single motor synchro drive. Make just one of those steering-driven wheels, add two rear free wheels, and you're done.

This trick of splitting the turning directions over two separate axles obviously won't cover all your needs for extra ports. In many cases, you must control both directions of your gearings, but you probably don't need all motors running at the same time. In a robotic arm with three independent movements, for example, you use three motors, but using just one at a time doesn't affect its global functionality. The idea is to use one motor to make a second motor switch among several possible outputs. This approach will always require two motors, and engages two output ports, but can give you a virtually unlimited number of independent bi-directional outputs, only one of them running at any time. A possible implementation of such a device is shown in Figure 9.26. The motor at the bottom drives five 16t gears all linked together. On the other side of the assembly there are five 8t output gears not connected to the previous 16t. A second motor at the top slides a switching rack that, through a 16t on one side and a 24t on the other, connects the input gears to one of the five possible outputs. We used a touch sensor to control the position of the switching rack: five black pegs close the switch in turn when the gears are in one of the five matching positions. Due to its large size, this setup is probably more suitable for static robots than for mobile ones.

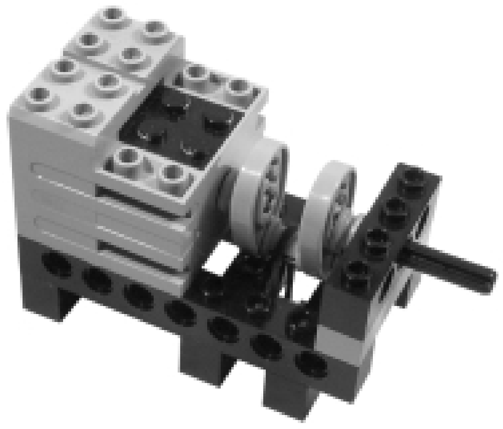
Figure 9.26 Switching a Motor among Five Output Axles





The previous example requires two output ports and one input port. In this case, as well as some others, we can save an input port by implementing a sort of *stepper motor.* A stepper motor is a motor that, under a given impulse, turns precisely at a known angle, just a single step of a turn. Stepper motors are widespread devices. You can find them in any computer printer or plotter, and in digital machine tools. LEGO doesn't make a stepper motor, nor does the RCX have dedicated instructions for them, but Robert Munafo found a pure-LEGO solution. Our version is a slight variation of Robert's original setup (see Figure 9.27). A rubber band keeps the output axle down in its default position. You have to power the motor for a short time, enough to make the axle get past the resistance of the rubber band and make a bit more than half a turn. Now put the motor in float mode, wait another short interval, and let the rubber band complete the turn of the axle. For any impulse made of a run time and a float time, the output shaft makes exactly one turn.

Figure 9.27 A Stepper Motor



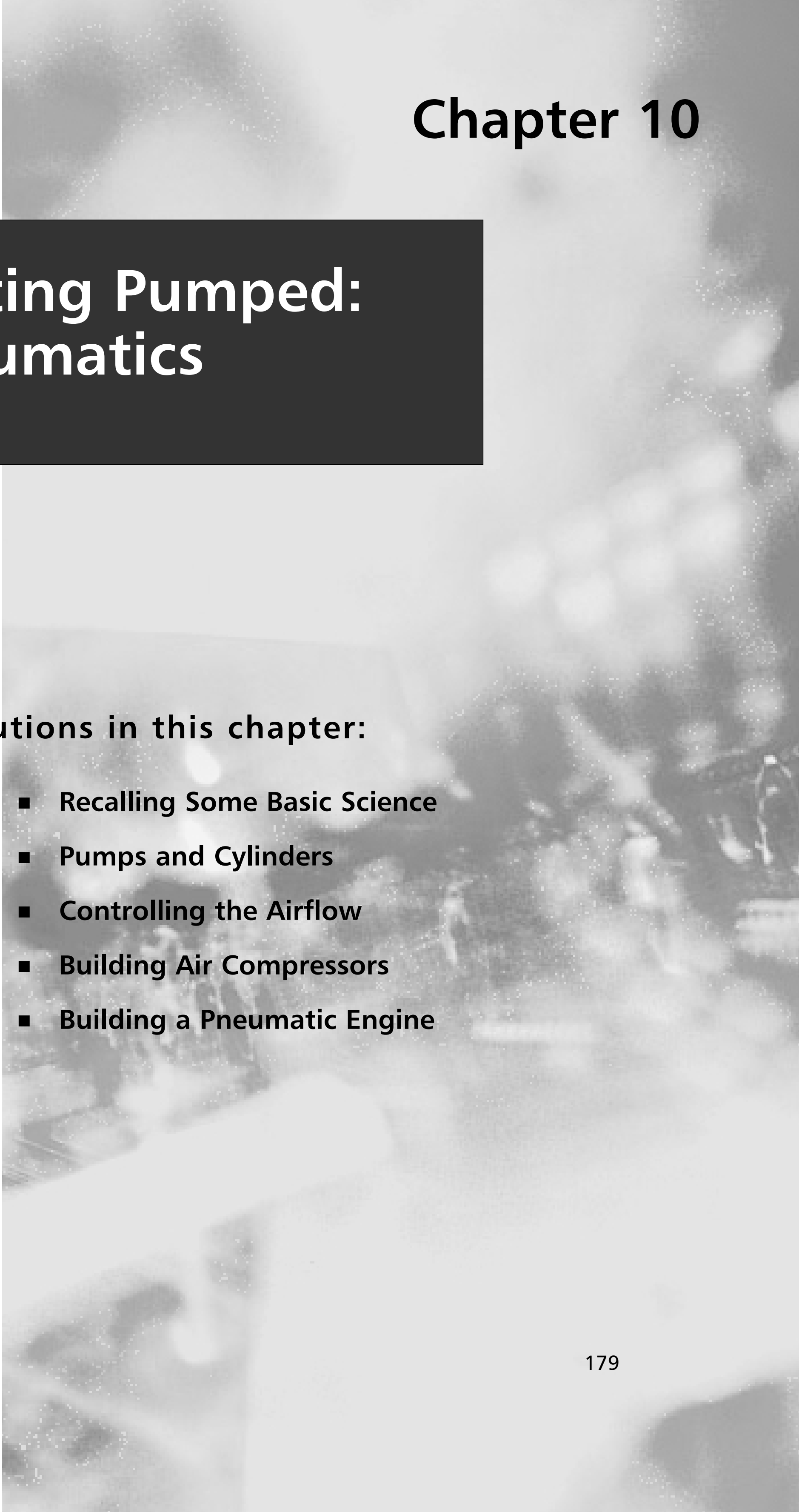
The beauty of the system is that timing is not at all critical. The on time can be any interval that makes the axle rotate more than half a turn but less than one and a half, while the float time can be any interval equal to or greater than the time needed for the rubber band to return to its default position.

Summary

In this chapter, we have been discussing extra parts, expansion sets, custom sensors, and tricks for using the same motor for more than one task:

- Extra parts come from either sets or service packs. Unfortunately, it's not always easy to buy just the parts you need, because sometimes they don't come in a service pack and you have to buy a set that contains many other elements you don't need. The LEGO Direct Internet shop is growing quickly, and it promises to become a very thorough and practical service. DACTA supplier and fan-run online shops fill the gap in the offer of spare parts, giving you countless opportunities to improve your equipment set.
- Custom sensors are a new frontier, and reveal a whole new world of possibilities. Would you like your robot to measure the distances of the objects around it? It's possible. Would you like it to recognize vocal commands? Again, it can be done. Proximity detectors, sound sensors, magnetic switches, electronic compasses, input multiplexers… the Internet is crowded with Web sites that teach you how to build your own MINDSTORMS-compatible custom sensors, or that sell them ready to use.
- Mechanical tricks enable you to use the same motor to power multiple mechanisms. Through the use of a differential gear and a couple of ratchet beams, you can split the output of a motor between two output axles. This principle extends to the point of driving a complete platform with a single motor.

There's a common denominator for these apparently unconnected topics—we want to push the limits farther. What this means (and can mean) depends on you, on what your rules are in regards to using non-LEGO parts, on how much you can spend on expansion sets, and how imaginative you are in finding new solutions to problems. Don't give up without a fight! Reverse the problem, or start again from scratch, or let the problem rest for a while before you attack it again. Look around you for inspiration, and talk to friends. Most of the greatest MINDSTORMS robots ever seen came from ideas that seemed impossible at first glance.



Chapter 10

Getting Pumped: Pneumatics

Solutions in this chapter:

- Recalling Some Basic Science
- Pumps and Cylinders
- Controlling the Airflow
- Building Air Compressors
- Building a Pneumatic Engine

Introduction

In Chapter 9, we mentioned that *pneumatics* might be a nice addition to your robotic equipment. Now, we'll explore the topic in more detail. Pneumatics is the discipline that describes gas flows and how to use its properties to transmit energy or convert the same into force and motion. Most pneumatic applications use that gaseous mixture most widely available—air—and the LEGO world is no exception.

Pneumatics is a great tool for robotics, and is especially useful when your mechanisms need linear motion or an elastic behavior. It's also very useful as a way to store energy for subsequent uses. We will briefly cover the basic concepts of pneumatics, then put those theories into practice, explaining how LEGO pneumatic components work and what you can do with them, along the way showing you how to stop and start airflow in order to produce motion in your robot. By the end of the chapter, you should hopefully be up to speed on many pneumatic components, including: valves, pumps, cylinders, compressors, and pneumatic engines.

Recalling Some Basic Science

To understand pneumatics, you have to recall the properties of gases. The most important property is that they have neither specific shape nor volume, because they expand and fill all available space within a container. This means that the quantity of gas inside a tank does not solely depend on the tank's volume. The greater the quantity of gas in a given volume, the higher its *pressure*.

NOTE

The science that describes the properties of gases is called *thermodynamics*. Its *Ideal Gas Law* relates four quantities: volume, pressure, temperature, and mass (expressed in moles). In our simplified discussion, we will deliberately ignore temperature, since, in our situation, it shall essentially remain constant throughout.

We all have the opportunities to experiment with pneumatics using everyday objects. The tires of a bicycle are a good example: Their inner volume is constant, but you can increase their pressure by pumping air in. The more air inside, the greater the pressure, and the more it opposes external forces—in other words, the tires become harder.

This example leads to a second important property of compressed gases: Their pushing outward on the walls of their containers illustrates their *elasticity*. Elasticity is the property of an object that allows it to return to its original shape after deformation. The greater the elasticity, the more precisely it returns to its original configuration. In the example of the bicycle tire, if you push your finger against it, you can temporarily create a dimple in the surface, but as soon as you remove the finger, the tire resumes its shape—the greater the pressure inside, the higher the resistance to deformation.

The fact that gases are so easy to compress is what makes pneumatics different from *hydraulics* (the science of liquid flow). Essentially, liquids are uncompressible.

When you compress a gas into a tank, increasing its pressure, you are storing energy. Pressure can be interpreted also as a *density of energy*, that is, the quantity of energy per unit volume. This leads to a very interesting application of pneumatics: You can use tanks to accumulate energy, which can then be later released when needed. You pump gas in to increase the pressure in the tank, storing energy, and draw gas out to use that energy, converting it into motion.

A flow of air or gas in general is produced by a difference in pressure: The air flows from the container with the higher pressure into the one with the lower pressure, until the two equalize. (In this context, we've given the term *container* the widest possible meaning. It can be a tank, a pipe, or the inner chambers of a pump or cylinder.)

Pumps and Cylinders

LEGO introduced the first pneumatic devices in the TECHNIC line during the mid-eighties, then a few years later modified the system to make it more complete and efficient. After a long tradition of impressive pneumatic TECHNIC sets, including crane trucks, excavators, and bulldozers, they discontinued the production of air-powered models. Fortunately, LEGO pneumatic devices have been recently reissued in a specific service pack (#5218) available through Shop-At-Home or at the LEGO Internet shop.

The basic components of the LEGO pneumatic systems are *pumps* and *cylinders* (see Figure 10.1). The function of a pump is to convert mechanical work into air pressure. They come in two kinds, the large variety, designed to be used by hand, and its smaller cousin, suitable for operation with a motor. Cylinders, on the other hand, convert air pressure back into mechanical work, and come in two different sizes as well.

Figure 10.1 Pumps and Cylinders

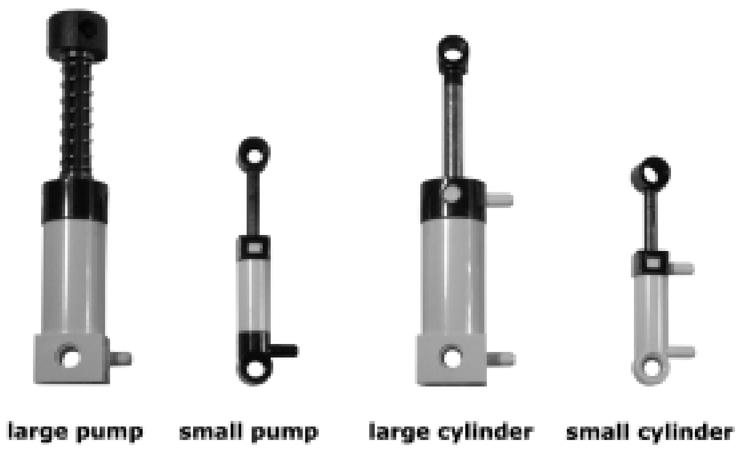


Figure 10.2 shows a cutaway of the large pump in action. When you press its piston down, you reduce the volume of the interior section, thus increasing the pressure and forcing air to exit the output port until the inner pressure equals that outside. When you release the piston, the spring pushes the piston up again; a valve closes the output port so as not to let the compressed air come back inside the pump, while another valve lets new air come in around the piston-rod. The small pump follows the same working scheme exactly, with the difference being that it doesn't contain a spring and its piston needs to be pulled after having been pushed. It's designed to be operated through an electric motor.

Cylinders are slightly different from pumps. Their top is airtight and doesn't let air escape from around the piston-rod. The piston divides the cylinder into both a lower and upper chamber, each one provided with a port. The basic property of a pneumatic cylinder is that its piston tends to move according to the difference in pressure between the chambers, expanding the volume of the one with higher pressure and reducing the other until the two pressures equalize, or until the piston comes to the end of its stroke. When you connect the lower port to a pump using a tube, and supply compressed air into the lower chamber, its pressure pushes the piston up. Doing this, the volume of the chamber increases, and this lowers the pressure until it's equal to that of the upper chamber. During the operation, the port of the upper chamber has been left open, so its air can freely escape, reaching equilibrium with the outside air pressure. Similarly, when you connect the upper port to the pump, and supply compressed air, the piston moves down (Figure 10.3).

Figure 10.2 Cutaway of the Large Pump in Action

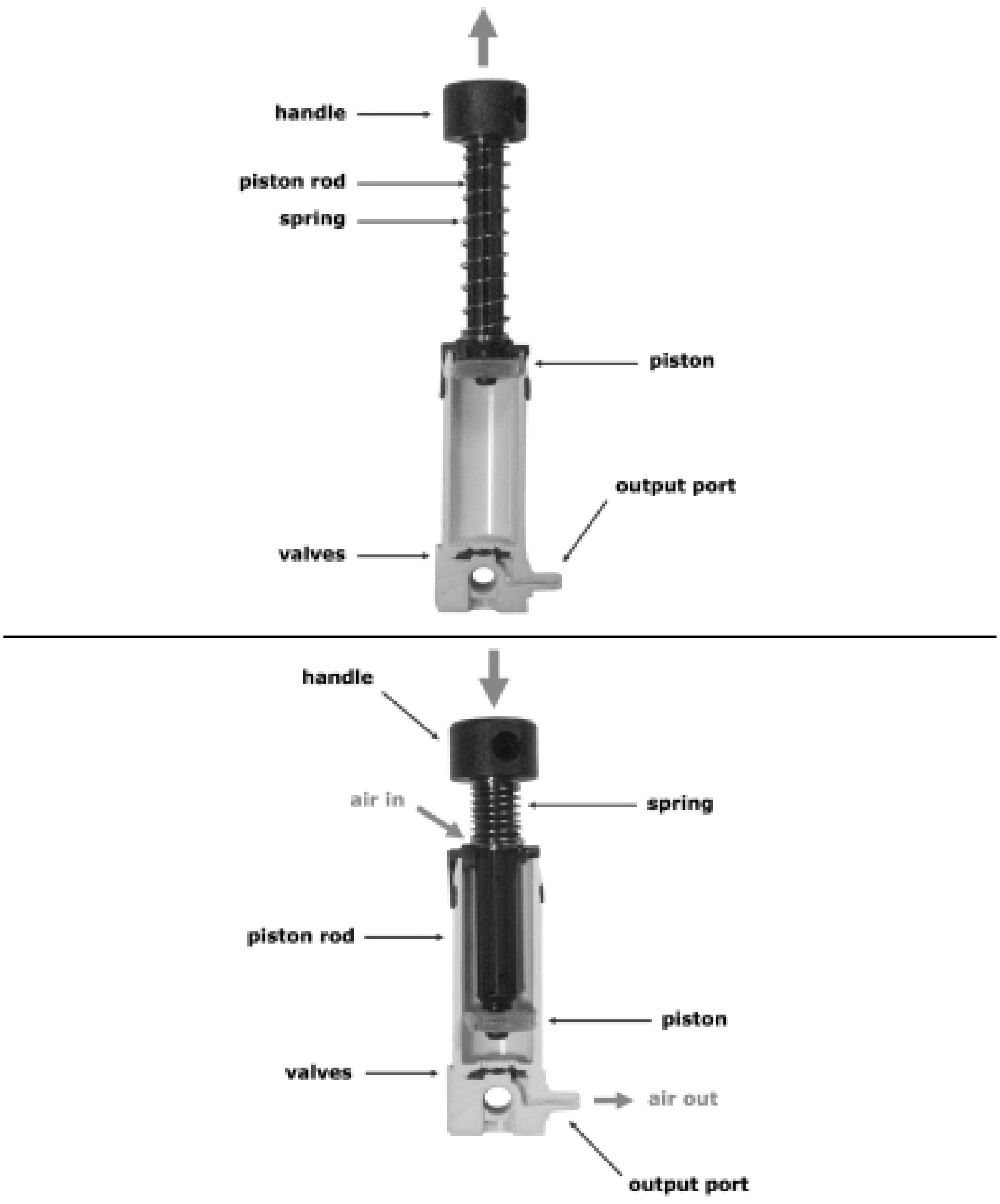
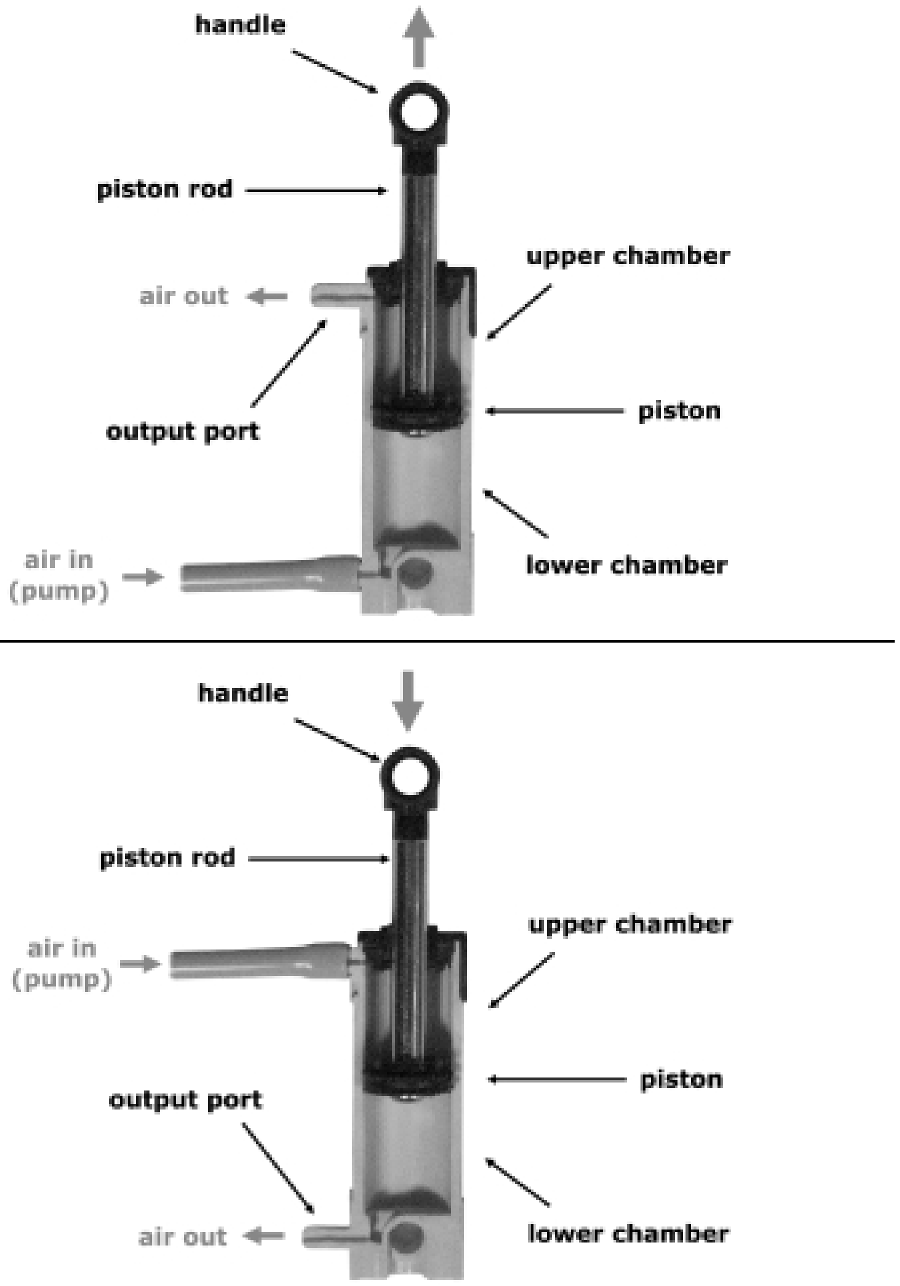
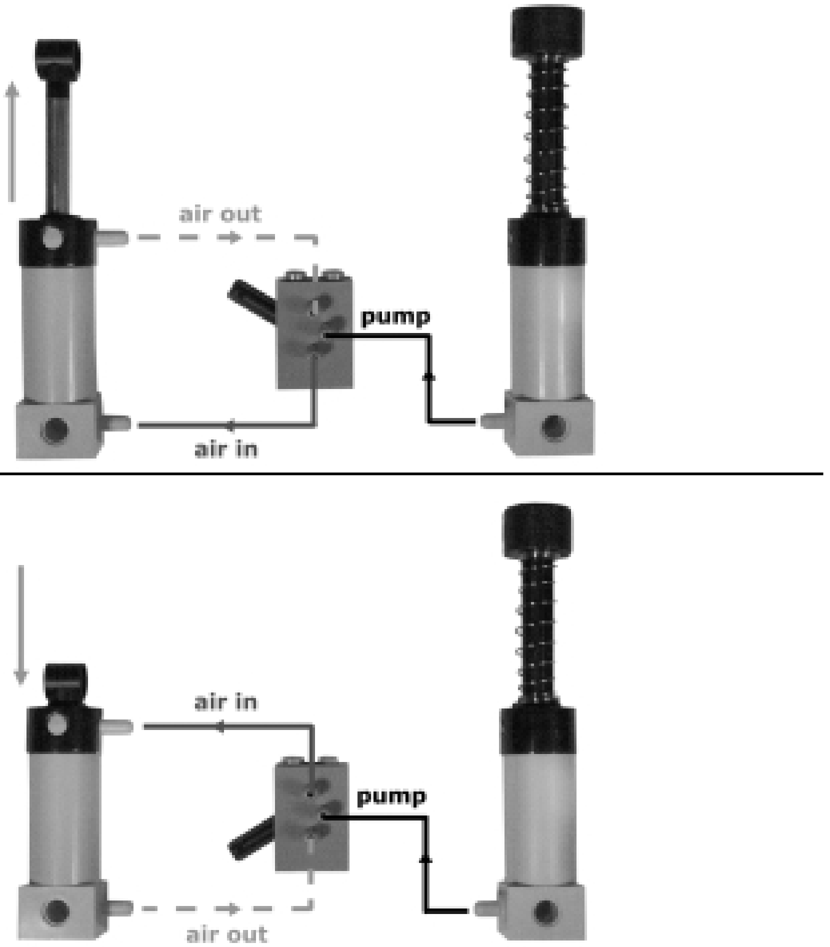


Figure 10.3 Cutaway of the Large Cylinder in Action



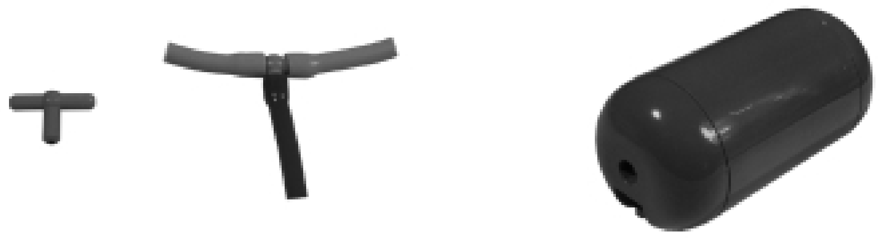
Surely you don't want to move the tube from one port or the other to operate the cylinder. It may work, but it's not very practical. The LEGO *valve* has been designed precisely for this task: It can direct the airflow coming from a pump to any one of the two ports of a cylinder, while at the same time let the pressure from the other chamber of the cylinder discharge into the atmosphere (see Figure 10.4). The valve also has a central (neutral) position, which traps the air in the system so the cylinder can neither move up nor down.

Figure 10.4 The Basic Pneumatic Connection



The LEGO tubing system is completed by a *T-junction* and a *tank* (see Figure 10.5). T-junctions allow you to branch tubes, typically to bring air from the source to more than a single valve. The tank is very useful for storing a small quantity of compressed air to be used later. We explained that increasing pressure is like storing energy, thus the air tank can be effectively considered an accumulator: charge it with compressed air and release it through the valve when necessary to convert that energy into mechanical work.

Figure 10.5 A T-Junction and a Tank



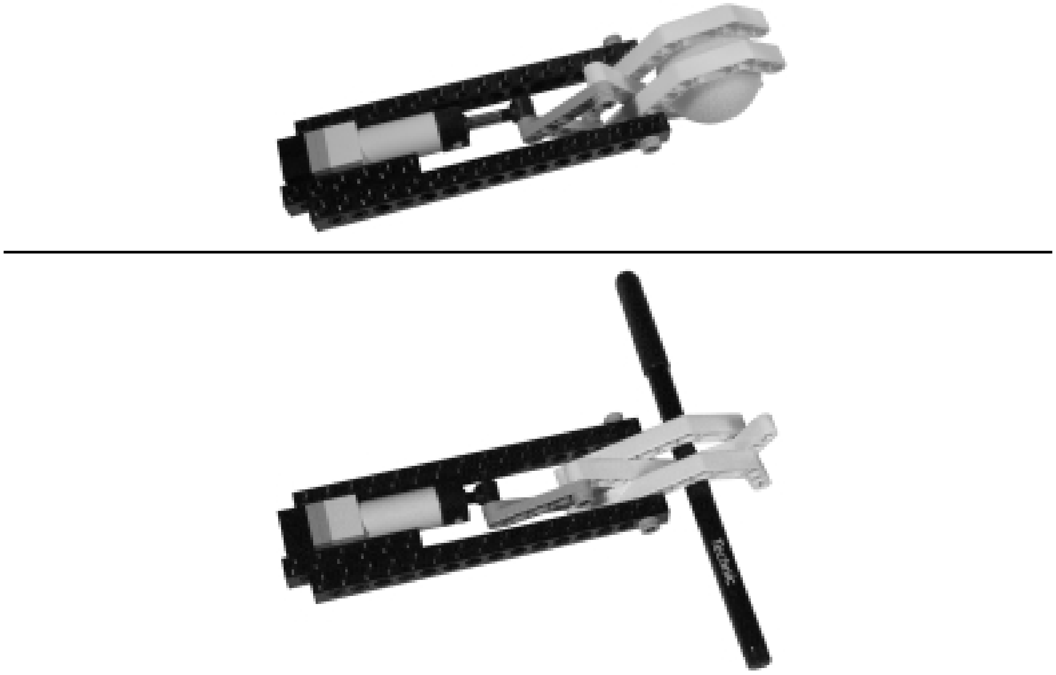
Pneumatic cylinders provide high-power linear motion, and thus are the ideal choice for a broad range of applications: articulated arms or legs, hands, pliers, cranes, and much more. In describing the basic concepts about pneumatics, we told you that compressed gases tend to make their containers react elastically to external forces. You can test this property with LEGO cylinders, too: connect a cylinder to a pump and operate the pump until the piston of the cylinder extends in full. Now, press the rod of the cylinder. You can push it down, but as soon as you stop applying force, the rod comes back up again. This property is quite desirable in many situations.

Let's suppose you're going to build a robotic hand. If you try to use an electric motor to open and close the hand, you must somehow know when to stop it. To do this, you can use some kind of sensor as a feedback control system that tells your RCX the object has been grabbed and the motor can be stopped. However, a pneumatic cylinder, in most cases, needs no feedback. The air pressure closes the hand until it encounters enough resistance to stop it. This approach works in a wide variety of objects. (If your robot is designed to hold eggs, be sure it exerts a very gentle pressure!) Figure 10.6 shows a simple pneumatic hand grabbing different kinds of objects. You see that we used a scissor-like setup that gives our hand a rather large range in regards to the size of the things it can handle.

The previous example gives you an idea of what pneumatics can be used for. Likely, you're already imagining other interesting applications. Unfortunately, the LEGO pneumatic system was not designed to be electrically controlled, so to effectively use it in your robotic projects you need an interface that allows your

RCX to open and close valves. And, unless you plan to run behind your robot pumping like crazy, you probably would like to provide it with an automatic compressor.

Figure 10.6 A Simple Pneumatic Hand

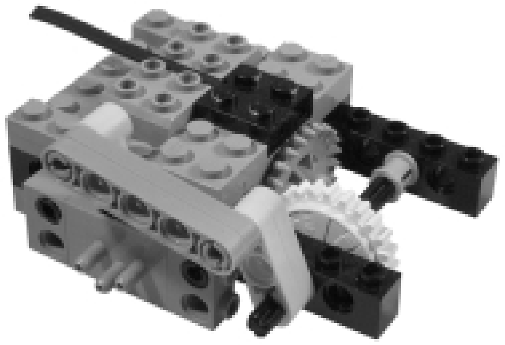


Controlling the Airflow

Almost every LEGO robotics fan would like LEGO to release an electric valve to control pneumatic cylinders, but until it does, you have to get by with mechanical solutions.

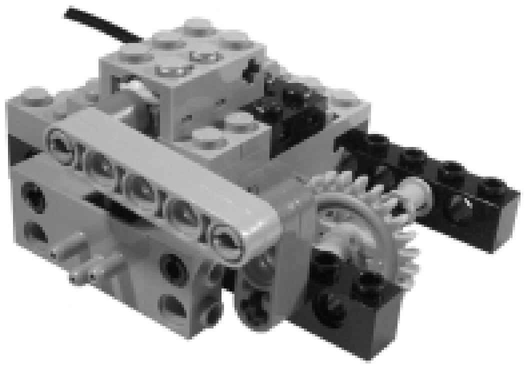
What you need in this case is a kind of indirect control similar to the one we showed in Chapter 2 when talking about the polarity switch. Figure 10.7 shows one of many possible solutions: The motor turns the clutch gear through a crown gear; on the same axle of the clutch gear there's a liftarm that operates the valve. We used the clutch gear as usual to make the timing less critical and avoid any motor problems should it stay on a bit longer than required. You can use a standard 24t gear as well. This might even be necessary if you find a valve stiffer than average. They're not all the same, and some are really hard to operate.

Figure 10.7 An Electric Valve



The downside of our electric valve is that it's able to switch between the two outermost positions, but centering the valve in its neutral position is almost impossible. This is not a big problem, because in most applications you can leave the cylinders connected to the air supply. However, if you really need the central position, you can use a touch sensor that controls when the valve is centered, and utilize a slightly slower gearing, like that shown in Figure 10.8.

Figure 10.8 The Electric Valve with a Sensor



This electric valve is not very compact, but there's not much more you can do considering the size of the LEGO motor. Just the same, it works rather well, and you may feel satisfied with it. But could you make something better? Try applying some of the tricks you learned in previous chapters. For example, you know you can control more than one valve with a single motor. You have seen that, using a differential, it's possible to separate the two turning directions of a motor on two different axles. Now you only need to connect each axle to a valve so the valve cycles between its positions using only one turning direction. This is done using a liftarm as a connecting rod, like in old steam locomotives (see Figure 10.9).

Figure 10.9 A Cycling Valve

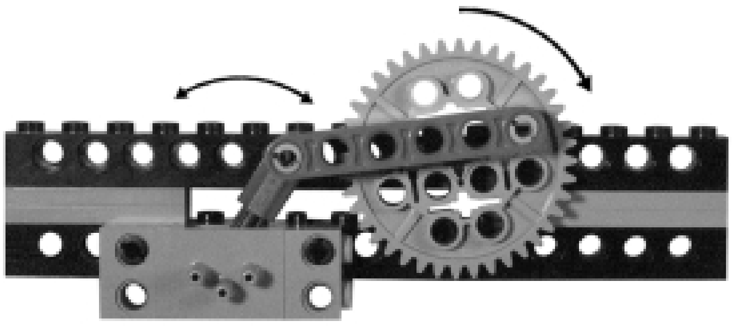
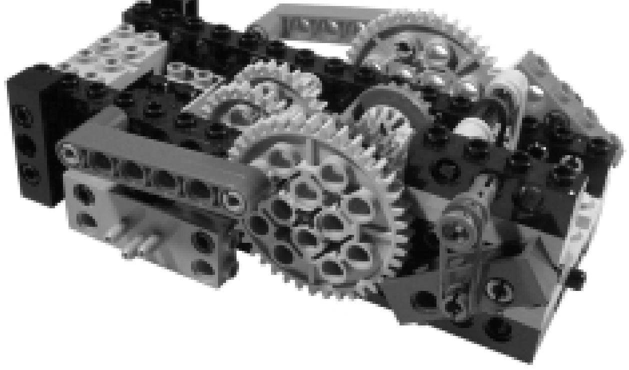


Figure 10.10 shows a prototype of a complete double electric valve, which combines two setups like those of Figure 10.9 with a motor, a differential gear and some additional gearing. We had to use a worm gear to drive the body of the differential because this mechanism requires a lot of torque to be operated. The differential splits the power onto two 40t gears, each one featuring a ratchet beam that lets it rotate only in a specific direction. Thus, when the motor turns clockwise, one valve moves, while if it turns counterclockwise, the other does, each one cycling between all positions.

Figure 10.10 A Single Motor Dual Electric Valve



There's one last problem to solve: How do you know which state each valve is in? And how do you stop the motor precisely when a valve reaches the desired position? Timing it is not an option. As we've said before, it's very difficult to control mechanisms through timing only. You can use sensors, of course, but where should you put them?

Two touch sensors per valve would add up to four sensors; that's a bit too much for your RCX, and anyway you are looking for an option that conserves ports. If you use just two touch sensors, putting them so they close when the valves are at an extreme, you can use timing to go to the other position. In this case, you would avoid timing errors, because you have a sensor that gives you absolute positioning. Can you connect the two sensors to the same port? No, because you wouldn't be able to tell which valve closed the sensor.

What if you placed the sensor so it's closed when the valves are centered? You are not going to use that position to control the pneumatics, but it would still be useful as a reference point for positioning. In order to change the position of the valve, your code has to drive the valve until the sensor closes, and from then on keep the motor running for a small interval to reach the limit point. As in the previous case, you partly rely on timing, but without cumulative errors. The advantage of this configuration is that you can connect both sensors to a single port, because they only close as they pass through a reference point.

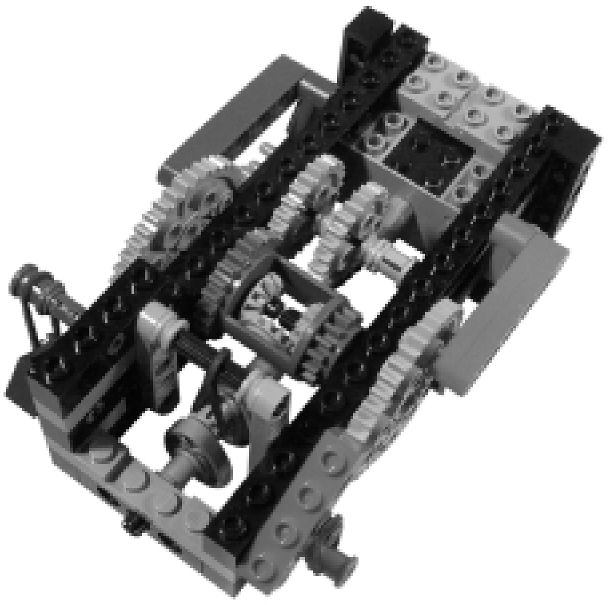
So, we finally figured out how to use one input and one output port instead of two output ports. It's one alternative, and not a big advantage—you can still make it better. If you could just count motor rotations without using a sensor but you can! Do you remember the stepper motor from Chapter 9? Using that configuration, you can avoid using any sensors, thus fully operating two valves with a single motor! The resulting, rather complicated setup is shown in Figure 10.11.

To be honest, we've never used such a thing in any model. It's more of an academic issue, used here to make you understand there are always many ways to solve a problem, and many different paths by which to reach your goals.

Building Air Compressors

Now that you have discovered a way to operate pneumatic cylinders from your RCX, the next step is to provide them with a good supply of compressed air. Some applications require only a small quantity of air for each motion, in which case you have the option to preload a tank by pumping it manually before you run the robot. A good example is a robot that blows out a candle. All it has to do is find the candle in the room, then release its air supply to blow it out. You can extend the range using more tanks (In Chapter 27, we'll describe a robot with seven air tanks.), but for most practical applications you will need something more substantial: an unlimited source of compressed air.

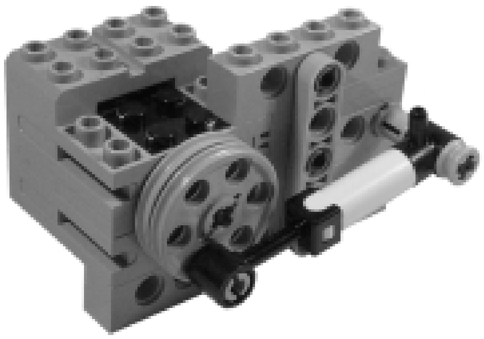
Figure 10.11 A Stepper Motor Dual Electric Valve



This goal is easily achieved by building an electric compressor, like the one shown in Figure 10.12. The small LEGO pump is connected to a pair of pulleys mounted on the shaft of a motor. There are many possible setups, but it's very important you design yours to take advantage of the entire stroke of the pump, because this will make it more efficient. In fact, if your compressor, for example, uses half of the stroke of the pump, it will release only half the maximum quantity of air it could potentially release. In our example, we adjusted the distance using a 1 x 2 two-hole beam, but there are many other possibilities.

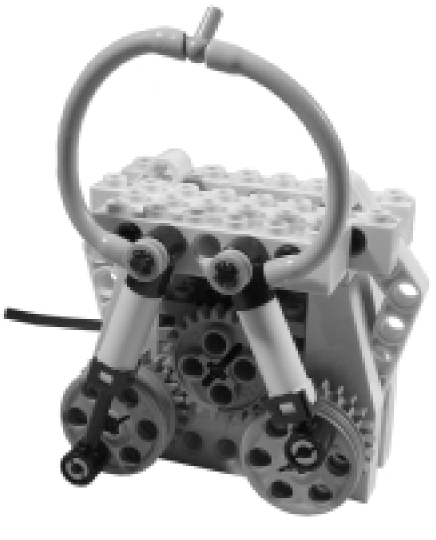
The whole LEGO robotics community is grateful to C.S. Soh, who carefully tested many different compressors, some using two or even four pumps, others using the large hand pump with the spring removed. Using a pressure sensor connected to the RCX, he tested all the common designs and published the results on his site, which, by the way, contains a huge amount of information about LEGO pneumatics in general.

Figure 10.12 A Simple Compressor



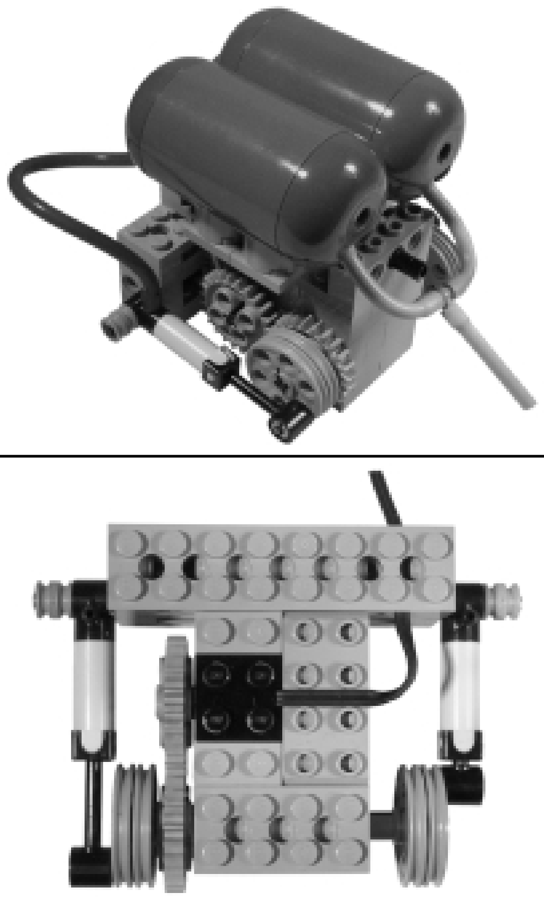
According to Soh's results, the most efficient design is a slightly modified version of Ralph Hempel's compressor (see Figure 10.13). It uses two small pumps and belongs to the category of *double acting compressors*, meaning that one of the pumps takes air in while the other is pumping, thus providing a continuous flow.

Figure 10.13 Ralph Hempel's Double Acting Compressor



In Figure 10.14, you see another double acting compressor with a different design but with an efficiency comparable to Hempel's.

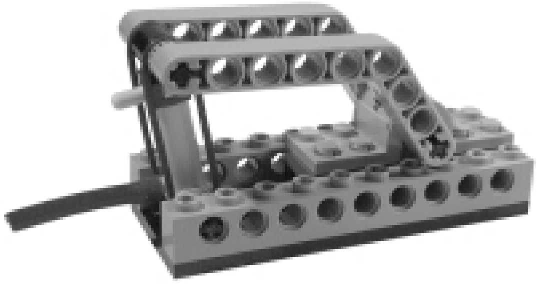
Figure 10.14 Another Double Acting Compressor



The nice thing about compressors is that they don't need to be wired to one of the precious output ports of your RCX: A battery box is enough to run them. But you might wonder when you should stop your compressor, and how. The simplest option is not to stop it. Instead, you can place a torque-limiting component in the gearing, like a pulley or a clutch gear, so that when the pressure reached a given level, the gearing idles. A much more elegant solution again comes from Ralph Hempel and is shown in Figure 10.15.

This clever pressure switch is built around a LEGO polarity switch, a small cylinder, two rubber bands, and some structure beams and plates. The bottom cylinder inlet connects to the air supply circuit of your pneumatic system, and as the pressure increases, the cylinder starts overcoming the resistance of the rubber bands. The movable part of the cylinder connects to two liftarms that operate the polarity switch, one side of which is wired to the battery box, while the other is hooked to the compressor. The polarity switch has three positions: forward, off, and reverse. In this application, you use the first two of them. When the cylinder is retracted, the polarity switch connects the battery box to the compressor. Just before the cylinder reaches its maximum extension, the polarity switch turns off, thus stopping the motor. By adjusting the number and strength of the rubber bands, you can set your pressure switch for the maximum desired pressure, complementing your compressor in a totally automatic system.

Figure 10.15 A Pressure Switch



Building a Pneumatic Engine

We mentioned before that you can make cylinders control other cylinders. This is accomplished by making a cylinder operate the valve that controls a second cylinder. This is not useful in itself, but you can make a cylinder do something *and* move a valve. One very interesting case is one in which you connect two cylinders in a loop where each one controls the other, resulting in an unstable system that continuously, and automatically, changes its state (Figure 10.16). Provided that you have a supply of compressed air, you can take advantage of this feature to make your robot perform an action.

Figure 10.17 shows a diagram of this pneumatic circuit. Cylinder 1 operates valve 1, which controls cylinder 2, which operates valve 2, which controls cylinder 1!

Probably the first robot based on this system to appear publicly on the Internet was Bert van Dam's pneumatic insect. Our slightly modified replica is shown in Figure 10.18.

Figure 10.16 An Unstable Pneumatic System

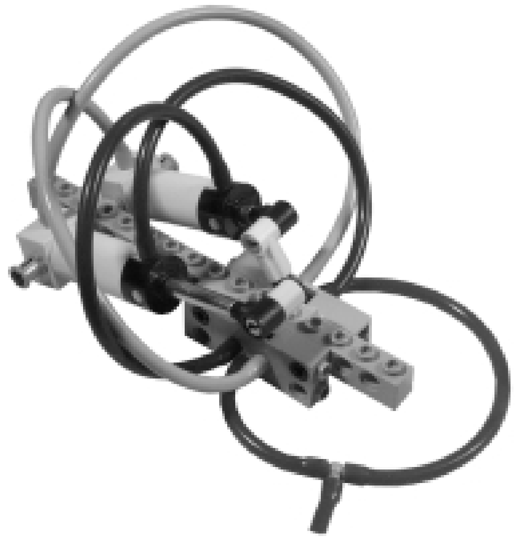


Figure 10.17 Diagram of the Cyclic Pneumatic System

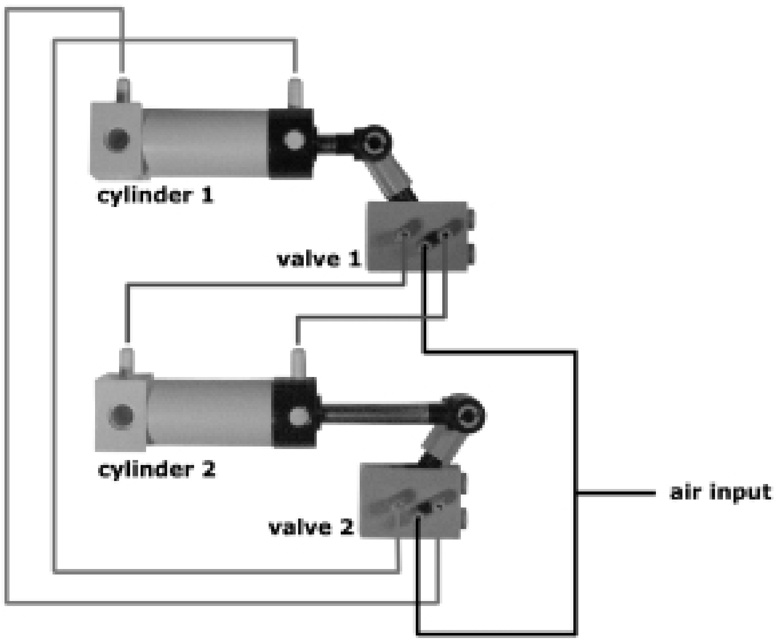
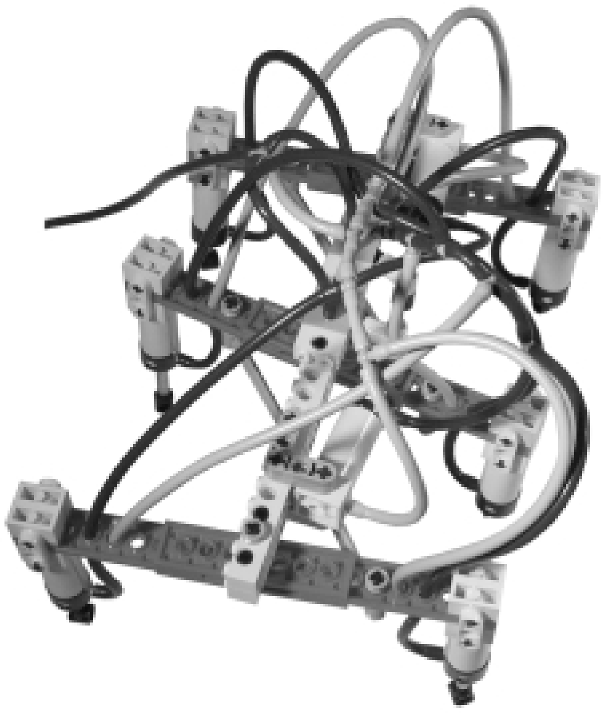
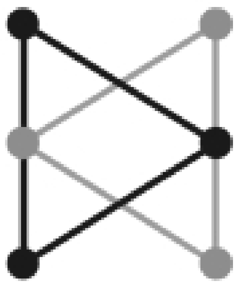


Figure 10.18 Bert van Dam's Pneumatic Insect



The complicated tubing hides the same basic circuit shown in Figure 10.16—one of the control cylinders moves the three leg assemblies forward and backward, while the other moves the legs up and down. These are made of six cylinders, split into two groups of three, controlled by the same valve. Each group has a leg in a central position on one side, and one leg front and one leg rear on the other side (see Figure 10.19).

Figure 10.19 Leg Connection Scheme for the Pneumatic Insect

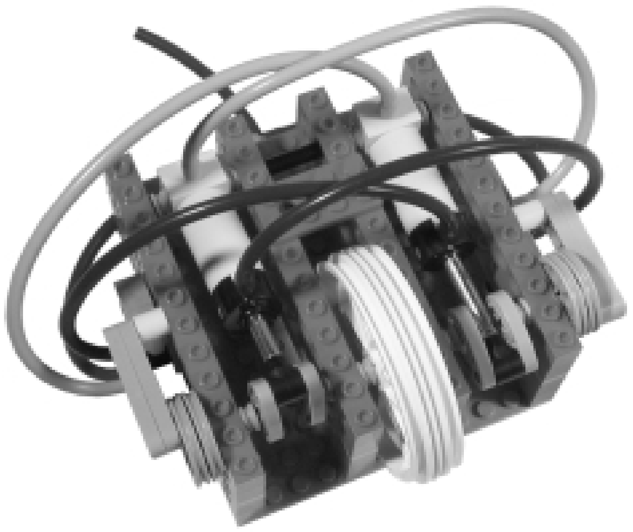


Though rather complicated to build, and more academic an example than practical, Van Dam's insect is quite amazing to see in action.

Using the same principle, it's possible to build a true pneumatic engine, where the push of the cylinders is converted into rotary motion exactly like in steam engines. Figure 10.20 shows our implementation of C.S.Soh's pneumatic engine. The key points are:

- Each cylinder has a dead point in its cycle, when it is either fully extended or retracted. In this position the cylinder is not able to perform any work, as its push/pull force cannot be converted into rotary motion. This happens because the two connection points of the cylinder (on the chassis and the wheel) and the fulcrum of the wheel align along the same line. For this reason, a pneumatic engine with a single cylinder would not work. The addition of a second cylinder solves the problem: You must mount it with a difference of 90° in its phase against the first one, so when one reaches a dead point, the other is at mid-stroke.
- The phasing of the valves is very important: You must take care to position them precisely, otherwise your engine won't work. Mount the wheels on the axles in such a way as to align one of their holes with the holes on the cams. Attach the liftarms to that hole with a gray pin. Connect the tubing exactly like that shown in Figure 10.20.

Figure 10.20 Soh's Pneumatic Engine



Pneumatic engines are capable of high torque, but due to their intrinsic friction are not suitable for high-speed applications. Most of the friction comes from the cylinders themselves, which, in order to be airtight, are a bit stiff to move.

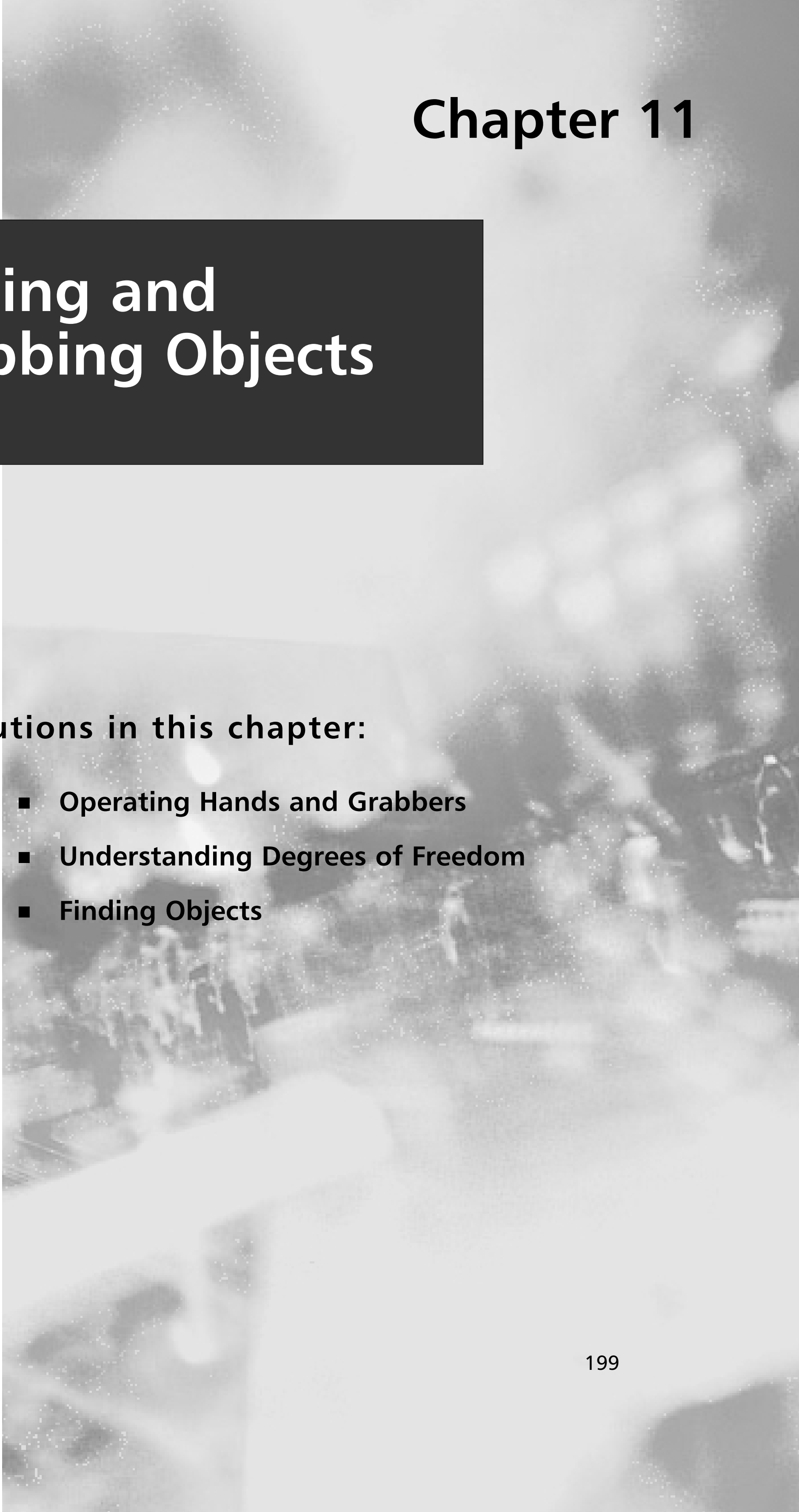
Generally speaking, a vehicle moved by this engine, and supplied by an onboard compressor, is not very efficient. But it's indeed fun to see in action and might have its special uses, too (see Part III).

Summary

Beyond the fascinating sight of all those tubes, and the dramatic hissing of the air coming out of the valves, pneumatics have their practical strong points. In this chapter, you reviewed some basic concepts about the properties of gases, and learned how to exploit these when building your robots. Cylinders are definitely a better choice than electric motors for performing particular tasks, and, most significantly, have the capability to grab objects and create linear motion.

Electric compressors can provide a constant airflow to supply your cylinders, and can be used to control this flow from the RCX. Unfortunately, interfacing pneumatics to the RCX is not so simple, and requires a bulky assembly that includes an electric motor and some gearing. Perhaps in the future, the LEGO Company will produce a smart and compact interface able to control many valves from a single output port.

Pneumatics also offer the opportunity to implement simple automation based on cyclical operation, as we showed in the six-legged walker with its pneumatic engine.



Chapter 11

Finding and Grabbing Objects

Solutions in this chapter:

- Operating Hands and Grabbers
- Understanding Degrees of Freedom
- Finding Objects

Introduction

It's always great fun and very satisfying to see your robot pick things up from the ground, or take an object when you offer it. In this chapter, we'll illustrate some ways to build arms, hands, clamps, pliers, and other tools to grab and handle objects. One of the basic measurements of movement we'll explore is the *degree of freedom* (DOF), or the number of directions in which an object (like a robotic arm) has a range of motion. In the last part of the chapter, we'll show you methods by which your robot can find the objects, the most challenging part of the job.

Operating Hands and Grabbers

In Chapter 10, we illustrated that pneumatic cylinders are generally the ideal choice to make grabbing devices, or *grippers.* Unfortunately, pneumatics is not always a possible option. You might not have LEGO pneumatic parts, or you don't have room on your robot to fit a pneumatic compressor plus a pressure switch and some motor-driven valve switches. We've seen that RCX-to-pneumatics interfaces are rather cumbersome. So you must fall back on good old electric motors to drive your gripper.

The problem with motors is not opening or closing the hand, it's in getting the hand to apply a continuous pressure on the object to prevent it from falling. This means you cannot only position the fingers around it, you must also exert a force that tightens around the object even though you are not moving the fingers anymore. We have explained in Chapters 2 and 3 that if there's one thing that damages electric motors it is having them stalled, or rather having them powered but their movements blocked. For this reason, you cannot simply keep a motor turned on as the hand holds the object, you must employ a trick to prevent the motor from being permanently damaged. When you know you're going to handle a soft object that has some intrinsic elasticity, you can sometimes simply stop the motor and let the friction among gears keep the fingers against it. You can see a simple example of this in Figure 11.1, with an asymmetrical hand designed to grab sponge balls. The worm gear that drives the fingers prevents them from releasing the ball when the motor is not powered. Recall, from Chapter 2, that the worm gear is a one-way gear: It can turn a meshing gear but cannot be turned by it.

Figure 11.1 A Simple Hand Operated with a Worm Gear

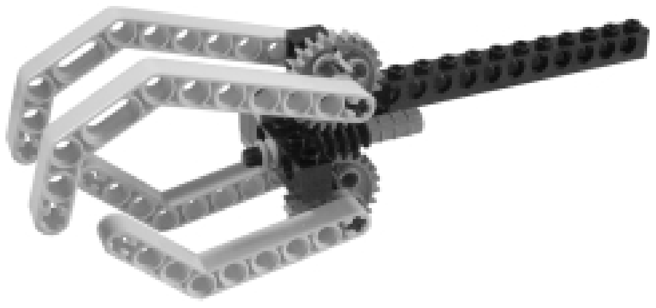
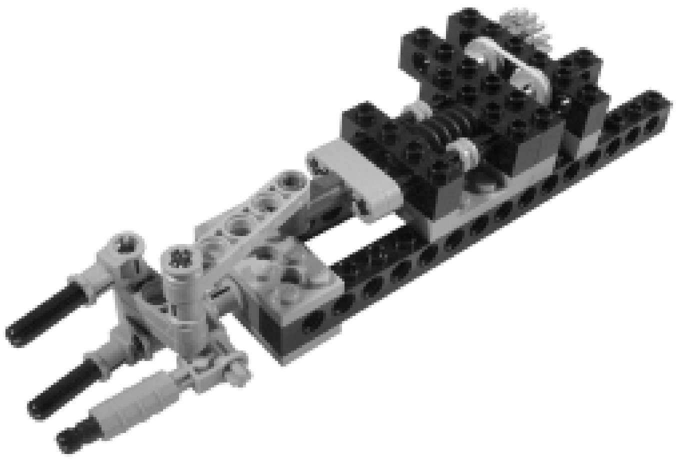


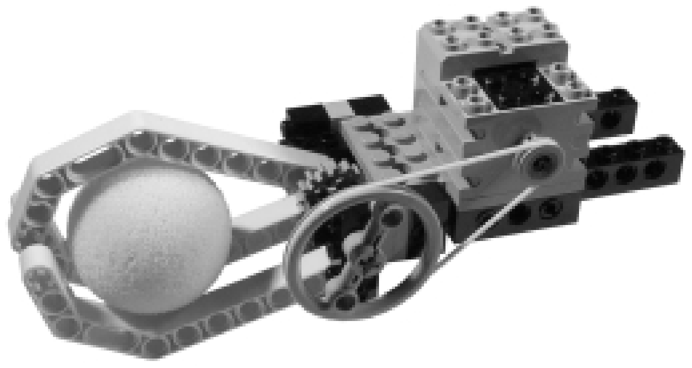
Figure 11.2 shows a different design, where the rotary motion from the motor gets converted into linear motion through a worm gear and two translating axles. It's this motion that operates the movable fingers of a small hand. This mechanism is based on Leo Dorst's Electric Piston: Two half-bushings mesh the teeth of a worm gear; when the worm gear rotates, the bushings get pushed or pulled, and the axles where they are mounted move accordingly. Dorst's solution solves the problem of converting rotary to linear motion using a very compact scheme.

Figure 11.2 This Small Hand Uses Linear Motion



This approach doesn't work when your robotic hand is expected to handle rigid objects or ones of unknown shape and consistency. In these cases, you must introduce some elasticity into your system. Recall from Chapter 2 that you can use a pulley-belt setup to keep the motor running with no harm done even if the system gets blocked. Figure 11.3 shows a simple hand based on this principle: When you turn the motor on, the hand moves until it encounters enough resistance that the belt slips. While you keep the motor on, the belt transmits some force to the fingers and they hold the object. As soon as you stop the motor, however, the pressure of the finger drops and the object is released.

Figure 11.3 Running the Motor to Hold the Object

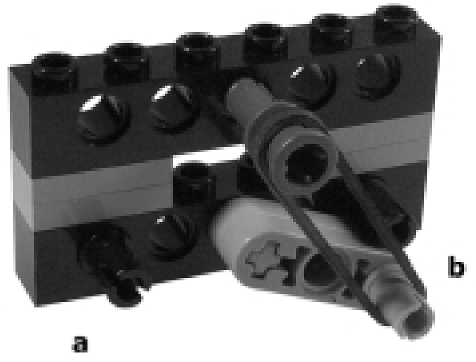


Even though it works, this solution is not very elegant because you're compelled to keep the motor running the entire time you want to hold the object. We suggest you use this system only if the robot must hold the object a very short time. In all other cases, you need something more reliable.

We've repeatedly said that pneumatic cylinders are your best choice in this field, but let's analyze what makes them so good to see if we can learn something and replicate the same behavior. A pneumatic cylinder can be considered a two-state system: The cylinder is either extended or retracted. (We are deliberately ignoring that you can somehow manually stop the cylinder in an intermediate position, centering the switch, and assuming that the switch is either in one of its extreme positions.) If something prevents the cylinder from actually reaching one of these states, it can, however, continue to push in that direction. Its natural behavior is to move until it finds resistance that balances its inner pressure. This pressure is what keeps the fingers applying a force to the object, thus making your robotic hand hold it firmly.

The point now is to replicate this behavior in a nonpneumatic device. Is it possible? Yes. Figure 11.4 contains an example of a simple *bi-stable* system, called bi-stable because it has two default states, two possible rest positions which it tends to go to. A rubber band forces the liftarm to stay against one of the two black pegs, either in A or in B. If you move the liftarm slightly from the peg and then release it, it goes back against the peg. If you move it a bit more and pass the midpoint between A and B, it goes to the other peg. You need to provide only enough force to make the system switch from one to the other; the rubber band will do the rest.

Figure 11.4 A Simple Bi-Stable Mechanism



Applying this principle, we designed the pliers shown in Figure 11.5, which are suitable for grabbing very small objects like a 1 x 2 brick (seen at the bottom of the Figure between the two plates). To actually use them in a robot you must add a motor that, through brief impulses, pushes the pliers into their open or closed states. As usual, you would probably involve a belt or a clutch gear to make the timing of the motor not critical.

The same approach can be used for larger and more complex hands, like the one shown in Figure 11.6, where the bi-stable mechanism has been placed on intermediate gearing.

Transferring Motion Using Tubing

In discussing the advantages of pneumatics when grabbing objects, we must also mention that tubing provides a simple way to keep bulky things far from the movable parts. Compare the simplicity of the pliers in Figure 11.7 with the complex gearings of the previous examples. The difference is dramatic.

Figure 11.5 Bi-Stable Pliers

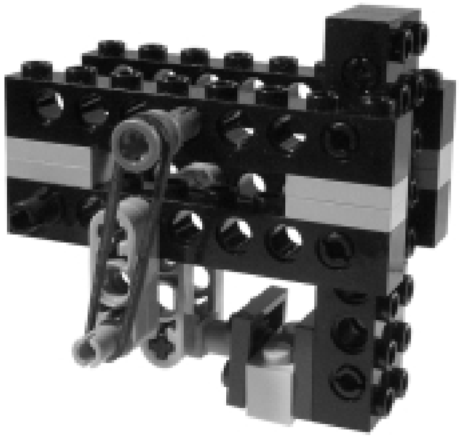


Figure 11.6 A Bi-Stable Large Hand

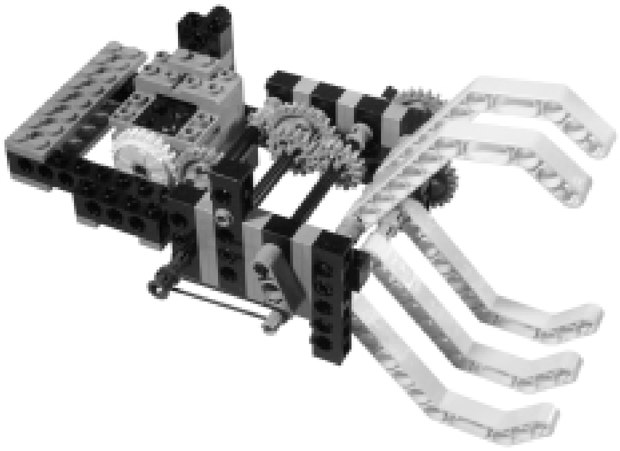
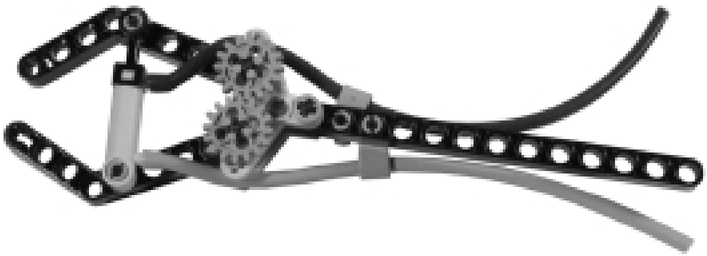
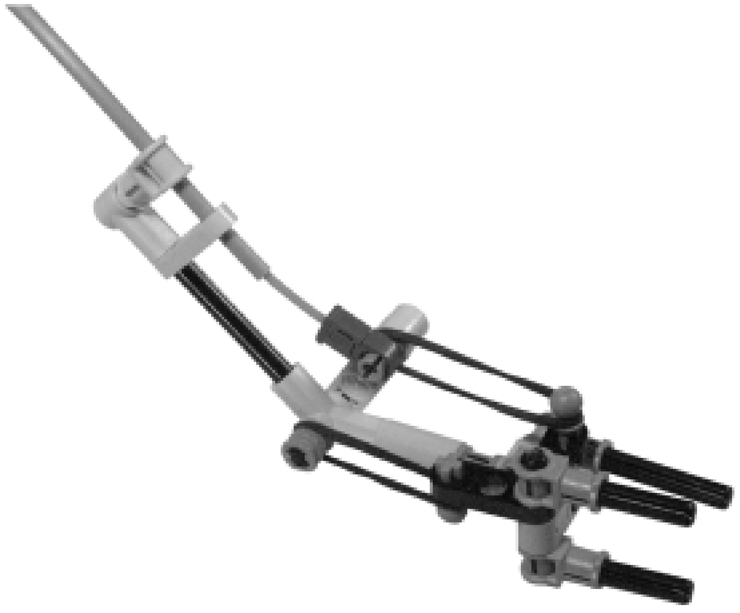


Figure 11.7 Pneumatics Helps in Making Essential and Clean Assemblies



The flex system we briefly described in Chapter 9 has similar properties, allowing you to transfer motion to distant parts. Our robot, Cinque, features a small operating hand based on this technique (see Figure 11.8). A pair of opposing rubber bands introduce a degree of elasticity into the system, and help the fingers return to their default setting once the hand comes to rest in its open position.

Figure 11.8 The Flex System Helps in Making Lightweight Hands



Understanding Degrees of Freedom

If you look carefully at your hands, you'll discover they are an incredible piece of machinery, capable of handling a wide array of objects of every size and shape. Just think about the long list of verbs describing all the things hands can do: grab, handle, hold, take, squeeze, grip, point, pinch, shake, roll, press, grasp, push, pull and those are only a few of the terms. Where does all this versatility come from?

Observe a finger while you move it, you notice four independent movements: three for the joints—from the finger tip to the hand—that let you bend the finger, and a fourth that allows for slight left-right motion where the finger joins the hand. Multiply this by five (for a hand's five fingers) and add the mobility given by the wrist, and 25 movements or so come to mind, which, in turn, lead to a huge number of combinations and configurations. This is what makes your hand able to conform to the shape of the object you want to handle. To complete the picture, consider that you can control the strength of each muscle so finely that you can pick up a delicate wine glass without damaging it, yet so firmly grip a baseball bat that you can send a ball over the right field wall.

Every independent movement represents a degree of freedom (DOF), something that can happen without affecting and being affected by other movements in the same device. Our previous examples were very simple mechanisms with just one degree of freedom, being that all the possible positions of the "fingers" were determined by a single motor or pneumatic cylinder. The DOF concept helps you understand in terms of numbers why those simple hands diverged so widely from the flexibility a human hand has.

Obviously, you cannot aim at making a robotic hand with 25 degrees of freedom using your MINDSTORMS kit. Each degree of freedom will typically require a dedicated motor or pneumatic cylinder, and this puts the task out of reach. You should stay with something much simpler and consequently reduce the range of objects your mechanical hand will be able to grab. This is sometimes limiting, but in many situations you will know in advance the type and shape of objects your robot will be expected to handle, making your task less demanding. In contests that involve collecting things, for example, your robot usually will deal with very specific objects like soda cans, small LEGO cubes or marbles, and because of this you can design it to target those types of objects.

It is possible, however, to build more versatile hands with more degrees of freedom. Figure 11.9 shows a 3 degrees of freedom pneumatic finger. This is a nice design, but it's a pity it requires all three ports of your RCX to be fully controlled. How could you control more than a finger if you are already out of ports? To make the system simpler, though still useful, you can connect all the cylinders together. (You won't be able to move a single segment of the finger by itself, but the finger can still adapt well to the shape of many different objects.) This is the technique we used in the three-finger pneumatic hand shown in Figures 11.10 and 11.11, which is controlled by a single valve switch.

Figure 11.9 A Three Degrees of Freedom Pneumatic Finger

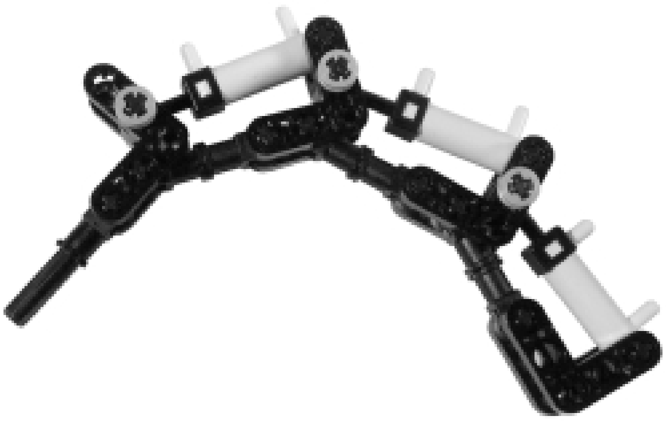


Figure 11.10 A Three-Finger Pneumatic Hand

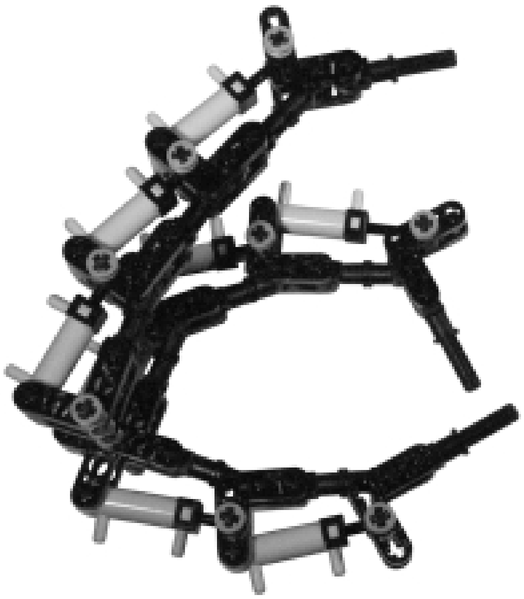
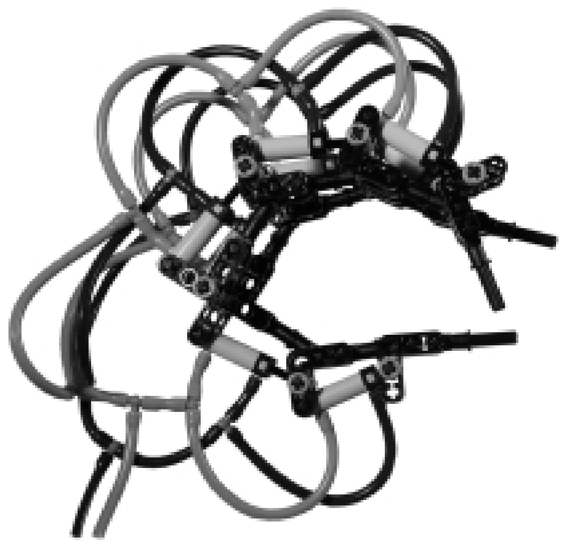
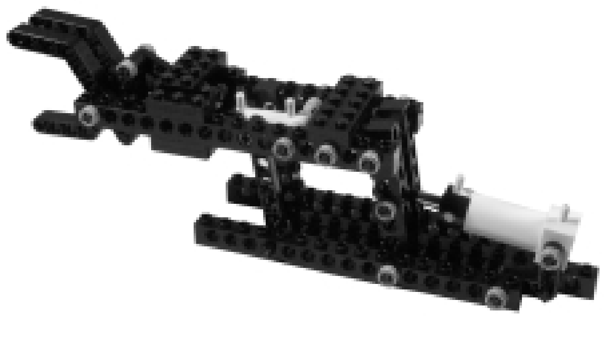


Figure 11.11 The Three-Finger Pneumatic Hand with Complete Tubing



The degrees of freedom concept applies not only to hands but to any mechanical device. The arm of Figure 11.12, taken from our R2-D2 styled robot "Otto," has two degrees of freedom: A large cylinder extends the arm from the body of the robot, and a small one operates the hand.

Figure 11.12 The Robotic Arm from Our "Otto" Robot



Generally speaking, locating a point in a plane requires two DOFs, while locating a point in space requires three. There are many examples of 2 DOF- and 3 DOF-mechanisms in everyday objects: An ink-jet printer has two DOFs, one corresponding to the head movement and the other to the paper feeding. A construction crane is an example of a machine with three DOFs: The hook can go up and down, it's attached to a carriage that moves back and forth along the boom, and, finally, the boom can rotate. With the three output ports of your RCX, you can drive a robotic arm that addresses any point inside a delimited space, called the *operating envelope*, exactly like the crane of the previous example. If you also want to pick up and drop objects, you would need another port, or use some of the tricks from Chapter 9 to control more than a DOF with a single motor.

Finding Objects

Building robotic arms and hands is the easy part of the job, the hard part is finding the objects to grab. We will skip the case where your robot *knows* the position of one of the objects, because this brings into play a general navigation problem we'll discuss in Chapter 13. So, for the time being, we'll stick with the fact that the robot knows nothing about the location of the object.

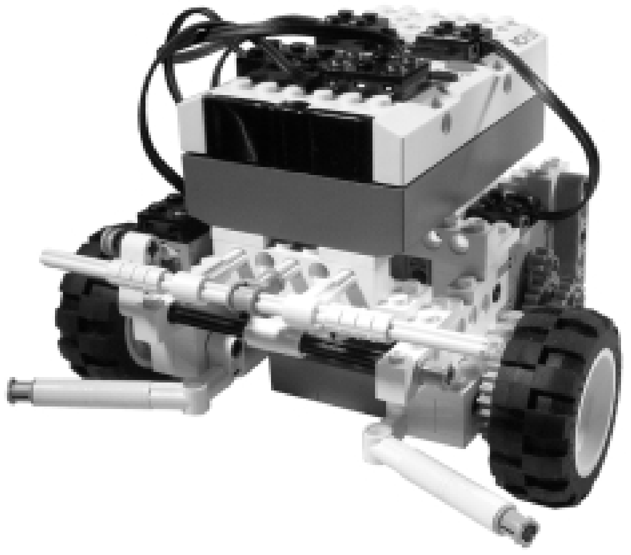
As we explained when talking about bumpers in Chapter 4, navigation in real environments is quite a tough task, and distinguishing a specific object from others makes things much harder. So the second assumption we make here is that you know what kind of object you're expected to handle, as well as all the details of the environment where your robot moves (typically an artificial one prepared for the task). You might think that we are introducing too many simplifications here, but even in these conditions, the task remains quite hard. It's very important that you progress in short steps. The most common mistake of beginning builders is to start out with goals too difficult for their robots, where mechanical and programming difficulties add to navigation problems. As a general approach, we suggest you apply the "divide and conquer" strategy and solve the problems one by one.

Let's make an example: A simple variation on line following that might involve removing objects placed along the path. A very simple bumper is probably enough to detect objects. The arm will be more or less sophisticated depending on whether you have to collect them or just move them out of the way.

In wider environments, things become trickier. Imagine you have to find things in a delimited space with no walls. (How could a space be delimited without having walls? By using different colors on the floor and reading them with a light sensor facing down!) You can still use a bumper, and make your robot move around at random or follow some kind of scheme. Depending on whether you are participating a contest with specific rules, you could make this approach more efficient using a sort of funnel to convey the objects against the bumper, or some long antennas to help you detect contacts in a wider area. The robot of

Figure 11.13 was designed to find small LEGO cubes during a contest, and takes advantage of the fact that the height of the object is known precisely enabling us to detect the cubes with a top bumper.

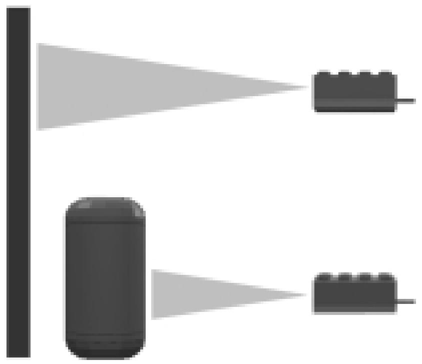
Figure 11.13 StudWhite, a LEGO Cubes Finder

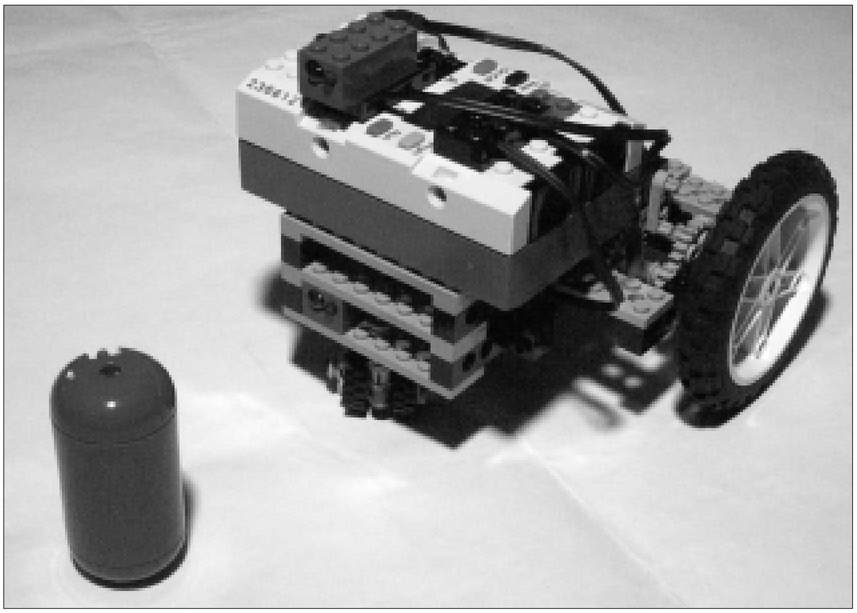


In other situations, you can apply the proximity detection technique, either with standard LEGO components as described in Chapter 4, or with custom IRPD sensors like the one shown in Chapter 9. Let's go back to the example where there are no walls. You can use proximity to "see" the objects, maybe improving final detection with a bumper as in previous scenarios. And if there *are* walls? Well, you'll need a way to distinguish the objects from the walls.

The easiest approach is to rely again on the shape of the object. Usually the walls are taller then the soda cans or marbles you have to find, so you can prepare two bumpers at different heights and see which one closes to decide what your robot ran into. The same works with proximity detection: Placing two sensors at different heights will tell you whether you've found a soda can or the wall (Figure 11.14). Be careful though… Two or more active custom proximity sensors, the kind that emit their own IR beam, can interfere with each other, resulting in unreliable readings. Instead of receiving back just the IR light that they emit, each one will also receive the IR light emitted by their brother. To avoid this problem, you have to write your software to make them active one at a time. This can be achieved configuring them as passive sensors (for example, as touch sensors), so they don't receive any power, and consequently don't emit any IR beam. Your program will configure them as active sensors just before performing the reading, and will change them to passive sensors again afterward.

Figure 11.14 Using Two IRPDs to Distinguish Objects from Walls





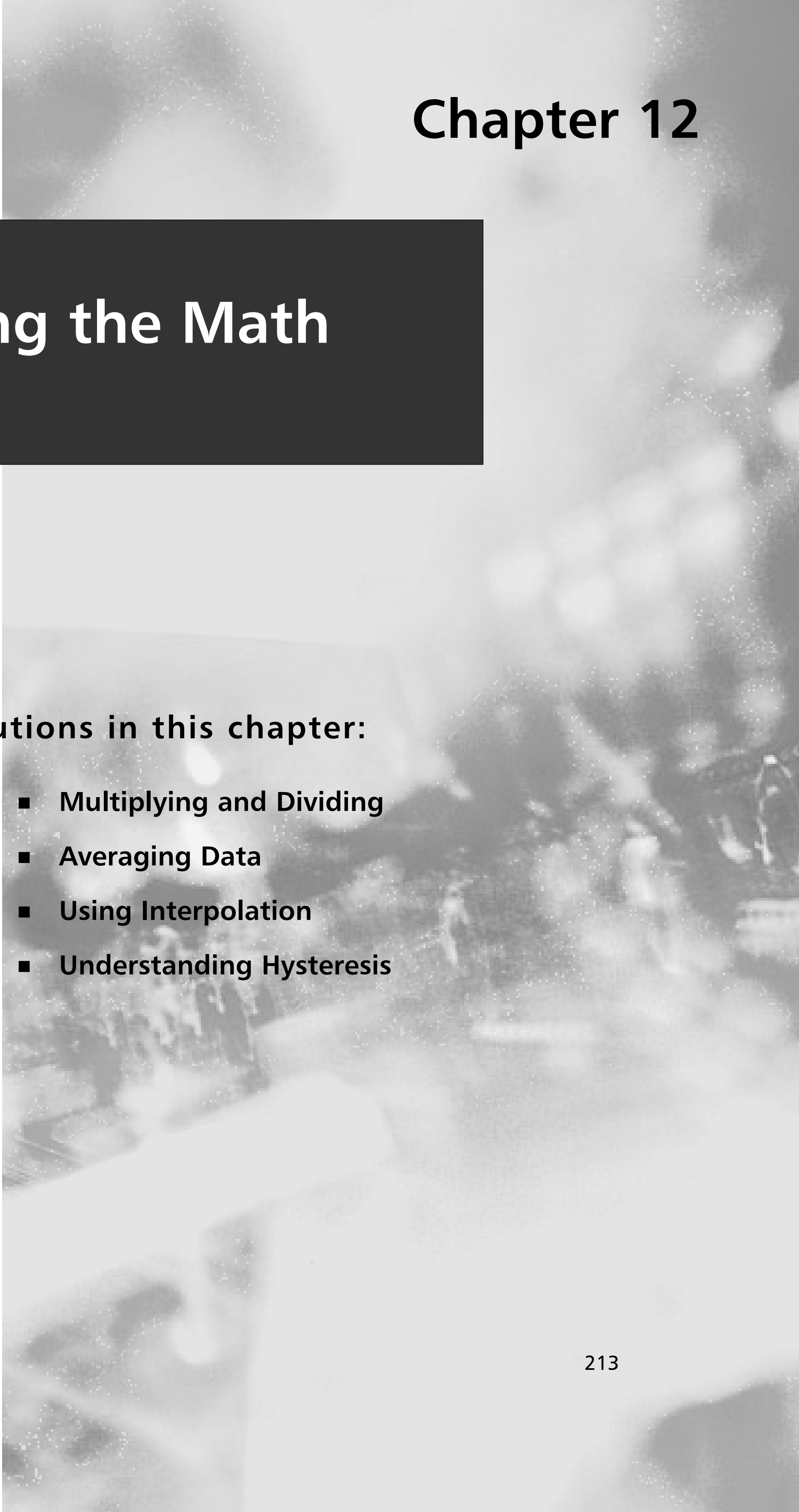
A different case is when you want to manually trigger your robot to grab or release objects. This is very easy to implement with a touch sensor, a push button that you press when you want your robot to open or close its hand. Proximity detection makes your robot even more impressive to see in action. You can, for instance, build a robot that navigates the room, and that, when you offer it an object, stops to grab it. This technique is a bit tricky to use if your robot is expected to navigate a room with walls and other obstacles, because it won't be able to tell what triggered its proximity detection, unless you have a custom sensor that returns some relative or absolute measurement of the distance. In this case, you can continuously monitor the distance and interpret a sudden radical change in its movement as a request to grab or release objects.

Summary

Designing a good robotic hand or arm is more of an art than a technique. There are indeed technical issues when it comes to gearing and pneumatics that you must know and consider to successfully position the grippers or hands, apply the right amount of pressure, troubleshoot the elasticity of the object to be grabbed, and not allow your robot to drop the ball (or object rather). Even then, there's still a lot of space for good intuitions and heavy prototyping. You can choose pneumatic or nonpneumatic approaches, design for different degrees of freedom in your gripping arm, use a flex system with tubing for lightweight designs, and create solutions that reserve ports for additional functions.

To make an easy start, target your first projects around a specific type of object, then progress to more versatile grabbers only when you feel experienced and confident enough to meet the challenge.

We also explained that finding the object is the hardest part of the job, but there are cases where you can use a random search pattern, or where the object sits on the robot's path, as in the line following example.



Chapter 12

Doing the Math

Solutions in this chapter:

- **Multiplying and Dividing**
- **Averaging Data**
- **Using Interpolation**
- **Understanding Hysteresis**

Introduction

You may be surprised to find a chapter about mathematics in a book aimed at explaining building techniques. However, just as one can't put programming aside totally, so too we cannot neglect an introduction to some basic mathematical techniques. As we've explained, robotics involves many different disciplines, and it's almost impossible to design a robot without considering its programming issues together with the mechanical aspects. For this reason, some of the projects we are going to describe in Part II of the book include sample code, and we want to provide here the basic foundations for the math you will find in that code. Don't worry, the math we'll discuss in this chapter doesn't require anything more sophisticated than the four basic operations of adding, subtracting, multiplying, and dividing. The first section, about multiplying and dividing, explains in brief how computers deal with integer numbers, focusing on the RCX in particular. This topic is very important, because if you are not familiar with the logic behind computer math you are bound to run into some unwanted results, which will make your robot behave in unexpected ways.

The three subsequent sections deal with *averages*, *interpolation*, and *hysteresis*. Though they are not essential, you should consider learning these basic mathematical techniques, because they can make your robot more effective while at the same time keep its programming code simpler. Averages cover those cases where you want a single number to represent a sequence of values. School grades are a good example of this: They are often averaged to express the results of students with a single value (as in a grade point average). Robotics can benefit from averages on many occasions, especially those situations where you don't want to put too much importance on a single reading from a sensor, but rather observe the tendency shown by a group of spaced readings.

Interpolation deals with the estimating, in numerical terms, of the value of an unknown quantity that lies between two known values. Everyday life is full of practical examples—when the minute hand on your watch is between the Three and Four marks, you interpolate that data and deduce that it means, let's say, eighteen minutes. When a car's gas gauge reads half a tank, and you know that with the full tank the car can cover about four hundred miles, you make the assessment that the car can currently travel approximately two hundred miles before needing refueling. Similarly in robotics, you will benefit from interpolation when you want to estimate the time you have to operate a motor in order to set a mechanism in a specific position, or when you want to interpret readings from a sensor that fall between values corresponding to known situations.

The last tool we are going to explore is *hysteresis*. Hysteresis defines the property of a variable for which its transition from state A to state B follows different rules than its transition from state B to state A. Hysteresis is also a programmed behavior in many automatic control devices, because it can improve the efficiency of the system, and it's this facet that interests us. Think of hysteresis as being similar to the word "tolerance," describing, in other words, the amount of fluctuation you allow your system before undertaking a corrective action. The hysteresis section of the chapter will explain how and why you might add hysteresis to the behavior of your robots.

Multiplying and Dividing

If you are not an experienced programmer, first of all we want to warn you that in the world of computers, mathematics may be a bit different from what you've been taught in school, and some expressions may not result in what you expect. The math you need to know to program the small RCX is no exception.

Computers are generally very good at dealing with *integer* numbers, that is, whole numbers (1, 2, 3...) with the addition of zero and negative whole numbers. In Chapter 6, we introduced *variables*, and explained that variables are like boxes aimed at containing numbers. An *integer variable* is a variable that can contain an integer number. What we didn't say in Chapter 6 is that variables put limits on the size of the numbers you can store in them, the same way that real boxes can contain only objects that fit inside. You must know and respect these limits, otherwise your calculations will lead to unexpected results. If you try to pour more water in a glass than what it can contain, the exceeding water will overflow. The same happens to variables if you try to assign them a number that is greater than their capacity—the variable will only retain a part of it.

The firmware of the RCX has been designed to manipulate integer numbers in the range –32768 through 32767. This means that when using either RCX Code, NQC, or any other language based upon the LEGO firmware, you must keep the results of your calculations inside this range. This rule applies also to any intermediate result, and entails that you learn to be in control of your mathematics. If your numbers are outside this range, your calculations will return incorrect results and your robot will not perform as expected; in technical terms, this means you must know the *domain* of the numbers you are going to use. Multiplication and division, for different reasons, are the most likely to give you trouble.

Let's explain this statement with an example. You build a robot that mounts wheels with a circumference of 231mm. Attached to one wheel is a sensor geared to count 105 ticks per each turn of the wheel. Knowing that the sensor reads a count of 385, you want to compute the covered distance. Recall from Chapter 4 that the distance results from the circumference of the wheel multiplied by the number of counts and divided by the counts per turn:

231 x 385 / 105 = 847

This simple expression has obviously only one proper result: 847. But if you try to compute it on your RCX, you will find you can *not* get that result. If you perform the multiplication first, that is, if the expression were written as follows:

(231 x 385) / 105

you get 222! If you try and change the order of the operations this way:

231 x (385 / 105)

you get 693, which is closer but still wrong! What happened? In the first case, the result of performing the multiplication first (88,935) was outside the upper limit of the allowed range, which is only 32,767. The RCX couldn't handle it properly and this led to an unexpected result. In the second case, in performing the division operation first, you faced a different problem: The RCX handles only integers, which cannot represent fractions or decimal numbers; the result from 385 / 105 should have been 3 2/3, or 3.66666..., but the processor truncated it to 3 and this explains the result you got.

Unfortunately, there is no general solution to this problem. A dedicated branch of mathematics, called *numerical analysis*, studies how to limit the side effects of mathematical operations on computers and quantify the expected errors and their propagation along calculations. Numerical analysis teaches that the same error can be expressed in two ways: *absolute errors* and *relative errors*. An absolute error is simply the difference between the result you get and the true value. For example, 4355 / 4 should result in 1088.75; the RCX truncates it to 1088, and the absolute error is 1088.75 – 1088 = 0.75. The division of 7 by 4 leads to the same absolute error: The right result is 1.75, it gets truncated to 1, and the absolute error is again 0.75. To express an error in a relative way, you divide the absolute error by the number to which it refers. Usually, relative errors gets converted into *percentage errors* by multiplying them by one hundred. The percentage errors of our previous examples are quite different one from the other: 0.07 percent for the first one (0.75 / 1088.75 x 100) and an impressive 42.85 percent error for the latter (0.75 / 1.75 x 100)! Here are some useful tips to remember from this complex study:

- You have seen that integer division will result in a certain loss of precision when decimals get truncated. Generally speaking you should perform divisions as the *last step* of an expression. Thus the form (A x B) / C is better than A x (B / C), and (A + B) / C is better than its equivalent A / C + B / C.
- While integer divisions lead to small but predictable errors, operations that go off-range (called *overflows* and *underflows*) result in gross mistakes (as you discovered in the example where we multiplied 231 by 385). You must avoid them at all costs. We said that the form (A x B) / C is better than A x (B / C), but *only* if you're sure A x B doesn't overflow the established range!
- When dividing, the larger the dividend over the divisor, the smaller the relative error. This is another reason (A x B) / C is better than A x (B / C): The first multiplication makes the dividend bigger.
- Prescaling values to relocate them in a different range is sometimes a good option, especially if you can do so without a loss in accuracy. For example, if you are dealing with raw values coming from a light sensor, they will likely be in the range of 550 to 850. In the event you had to multiply them with other numbers, you could subtract 500 from all your readings to move them down into the range of 50 to 350.

Designing & Planning…

Floating-Point Numbers

If you are familiar with computer programming, you probably know that many languages support another common numeric format: *floating-point*. The internal representation of a floating-point number is made up of two values, a *mantissa* and an *exponent*, and corresponds to the number that results multiplying the mantissa by a conventional *base* raised to the exponent. This technique allows floating-point variables to handle numbers in a very wide domain.

Continued

Up to this point, we deliberately omitted talking about them for two reasons. First, the RCX firmware doesn't support floating-points (currently only legOS and leJOS can handle them), and second, they don't result, by themselves, in a greater precision. As for integers, their precision is limited to the number of bits used to map them in memory.

We admit that they do provide a convenient way to represent values from a wider range then integers, with fewer concerns about overflows and truncations, but in robotics it's really possible to face most situations with only integer math.

Averaging Data

There are situations when you may prefer that your robot base its decisions not on a single sensor reading but on a *group* of them, to achieve more stable behavior. For example, if your robot has to navigate a pad made up of colored areas rather than just black and white, you would want it to change its route only when it finds a different color, ignoring transition areas between two adjacent colors (or even dirt particles that could be "read" by accident).

Another case is when you want to measure ambient light, ignoring strong lights and shadows. *Averaging* provides a simple way to solve these problems.

Simple Averages

You're probably already familiar with the simple average, the result of adding two or more values and dividing the sum by the number of addends. Let's say you read three consecutive light values of 65, 68, and 59, their simple average would be:

$$(65 + 68 + 59) / 3 = 64$$

which is expressed in the following formula:

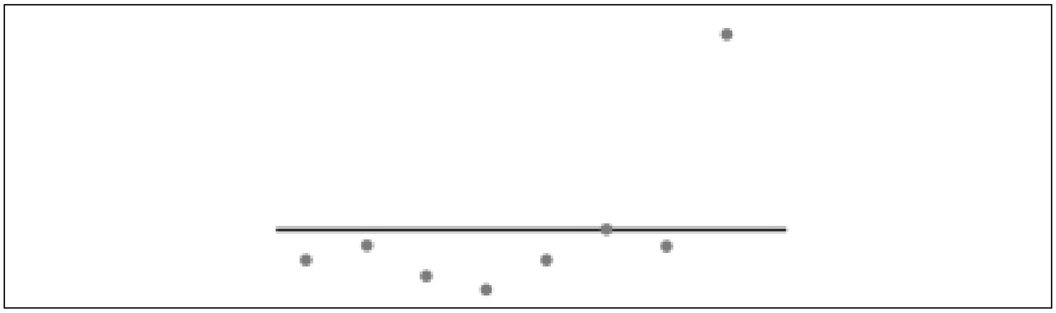
$$A = (V_1 + V_2 + \ldots + V_n) / n$$

The main property of the average, what actually makes it useful to our purpose, is that it smoothes single peak values. The larger the amount of data averaged, the smaller the effect of a single value on its result. Look at the following sequence:

60, 61, 59, 58, 60, 62, 61, 75

The first seven values fall in the range of 58 to 62, while the eighth one stands out with a 75. The simple average of this series is 62, thus you see that this last reading doesn't have a strong influence (Figure 12.1).

Figure 12.1 How Averaging Smoothes Peaks and Valleys in the Data



In your practical applications, you won't average all the readings of a sensor, usually just the last n ones. It is like saying you want to benefit from the smoothing property of an average, but only want to consider more recent data because older readings are no longer relevant.

Every time you read a new value, you discard the oldest one with a technique called the *moving average*. It's also known as the boxcar average. Computing a moving average in a program requires you to keep all the last n values stored in variables, then properly initialize them before the actual executions begins. Think of a sequence of sensor values in a long line. Your "boxcar" is another piece of paper with a rectangular cutout in it, and you can see exactly n consecutive values at any one time. As you move the boxcar along the line of sensor values, you average the readings you see in the cutout. It is clear that as you move the boxcar by one value from left to right along the line, the leftmost value drops off and the rightmost value can be added to the total for the average.

Going back to the series from our previous example, let's build a moving average for three values. You need the first three numbers to start: 60, 61, and 59. Their average is (60 + 61 + 59) / 3 = 60. When you receive a new value from your sensor, you discard the oldest one (60) and add the newest (58). The average now becomes (61 + 59 + 58) / 3 = 59.333… Figure 12.2 shows what happens to the moving average for three values applied to all the values of the example.

When raw data shows a trend, moving averages acknowledge this trend with a "lag." If the data increases, the average will increase as well, but at a slower pace. The higher the number of values used to compile the average, the longer the lag.

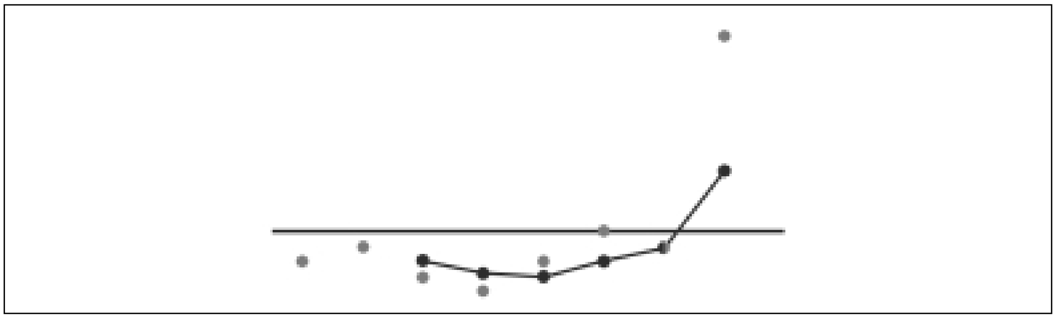
Suppose you want to use a moving average for three values in a program. Your NQC code could be as follows:

```
int ave, v1, v2, v3;

v2 = SENSOR_1;
v3 = SENSOR_1;

while (true)
{
  v1 = v2;
  v2 = v3;
  v3 = SENSOR_1;
  ave = (v1+v2+v3) / 3;
  // other instructions...
}
```

Figure 12.2 A Moving Average for Three Values



Note the mechanism in this code that drops the oldest value (v1), replacing it with the subsequent one (v2), and that shifts all the values until the last one is replaced with a fresh reading from the sensor (in v3). The average can be computed through a series of additions and a division.

When the number of reading being averaged is large, you can make your code more efficient using *arrays*, adopting a trick to improve the computation time and keep the number of operations to a minimum. If you followed the description of the boxcar cutout as it moved along the line, you would realize that the total of the values being averaged did not have to be calculated every time. We just need to subtract the leftmost value, and add the rightmost value to get the new total!

A circular pointer, for example, can be used to address a single element of the array to substitute, without shifting all the others down. The number of additions, meanwhile, can be drastically decreased keeping the total in a variable, subtracting the value that exits, and adding the entering one. The following NQC code provides an example of how you can implement this technique:

```
#define SIZE 3
int v[SIZE],i,sum,ave;

// initialize the array
sum = 0;
for (i=0;i<SIZE-1;i++)
{
  v[i] = SENSOR_1;
  sum += v[i];
}

// first element to assign is the last of the array
i=SIZE-1;
v[i]=0;

// compute moving average
while (true)
{
  sum -= v[i]; //
  v[i] = SENSOR_1;
  sum += v[i];
  ave = sum / SIZE;
  i = (i+1) % SIZE;
  // other instructions...
}
```

The constant SIZE defines the number of values you want to use in your moving average. In this example, it is set to 3, but you can change it to a different number. The statement **int** declares the variables; **v[SIZE]** means that the variable **v** is an *array*, a container with multiple "boxes" rather than a single "box." Each *element* of the array works exactly like a simple variable, and can be addressed specifying its position in the array. Array elements are numbered starting from 0, thus in an array with 3 elements they are numbered 0, 1, and 2. For example, the second element of the array **v** is **v[1]**.

This program starts initializing the array with readings from the sensor. It uses the **for** control statement to loop SIZE-1 times, at the same time incrementing the **i** variable from 0 to SIZE-1. Inside the loop, you assign readings from the sensor to the first SIZE-1 elements of the array. At the same time, you add those values to the **sum** variable. Supposing that the first readings are 72 and 74, after initialization **v[0]** contains 72, **v[1]** contains 74, and **sum** contains 146. The initialization process ends assigning to the variable **i** the number of the first array element to replace, which corresponds to SIZE-1, which is 2 in our example.

Let's see what happens inside the loop that computes the moving average. Before reading a new value from the sensor, we remove the oldest value from **sum**. The first time **i** is 2 and **v[i]**, that is **v[2]**, is 0, thus **sum** remains unchanged. v**[i]** receives a new reading from the sensor and this is added to **sum**, too. Supposing it is 75, **sum** now contains 146 + 75 = 221. Now you can compute the average **ave**, which results in 221 / 3 = 73.666…, and which is truncated to 73. The following instruction prepares the pointer **i** to the address of the next element that will be replaced. The symbol **%** in NQC corresponds to the *modulo* operator, which returns the remainder of the division. This is what we call a *circular pointer*, because the expression keeps the value of **i** in the range from 0 to SIZE-1. It is equivalent to the code:

```
i = i+1;
if (i==SIZE) i=0;
```

which resets **i** to 0 when it reaches the upper bound. The resulting effect is that **i** cycles among the values 0, 1, and 2.

During the second loop **i** is 0, so **sum** gets decreased to **v[0]**, that is 72, and counts 221 – 72 = 149. **v[0]** is now assigned a new reading, for example 73, and **sum** becomes equal to 149 + 73 = 222. The average results 222 / 3 = 74, and **i** is incremented to 1. Then the cycle starts again, and it's time for **v[1]** to be replaced with a new value.

This program is definitely much more complicated than the previous one, but has the advantage of being more general: It can compute moving averages for any number of values by just changing the SIZE constant. The only limit to the maximum value of SIZE is the total number of variables allowed by the LEGO firmware, which is 32. Each array element counts as a variable.

Weighted Averages

We explained that simple averages have the property of smoothing peaks and valleys in your data. Sometimes, though, you want to smooth data to reduce the effect of single readings, yet at the same time put more importance on recent values than older ones. In fact, the more recent the data, the more representative the possible trend in the readings.

Let's suppose your robot is navigating a black and white pad, and that it's crossing the border between the two areas. The last three readings of its light sensor are 60, 62, and 67, which result in a simple average of 63. Can you tell the difference between that situation and one where the readings are 66, 64, and 59 using just the simple average? You can't, because both series have the same average. However, there's an evident diversity between the two cases—in the first, the readings are increasing, in the latter they are decreasing but the simple average cannot separate them. In this case, you need a *weighted* average, that is, an average where the single values get multiplied by a factor that represents their importance.

The general formula is:

$$A = (V_1 \times W_1 + V_2 \times W_2 + \dots + V_n \times W_n) \;/\; (W_1 + W_2 + \dots + W_n)$$

Suppose you want to give a weight of 1 to the oldest of three readings, 2 to the middle, and 4 to the latest one. Let's apply the formula to the series of our example:

$$(60 \times 1 + 62 \times 2 + 67 \times 4) / (1 + 2 + 4) = 64.57$$
$$(66 \times 1 + 64 \times 2 + 59 \times 4) / (1 + 2 + 4) = 61.43$$

You notice that the results are very different in the two cases: The weighted average reflects the trend of the data. For this reason, weighted averages seem ideal in many cases. They allow you to balance multiple readings, at the same time taking more recent ones into greater consideration. Unfortunately, they are memory- and time-consuming when computed by a program, especially when you want to use a large number of values.

Now, there is a particular class of weighted averages that can be of help, providing a simple and efficient way of storing historical readings and calculating new values. They rely on a method called *exponential smoothing*. (Don't let the name frighten you!)

The trick is simple: You take the new reading and the previous average value, and combine these into a new average value using two weights that together represent 100 percent. For example, you can take 40 percent of the new reading and 60 percent of the previous average, or instead take only 10 percent of the new reading and 90 percent of the previous average. The less weight you put on the new value, the more stable and slow to change the average will be.

The general equation for exponential smoothing is expressed as follows:

$$A_n = (V_n \times W_1 + A_{n-1} \times W_2) \ / \ (W_1 + W_2)$$

You can choose W_1 and W_2 to add up to 100, so that you can easily read them as a percentage. For example:

$$A_n = (V_n \times 20 + A_{n-1} \times 80) \ / \ 100$$

Let's apply this formula to the series of the previous example. The first number in the first series was 60. There is no previous value for the average, so we simply take this number:

$$A_1 = 60$$

When the next reading (62) arrives, you compute a new value for the average using the whole formula:

$$A_2 = (62 \times 20 + 60 \times 80) / 100 = 60.4$$

Then you apply the rule again for the third value:

$$A_3 = (67 \times 20 + 60.4 \times 80) \ / \ 100 = 61.72$$

The result tells you that the average is *slowly* acknowledging the trend in the data. This happens because the last reading counts only for 20 percent, while 80 percent comes from the previous value. If you want to make your average more reactive to recent values, you must increase the weight of the last factor. Let's see what happens by changing the 20 percent to 60 percent:

$$A_1 = 60$$
$$A_2 = (62 \times 60 + 60 \times 40) / 100 = 61.2$$
$$A_3 = (67 \times 60 + 61.2 \times 40) / 100 = 64.68$$

You notice that the formula is still smoothing the values, but gives much more importance to recent values. One of the advantages of exponential smoothing is that it is very easy to implement. The following is an example of NQC code:

```
int ave;

// initialize the average
ave = SENSOR_1;

// compute average
while (true)
{
  ave = (SENSOR_1 * 20 + ave * 80) / 100;
  // other instructions...
}
```

Simple, isn't it? You could be tempted to *reduce* the mathematical expression, but be careful, remember what we said about multiplying and dividing integer numbers. These are okay:

```
  ave = (SENSOR_1 * 2 + ave * 8) / 10;
  ave = (SENSOR_1 + ave * 4) / 5;
```

But this, though mathematically equivalent, leads to a worse approximation:

```
  ave = SENSOR_1 / 5 + ave * 4 / 5;
```

Designing & Planning…

Exponential Smoothing

Those of you with a gift for math might be interested in understanding where exponential smoothing got its name. Let's try to analyze the equation:

$A_n = (V_n \times W_1 + A_{n-1} \times W_2) \ / \ (W_1 + W_2)$

We can rewrite the weights W1 and W2 as fractions: $w_1 = W_1 / (W_1 + W_2)$ and $w_2 = W_2 / (W_1 + W_2)$, where w_1 and w_2 result in the range of 0 to 1. As $w_1 + w_2 = 1$, we can substitute w_2 with $(1 – w_1)$. Our equation then becomes:

$A_n = V_n \times w_1 + A_{n-1} \times (1 – w_1)$

Continued

Expanding the term A_{n-1} we get:

$$A_{n-1} = V_{n-1} \times w_1 + A_{n-2} \times (1 - w_1)$$

and substituting in the previous:

$$A_n = V_n \times w_1 + V_{n-1} \times w_1 \times (1 - w_1) + A_{n-2} \times (1 - w_1)^2$$

Continuing this expansion, we get the general form:

$$A_n = V_n \times w_1 + V_{n-1} \times w_1 \ (1 - w_1) + V_{n-2} \times w_1 \times (1 - w_1)^2$$
$$+ \dots + V_{n-m} \times w_1 \times (1 - w_1)^m + \dots + V_1 \times w_1 \times (1 - w_1)^n$$

This average is thus equivalent to an average of all the values, where the older they are the more they get smoothed by the exponential term $(1 - w_1)^m$. The term $(1 - w_1)$ being less than zero, means the higher the exponent, the smaller the result.

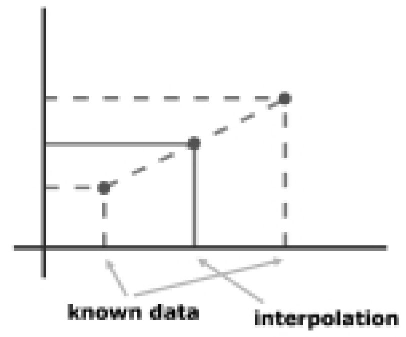
Using Interpolation

You've built a custom temperature sensor that returns a raw value of 200 at 0°C and 450 at 50°C. What temperature corresponds to a raw value of 315? Your robotic crane takes 10 seconds to lift a load of 100g, and 13 seconds for 200g. How long will it take to lift 180g? To answer these and other similar questions, you would turn to *interpolation*, a class of mathematical tools designed to estimate values from known data.

Interpolation has a simple geometric interpretation: If you plot your known data as points on a graph, you can draw a line or a curve that connects them. You then use the points on the line to guess the values that fall inside your data. There are many kinds of interpolation, that is, you can use many different equations corresponding to any possible mathematical curve to interpolate your data. The simplest and most commonly used one is *linear interpolation*, for which you connect two points with a straight line, and this is what we are going to explain here (Figure 12.3).

Please be aware that many physical systems don't follow a linear behavior, so linear interpolation will not be the best choice for all situations. However, linear interpolation is usually fine even for nonlinear systems, if you can break the ranges down into almost linear sections.

Figure 12.3 Linear Interpolation



In following standard terminology, we will call the parameter we change the *independent variable*, and the one that results from the value of the first, the *dependent variable*. With a very traditional choice, we will use the letter X for the first and Y for the second. The general equation for linear interpolation is:

$$(Y - Y_a) / (Y_b - Y_a) = (X - X_a) / (X_b - X_a)$$

Where Y_a is the value of Y we measured for $X = X_a$ and Y_b the one for X_b. With some simple work we can isolate the Y and transform the previous equation into:

$$Y = (X - X_a) \times (Y_b - Y_a) / (X_b - X_a) + Y_a$$

This is very simple to use, and allows you to answer your question about the custom temperature sensor. The raw value is your independent variable X, the one you know. The terms of the problem are:

$X_a = 200$ $Y_a = 0$

$X_b = 450$ $Y_b = 50$

$X = 315$ $Y = ?$

We apply the formula and get:

$$Y = (315 - 200) \times (50 - 0) / (450 - 200) + 0 = 23$$

To make our formula a bit more practical to use, we can transform it again. We define:

$$m = (Y_b - Y_a) / (X_b - X_a)$$

If you are familiar with college math, you will recognize in **m** the slope of the straight line that connects two points. Now our equation becomes:

$Y = m \times X - m \times X_a + Y_a$

As s, X_a and Y_a are all known constants, we compute a new term *b* as:

$b = Y_a - m \times X_a$

so our final equation becomes:

$Y = m \times X + b$

This is the standard equation of a straight line in the Cartesian plane. Looking back to our previous example, you can now compute **m** and **b** for your temperature sensor:

$m = (50 - 0) / (450 - 200) = 0.2$

$b = 0 - 0.2 \times 200 = -40$

$Y = 0.2 \times X - 40$

You can confirm your previous result:

$Y = 0.2 \times 315 - 40 = 23$

Implementing this equation inside a program for the RCX will require that you convert the decimal value 0.2 into a multiplication and a division, this way:

```
temp = (raw * 2) / 10 - 40;
```

Interpolation is also a good tool when you want to relocate the output from a system in a different range of values. This is what the RCX firmware does when converting raw values from the light sensor into a percentage (see Chapter 4). You can do the same in your application. Suppose you want to change the way raw values from the light sensor get converted into a percentage. The LEGO firmware defines that 1022 converts to 0 percent and 322 to 100 percent, but this range is quite wide with regard to the readings you actually experience with the light sensor. Let's say you want to fix an arbitrary range of 900 converting to 0 percent and 500 converting to 100 percent, and this is what you get from the interpolation formula:

$m = (0 - 100) / (900 - 500) = -0.25$

$b = 100 + 0.25 \times 500 = 225$

$Y= -0.25 \times X+ 225$

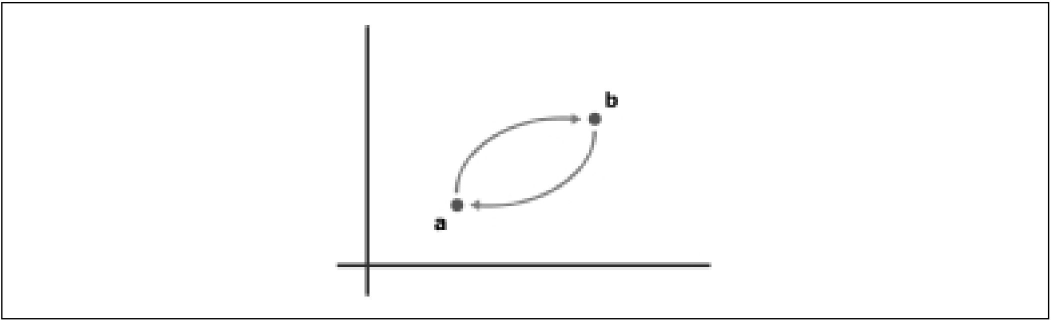
Multiplying by 0.25 is like dividing by 4, so we can write this expression in code as:

```
perc = - raw / 4 + 225;
```

Understanding Hysteresis

Hysteresis is actually more a physical than a mathematical concept. We say that a system has some hysteresis when it follows a path to go from state A to state B, and a different path when going *back* from state B to state A. Graphing the state of the system on a chart shows two different curves that connect the points A and B, one for going out and one for coming back (Figure 12.4).

Figure 12.4 Hysteresis in Physical Systems



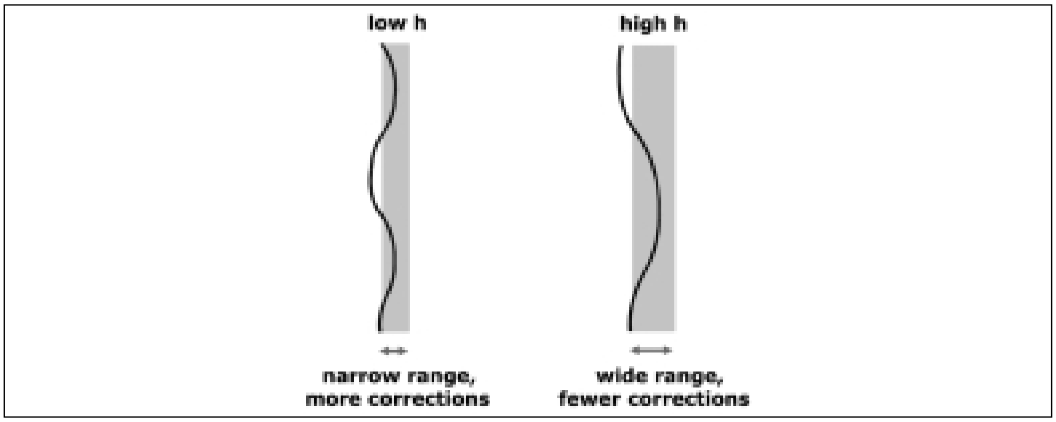
Hysteresis is a common property of many natural phenomena, magnetism above all, but our interest here is in introducing some hysteresis in our robotic programs. Why should you do it? First of all, let us say this is quite a common practice. In fact, many automation devices based on some kind of feedback have been equipped with artificial hysteresis.

A very handy example comes from the thermostat that controls the heating in your house. Imagine your heating system relies on a thermostat designed to maintain an exact temperature, and that during a cold winter you program your desired home temperature to 21°C (70°F). As soon as the ambient temperature goes below 21°C, the heating starts. In a few minutes the temperature reaches 21°C and heating stops, then a few minutes later starts again and so on all day long. The heater would turn off and on constantly as the temperature varies around that exact point. This approach is not the best one, because every start phase requires some time to bring the system to its maximum efficiency, just about when it gets stopped again. In introducing some hysteresis, the system can run smoother: We can let the temperature go down to 20.5°C, then heat up the house until it reaches 21.5°C. When the temperature in the house is 21°C, the heating can be either on or off, depending whether it's going from on to off or vice versa.

Hysteresis can reduce the number of corrective actions a system has to take, thus improving stability and efficiency at the price of some tolerance. Auto-pilots for boats and planes are another good example. Could you think of a task for your robots that could benefit from hysteresis? Line following is a good example.

In Chapter 4, in talking about light sensors, we explained that the best way to follow a strip on the floor is to actually follow one of its edges, the borderline between white and black. In that area, your sensor will read a **gray** value, some intermediate number between the white and black readings. Having chosen that value for **gray**, a robot with no hysteresis may correct left when the reading is greater than **gray** and right when reading is less than **gray**. To introduce some hysteresis, you can tell your robot to turn left when reading **gray+h** and right when reading **gray-h**, where *h* is a constant properly chosen to optimize the performance of your robot. There isn't a general rule valid for any system, you must find the optimal value for *h* by experimenting. Start with a value of about 1/6 or 1/8 of the total white-black difference; this way the interval **gray-h** to **gray+h** will cover 1/3 or 1/4 of the total range. Then start increasing or decreasing its value observing what happens to your robot, until you are happy with its behavior. You will discover that by reducing **h** your robot will narrow the range of its oscillations, but will perform more frequent corrections. Increasing **h**, on the other hand, will make your robot perform less corrections but with oscillations of larger amplitude (Figure 12.5).

Figure 12.5 How Hysteresis Affects Line Following



We suggest a simple experiment that will help you put these concepts into practice by building a real sensor setup that you can manipulate by hand to get a feeling of how the robot would behave. Write a simple program that plays tones to ask you to turn left or right. For example, it can beep high when you have to turn left and low to turn right. The NQC code that follows shows a possible implementation:

```
#define GRAY 50
#define H 3

task main()
{

  SetSensor(SENSOR_1,SENSOR_LIGHT);

  while (true)
  {
    if (SENSOR_1>GRAY+H)
      PlayTone(440,20);
    else if (SENSOR_1<GRAY-H)
      PlayTone(1760,20);
    Wait(20);
  }
}
```

Equip your RCX with a light sensor attached to input port 1 and you are ready to go. You should tune the value of the *GRAY* constant to what your sensor actually reads when placed exactly over the borderline between the white and the black. When the program runs, you can move the sensor toward or away from the line until you hear the beep that asks you to correct the direction. (Keep the sensor always at the same distance from the pad.) Experiment with different values for H to see how the accepted range of readings gets wider or narrower.

If you keep a pencil in your hand together with the light sensor, you can even perform this experiment blindfolded! Try to follow the line by just listening to the instructions coming from your RCX, and compare the lines drawn by your pencil for different values of H.

Summary

Math is the kind of subject that people either love or hate. If you fall in the latter group, we can't blame you for having skipped most of the content of this chapter. Don't worry, there was nothing you can't live without, just make an effort to understand the part about multiplication and divisions, because if you ignore the possible side effects, you could end up with some bad surprises in your calculations.

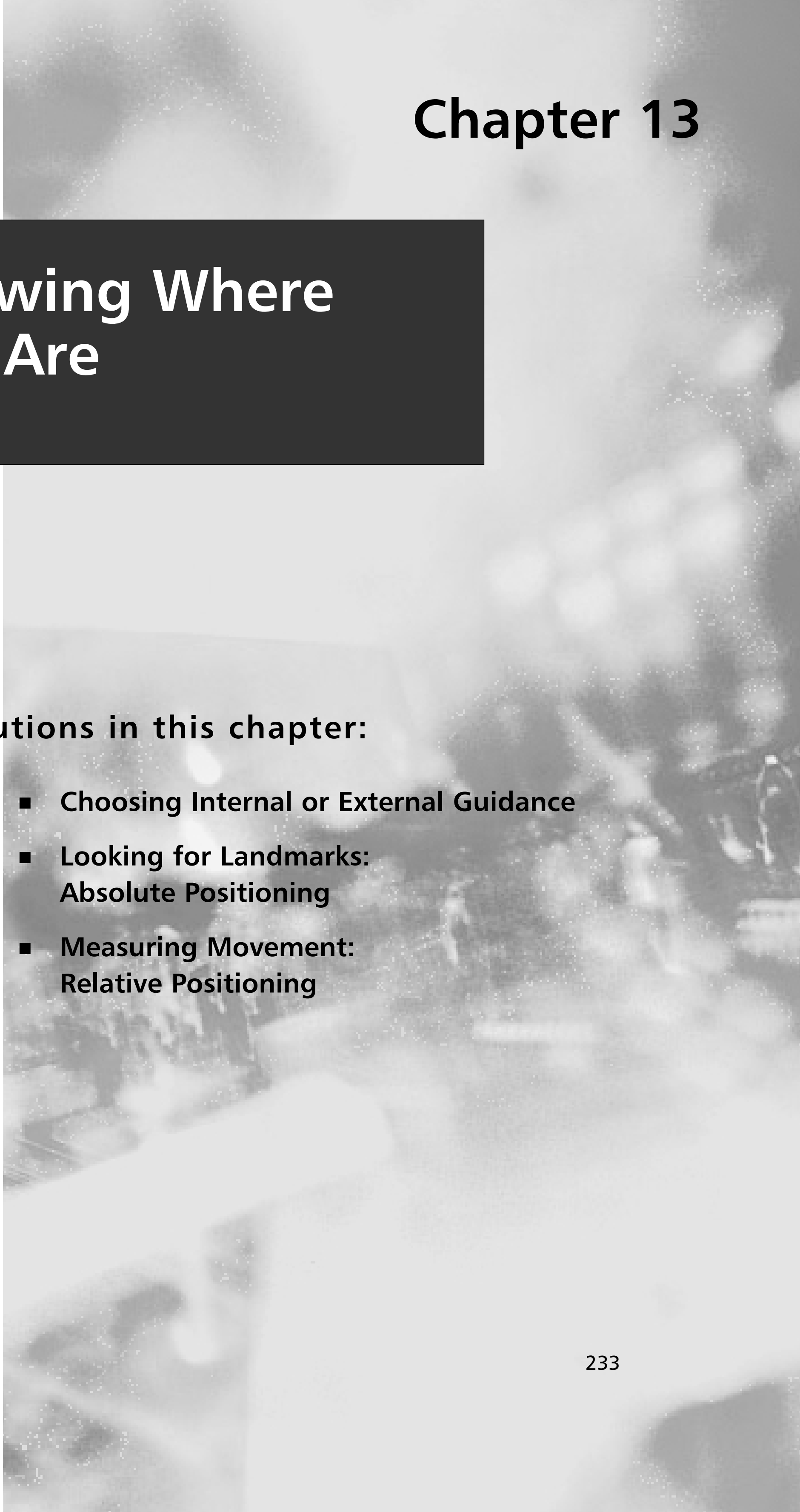
Consider the other topics—averages, interpolation, hysteresis—to be like tools in your toolbox. Averages are a useful instrument to soften the differences between single readings and to ignore temporary peaks. They allow you to group a set of readings and consider it as a single value. When you are dealing with a flow of data coming from a sensor, the moving average is the right tool to process the last **n** readings. The larger **n** is, the more the smoothing effect on the data.

Weighted averages have an advantage over simple averages in that they can show the trend in the data: You can assign the weights to put more importance on more recent data. Exponential smoothing is a special case of weighted averages, the results of which are particularly convenient on the implementation side, because they allow you to write compact and efficient code.

The interpolation technique proves useful when you want to estimate the value of a quantity that falls between two known limits. We described linear interpolation, which corresponds to tracing a straight line across two points in a graph. You then can use that line to calculate any value in the interval.

Hysteresis, a concept borrowed from physics, will help you in reducing the number of corrections your robots have to make to keep within a required behavior. By adding some hysteresis to your algorithms, your robots will be less reactive to changes. Hysteresis can also increase the efficiency of your system.

It's not necessary that you remember all the equations, just what they're useful for! You can always refer back to this chapter when you find a problem that might benefit from these mathematical tools.



Chapter 13

Knowing Where You Are

Solutions in this chapter:

- Choosing Internal or External Guidance
- Looking for Landmarks: Absolute Positioning
- Measuring Movement: Relative Positioning

Introduction

After our first few months of experimenting with robotics using the MINDSTORMS kit, we began to wonder if there was a simple way to make our robot know where it was and where it was going—in other words, we wanted to create some kind of navigation system able to establish its position and direction. We started reading books and searching the Internet, and discovered that this is still one of most demanding tasks in robotics and that there really isn't any single or simple solution.

In this chapter, we will introduce you to concepts of navigation, which can get very complex. We will start describing how positioning methods can be categorized into two general classes: *absolute* and *relative* positioning, the first based on external reference points, and the latter on internal measurements. Then we will provide some examples for both the categories, showing solutions and tricks that suit the possibilities of the MINDSTORMS system. In discussing absolute positioning, we will introduce you to navigation on pads equipped with grids or gradients, and to the use of laser beams to locate your robot in a room. As for relative positioning, we will explain how to equip your robot for the proper measurements, and will provide the math to convert those measurements into coordinates.

Choosing Internal or External Guidance

As we mentioned, there is no single method for determining the position and orientation of a robot, but you can combine several different techniques to get useful and reliable results. All these techniques can be classified into two general categories: *absolute* and *relative* positioning methods. This classification refers to whether the robot looks to the surrounding environment for tracking progress, or just to its own course of movement.

Absolute positioning refers to the robot using some external reference point to figure out its own position. These can be landmarks in the environment, either natural landmarks recognized through some kind of artificial vision, or more often, artificial landmarks easily identified by your robot (such as colored tape on the floor). Another common approach includes using radio (or light) beacons as landmarks, like the systems used by planes and ships to find the route under any weather condition. Absolute positioning requires a lot of effort: You need a prepared environment, or some special equipment, or both.

Relative positioning, on the other hand, doesn't require the robot to know anything about the environment. It deduces its position from its previous (known) position and the movements it made since the last known position. This is usually achieved through the use of encoders that precisely monitor the turns of the wheels, but there are also inertial systems that measure changes in speed and direction. This method is also called *dead reckoning* (short for *deduced reckoning)*.

Relative positioning is quite simple to implement, and applies to our LEGO robots, too. Unfortunately, it has an intrinsic, unavoidable problem that makes it impossible to use by itself: It accumulates errors. Even if you put all possible care into calibrating your system, there will always be some very small difference due to slippage, load, or tire deformation that will introduce errors into your measurements. These errors accumulate very quickly, thus relegating the utility of relative positioning to very short movements. Imagine you have to measure the length of a table using a very short ruler: You have to put it down several times, every time starting from the point where you ended the previous measurement. Every placement of the ruler introduces a small error, and the final result is usually very different from the real length of the table.

The solution employed by ships and planes, which use beacons like Loran or Global Positioning Systems (GPS) systems, and more recently by the automotive industry, is to combine methods from the two groups: to use dead reckoning to continuously monitor movements and, from time to time, some kind of absolute positioning to zero the accumulated error and restart computations from a known location. This is essentially what human beings do: When you walk down a street while talking to a friend, you don't look around continuously to find reference points and evaluate your position; instead, you walk a few steps looking at your friend, then back to the street for an instant to get your bearings and make sure you haven't veered off course, then you look back to your friend again.

You're even able to safely move a few steps in a room with your eyes shut, because you can deduce your position from your last known one. But if you walk for more than a few steps without seeing or touching any familiar object, you will soon lose your orientation.

In the rest of the chapter, we will explore some methods for implementing absolute and relative positioning in LEGO robots. It's up to you to decide whether or not to use any one of them or a combination in your applications. Either way, you will discover that this undertaking is quite a challenge!

Looking for Landmarks: Absolute Positioning

The most convenient way to place artificial landmarks is to put them flat on the floor, since they won't obstruct the mobility of your robot and it can read them with a light sensor without any strong interference from ambient light. You can stick some self adhesive tape directly on the floor of your room, or use a sheet of cardboard or other material over which you make your robot navigate.

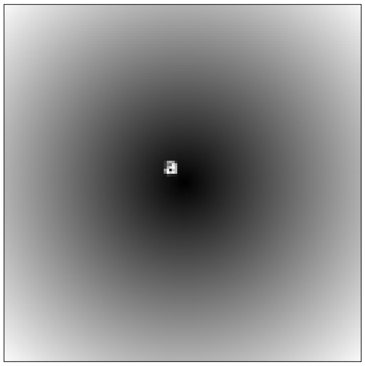
Line following, which we have talked a lot about, is probably the simplest example of navigation based on using an artificial landmark. In the case of line following, your robot knows nothing about where it is, because its knowledge is based solely on whether it is to the right or left of the line. But lines are indeed an effective system to steer a robot from one place to another. Feel free to experiment with line following; for example, create some interruptions in a straight line and see if you are able to program your robot to find the line again after the break. It isn't easy. When the line ends, a simple line follower would turn around and go back to the other side of the line. You have to make your software more sophisticated to detect the sudden change and, instead of applying a standard route correction, start a new searching algorithm that drives the robot toward a piece of line further on. Your robot will have to go forward for a specific distance (or time) corresponding to the approximate length of the break, then turn left and right a bit to find the line again and resume standard navigation.

When you're done and satisfied with the result, you can make the task even more challenging. Place a second line parallel to the first, with the same interruptions, and see if you can program the robot to turn 90 degrees, intercept the second line, and follow that one. If you succeed in the task, you're ready to navigate a grid of short segments, either following along the lines or crossing over them like a bar code.

You can improve your robot navigation capabilities, and reduce the complexity in the software, using more elaborate markers. As we explained in Chapter 4, the LEGO light sensor is not very good at distinguishing different colors, but is able to distinguish between differences in the intensity of the reflected light. You can play with black and gray tapes on a white pad, and use their color as a source of information for the robot. Remember that a reading at the border between black and white can return the same value of another on plain gray. Move and turn your robot a bit to decode the situation properly, or employ more than a single light sensor if you have them.

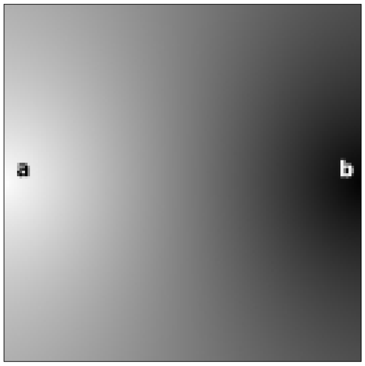
Instead of placing marks on the pad, you can also print on it with a special black and white gradient. For example, you can print a series of dots with an intensity proportional to their distance from a given point a. The closer to a, the darker the point; a is plain black (see Figure 13.1). On such a pad, your robot will be able to return to a from any point, by simply following the route until it reads the minimum intensity.

Figure 13.1 A Gradient Pad with a Single Attractor



The same approach can be used with two points a and b, one being white and the other black. Searching for the whitest route, the robot arrives at a, while following the darkest it goes to b (Figure 13.2). We first saw this trick applied during the 1999 Mindfest gathering at the Massachusetts Institute of Technology (MIT): two robots were playing soccer, searching for a special infrared (IR) emitting ball. When one got the ball, it used the pad to find the proper white or black goal. Months later, we successfully replicated this setup with Marco Berti during a demonstration at an exhibition in Italy.

Figure 13.2 A Gradient Pad with Two Attractors



There are other possibilities. People have suggested using bar codes on the floor: When the robot finds one, it aligns and reads it, decoding its position from the value. Others tried complex grids made out of stripes of different colors. Unfortunately, there isn't a simple solution valid for all cases, and you will very likely be forced to use some dead reckoning after a landmark to improve the search.

Designing & Planning…

Making a Gradient Pad

To print a gradient pad with a single attractor A simply make the darkness (or brightness) of any point proportional to its distance from A. If *ax* and *ay* are the coordinates of A, and *x, y* are the coordinates of the given pixel, the distance will be:

```
dist = sqrt((x-ax)*(x-ax)+(y-ay)*(y-ay))
```

Scale the distance so as to have 100 percent black in A and 0 percent at the maximum distance from A measured in your pad, then multiply it by the constant that represents the maximum brightness of a pixel in your system:

```
intensity = dist/maxdist*maxbrite
```

Now apply this value to all three color components of a pixel using something similar to:

```
pixels[x,y] = rgb(intensity,intensity,intensity )
```

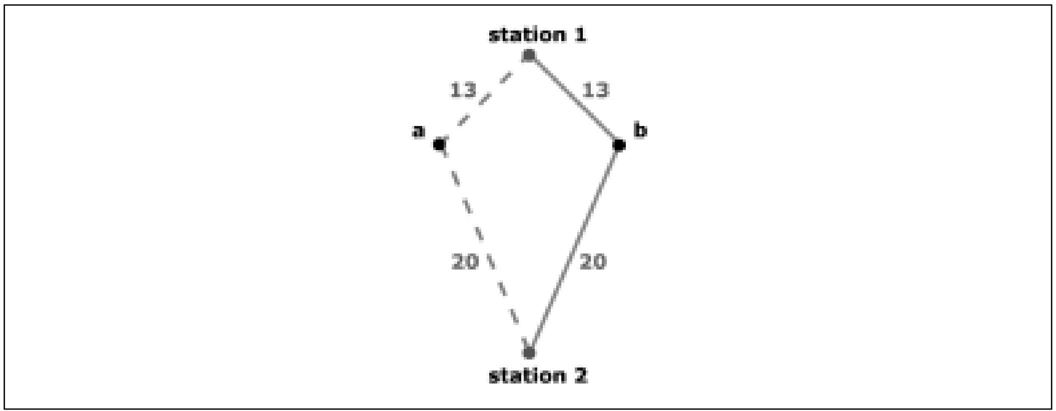
To generate a pad with two attractors A (*ax, ay*) and B (*bx, by*) respectively white and black, make the intensity of each pixel proportional to the ratio of the distance from B over the sum of the distances from A and from B:

```
adist = sqrt((x-ax)*(x-ax)+(y-ay)*(y-ay))
bdist = sqrt((x-bx)*(x-bx)+(y-by)*(y-by))
intensity = bdist/(adist+bdist)*maxbrite
pixels[x,y] = rgb(intensity,intensity,intensity)
```

Following the Beam

In the real world, most positioning systems rely on beacons of some kind, typically radio beacons. By using at least three beacons, you can determine your position on a two-dimensional plane, and with four or more beacons you can compute your position in a three-dimensional space. Generally speaking, there are two kinds of information a beacon can supply to a vehicle: its distance and its heading (direction of travel). Distances are computed using the amount of time that a radio pulse takes to go from the source to the receiver: the longer the delay, the larger the distance. This is the technique used in the Loran and the GPS systems. Figure 13.3 shows why two stations are not enough to determine position: There are always two locations A and B that have the same distance from the two sources.

Figure 13.3 There are Two Locations with the Same Distance from Two Stations



Adding a third station, the system can solve the ambiguity, provided that this third station does not lie along the line that connects the previous two stations (see Figure 13.4).

The stations of the VHF Omni-directional Range system (VOR) can not tell you the distance from the source of the beacon, but they do tell you their heading, that is, the direction of the route you should go to reach each station. Provided that you also know your heading to the North, two VOR stations are enough to locate your vehicle in most cases. Three of them are required to cover the case where the vehicle is positioned along the line that connects the stations, and as for the Loran and GPS systems, it's essential that the third station itself does not lay along that line (see Figure 13.5).

Figure 13.4 Three Nonaligned Stations Allow for Positioning with No Ambiguities Using Distances

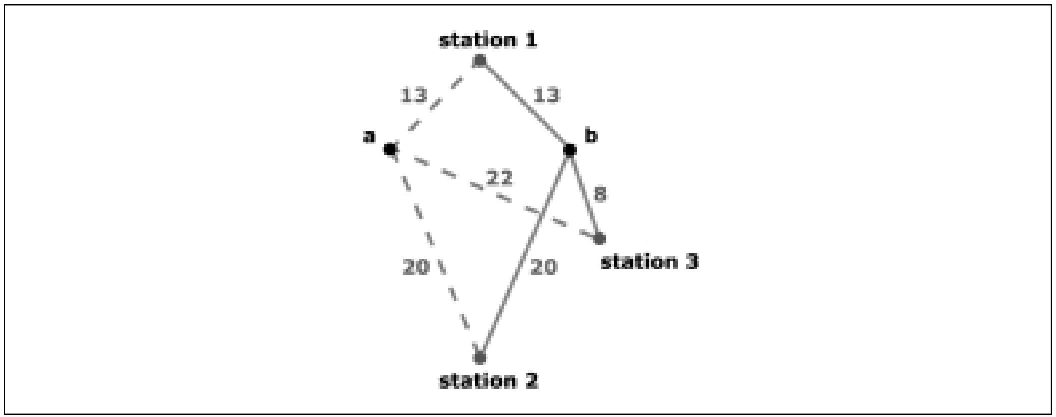
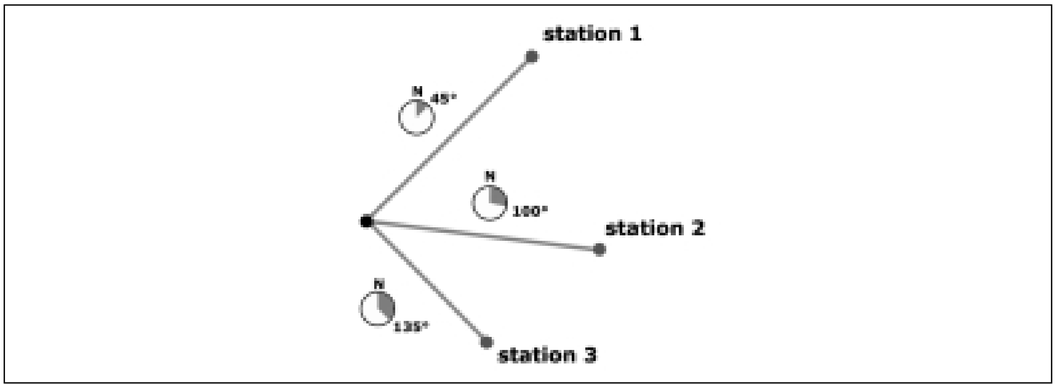


Figure 13.5 VOR-Like Systems Allow Positioning through the Headings of the Stations



Using three stations, you can do without a compass, that is, you don't need to know your heading to the North. The method requires that you know only the angles between the stations as you see them from your position (see Figure 13.6).

To understand how the method works, you can perform a simple experiment. Take a sheet of paper and mark three points on it that correspond to the three stations. Now take a sheet of transparent material and put it over the previous sheet. Spot a point anywhere on it which represents your vehicle, and draw three lines from it to the stations, extending the lines over the stations themselves. Mark the lines with the name of the corresponding stations. Now, move the transparent sheet and try to intersect the lines again with the stations. There's an unlimited number of positions which connect two of the three stations, but there's only one location which connects all three of them.

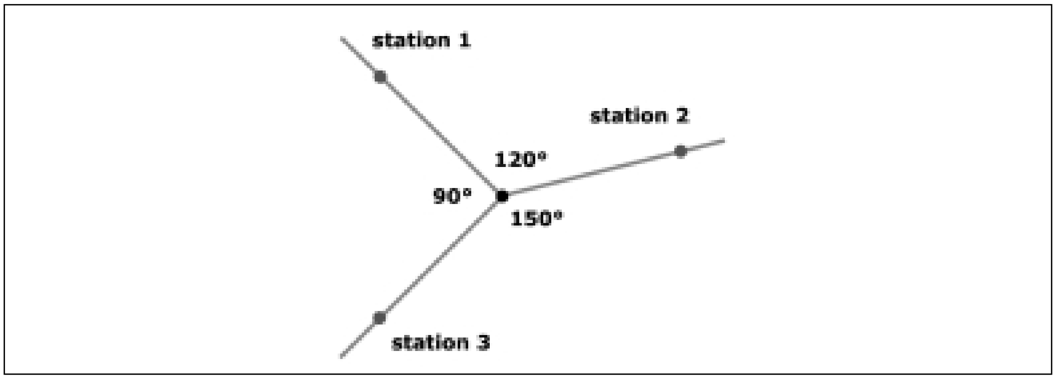


Figure 13.6 Three Nonaligned Stations Allow for Positioning with No Ambiguities Using Angles

If you want to give this approach a try, the first problem you have to solve is that there's currently nothing in the LEGO world that can emit beacons of any kind, so you have to look for some alternative device. Unless you're an electrical engineer and are able to design and build your own custom radio system, you better stick with something simple and easy to find. The source need not be necessarily based on radio waves—light is effective as well, and we already have such a detector (the light sensor) ready to interface to the RCX.

By using light sources as small lighthouses, you can, in theory, make your robot find its way. But there are a few difficulties to overcome first:

- The light sensor isn't directional—you must shield it somehow to narrow its angle.
- Ambient light introduces interference, so you must operate in an almost lightless room.
- For the robot to be able to tell the difference between the beacons, you must customize each one; for example, making them blink at different rates (as real lighthouses do).

Laser light is probably a better choice. It travels with minimum diffusion, so when it hits the light sensor, it is read at almost 100 percent. Laser sources are now very common and very cheap. You can find small battery-powered pen laser pointers for just a few dollars.

WARNING

Laser light, even at low levels, is very damaging to eyes—never direct it towards people or animals.

If you have chosen laser as a source of light, you don't need to worry about ambient light interference. But how exactly would you use laser? Maybe by making some rotating laser lighthouses? Too complex. Let's see what happens if we revert the problem and put the laser source *on the robot*. Now you need just one laser, and can rotate it to hit the different stations. So, the next hurdle is to figure out how you know when you have hit one of those stations. If you place an RCX with a light sensor in every station, you can make it send back a message when it gets hit by the laser beam, and using different messages for every station, make your robot capable of distinguishing one from another.

When we actually tried this, we discovered that the light sensor is a very small target to hit with a laser beam, and as a result, realized we had set an almost impossible goal. To stick with the concept but make things easier, we discovered you could build a sort of diffuser in front of it to have a wider detection area. Jonathan Brown suggested one made with a simple piece of paper, which worked very well.

Now you have a working solution, but it's a pity you need so many RCXs, at least three for the stations and one for your robot. Isn't there a cheaper option? A different approach involves employing the simple plastic reflectors used on cars, bikes, and as cat's-eyes on the side of the road. They have the property of reflecting any incoming light precisely back in the direction from which it came. Using those as passive stations, when your robot hits them with its laser beam they reflect it back to the robot, where you have placed a light sensor to detect it.

This really seems the perfect solution, but it actually still has its weak spots. First, you have lost the ability to distinguish one station from the other. You also have to rely on dead reckoning to estimate the heading of each station. We explained that dead reckoning is not very accurate and tends to accumulate errors, but it can indeed provide you with a good *approximation* of the expected heading of each station, enough to allow you to distinguish between them. After having received the actual readings, you will adjust the estimated heading to the measured one. The second flaw to the solution is that most of the returning beam tends to go straight back to the laser beam. You must be able to very closely align the light sensor to the laser source to intercept the return beam, and even with that precaution, detecting the returning beam is not very easy.

To add to these difficulties, there is some math involved in deducing the position of the robot from the beacons, and it's the kind of math whose very name sends shivers down most students spines: trigonometry! (Don't worry, you can find formulas in some of the resources referenced by Appendix A.) This leads to another problem: The standard firmware has no support for trig functions, and though in theory you could implement them in the language Not Quite C (NQC) using some tables and interpolation, the LEGO firmware does not give you enough variables to get useful results. If you want to proceed with using beacons, you really have to switch to leJOS or legOS, which both provide much more computational power.

If you're not in the mood to face the complexity of trigonometry and alternative firmware, you can experiment with simpler projects that still involve laser pointers and reflectors. For example, you can make a robot capable of "going home." Place the reflector at the home base of the robot, that is, the location where you want it to return. Program the robot to turn in place with the laser active, until the light beam intercepts the reflector and the robot receives the light back, then go straight in that direction, checking the laser target from time to time to adjust the heading.

Measuring Movement: Relative Positioning

Relative positioning, or dead reckoning, is based on the measurement of either the movements made by the vehicle or the force involved (acceleration). We'll leave this latter category alone, as it requires expensive inertial sensors and gyroscopic compasses that are not easy to interface to the RCX!

The technique of measuring the movement of the vehicle, called *odometry*, requires an *encoder* that translates the turn of the wheels into the corresponding traveled distance. Choosing among LEGO supplies, the obvious candidate is the rotation sensor. However, you already know from Chapter 4 that you can emulate the rotation sensor with a touch or a light sensor. You typically will need two of them, though by using some of the special platforms seen in Chapter 8 (the dual differential drive and synchro drive) you can implement odometry with a single rotation sensor. If you don't have any, look in Chapter 4 for some possible substitutes.

The equations for computing the position from the decoded movements depends on the architecture of the robot. We will explain it here using the example of the differential drive, once again referring you to Appendix A for further resources on the math used.

Suppose that your robot has two rotation sensors, each connected through gearing to one of the main wheels. Given D as the diameter of the wheel, R as the resolution of the encoder (the number of counts per turn), and G the gear ratio between the encoder and the wheel, you can obtain the conversion factor F that translates each unit from the rotation sensor into the corresponding traveled distance:

$$F = (D \times \pi) \ / \ (G \times R)$$

The numerator of the ratio, $D \times \pi$, expresses the circumference of the wheel, which corresponds to the distance that the wheel covers at each turn. The denominator of the ratio, $G \times R$, defines the increment in the count of the encoder (number of *ticks*) that corresponds to a turn of the wheel. F results in the unit distance covered for every tick.

Your robot uses the largest spoked wheels, which are 81.6mm in diameter. The rotation sensor has a resolution of 16 ticks per turn, and it is connected to the wheel with a 1:5 ratio (five turns of the sensor for one turn of the wheel). The resulting factor is:

$$F = 81.6 \text{ mm} \times 3.1416 \ / \ (5 \times 16 \text{ ticks}) \approx 3.2 \text{ mm/tick}$$

This means that every time the sensor counts one unit, the wheel has traveled 3.2mm. In any given interval of time, the distance T_L covered by the left wheel will correspond to the increment in the rotation sensor count I_L multiplied by the factor F:

$$T_L = I_L \times F$$

And similarly, for the right wheel:

$$T_R = I_R \times F$$

The centerpoint of the robot, the one that's in the middle of the ideal line that connects the drive wheels, has covered the distance T_C:

$$T_C = (T_R + T_L) \ / \ 2$$

To compute the change of orientation ΔO you need to know another parameter of your robot, the distance between the wheels B, or to be more precise, between the two points of the wheels that touch the ground.

$$\Delta O = (T_R - T_L) \ / \ B$$

This formula returns ΔO in radians. You can convert radians to degrees using the relationship:

$$\Delta O_{Degrees} = \Delta O_{Radians} \times 180 / \pi$$

You can now calculate the new relative heading of the robot, the new orientation O at time i based on previous orientation at time i – 1 and change of orientation ΔO. O is the direction your robot is pointed at, and results in the same unit (radians or degrees) you choose for ΔO.

$$O_i = O_{i-1} + \Delta O$$

Similarly, the new Cartesian coordinates of the centerpoint come from the previous ones incremented by the traveled distance:

$$x_i = x_{i-1} + T_C \times \cos O_i$$
$$y_i = y_{i-1} + T_C \times \sin O_i$$

The two trigonometric functions convert the vectored representation of the traveled distance into its Cartesian components.

O' this villainous trigonometry again! Unfortunately, you can't get rid of it when working with positioning. Thankfully, there are some special cases where you can avoid trig functions, however; for example, when you're able to make your robot turn in place precisely 90 degrees, and truly go straight when you expect it to. In this situation, either x or y remains constant, as well as the other increments (or decrements) of the traveled distance T_C. Thinking back to Chapter 8, two platforms make ideal candidates for this: the dual differential drive and the synchro drive.

Using a dual differential drive you need just one rotation sensor, attached to either the right or the left wheel. The mechanics guarantees that when one motor is on, the robot drives perfectly straight, while when the other is on, the robot turns in place. In the first case, the rotation sensor will measure the traveled distance T_C, while in the second, you must count its increments to turn exactly in multiples of 90 degrees. In Chapter 23, you will find a robot which turns this theory into a practical application, a dual differential drive that navigates using a single rotation sensor. We will go through the math again and will describe a complete NQC implementation of the formulas.

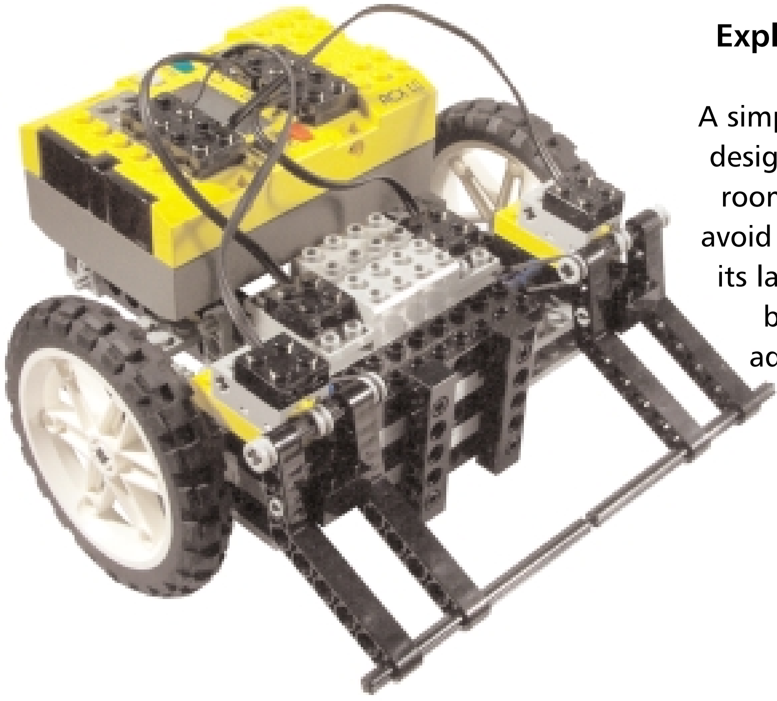
The synchro drive can be easily limited to turn its wheels only 90 degrees. With this physical constraint regarding change of direction, you can be sure it will proceed using only right angles. Connect the rotation sensor to the motion motor. Just as in the previous case you will use it to measure the traveled distance. In this setup, as with that using the dual differential drive, one rotation sensor is enough.

Summary

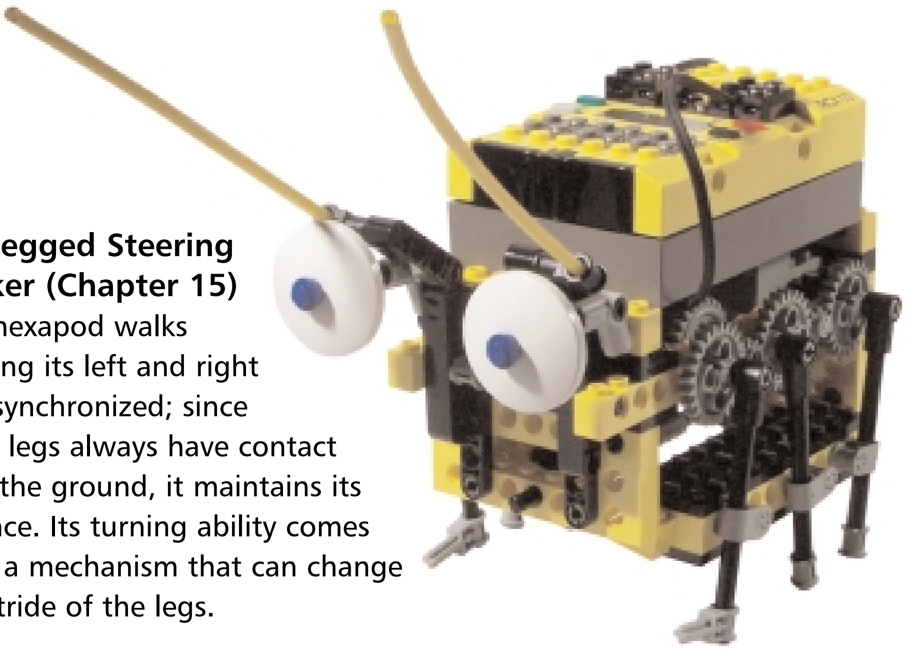
We promised in the introduction that this was a difficult topic, and it was. Nevertheless, making a robot that has even a rough estimate of its position is a rewarding experience.

There are two categories of methods for estimating position, one based on absolute positioning and the other on relative positioning. The first category usually requires artificial landmarks or beacons as external references, both of which we explored. Artificial landmarks can range from a simple line to follow, to a grid of marks, to more complex gradient pads. All of them allow a good control of navigation through the use of a light sensor. The MINDSTORMS line is not very well equipped for absolute positioning through the use of beacons; however, we described some possible approaches based on laser beams and introduced you to the difficulties that they entail.

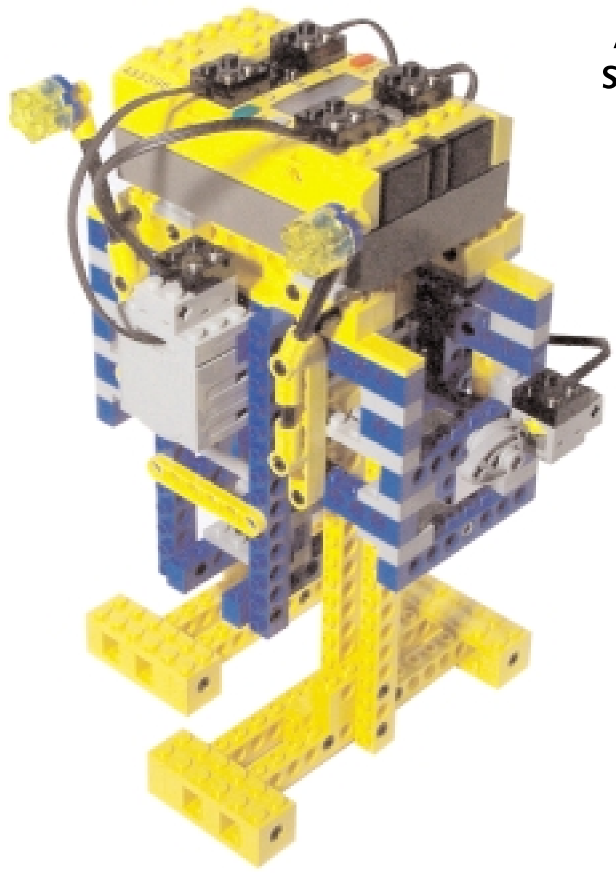
Relative positioning is more easily portable to LEGO robots. We explained the math required to implement deduced reckoning in a differential drive robot, and suggested some alternative architectures that help in simplifying the involved calculations. Unfortunately, relative positioning methods have an unavoidable tendency to accumulate errors; you can reduce the magnitude of the problem, but you cannot eliminate it. This is the reason that real life navigation systems—like those used in cars, planes, and ships—usually rely on a combination of methods taken from both categories: Dead reckoning isn't very computation intensive and provides good precision in a short range of distances, while absolute references are used to zero the errors it accumulates.



Exploring Your Room (Chapter 14)
A simple differential drive designed to explore your room and to detect and avoid any obstacles using its large bumpers. It can be equipped with an additional sensor that makes it capable of detecting drop-offs like stairways.

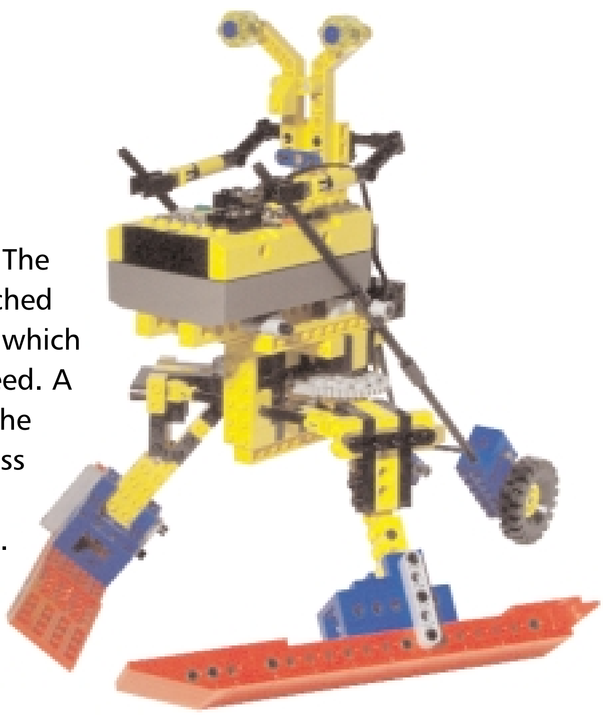


Six-Legged Steering Walker (Chapter 15)
This hexapod walks keeping its left and right legs synchronized; since three legs always have contact with the ground, it maintains its balance. Its turning ability comes from a mechanism that can change the stride of the legs.

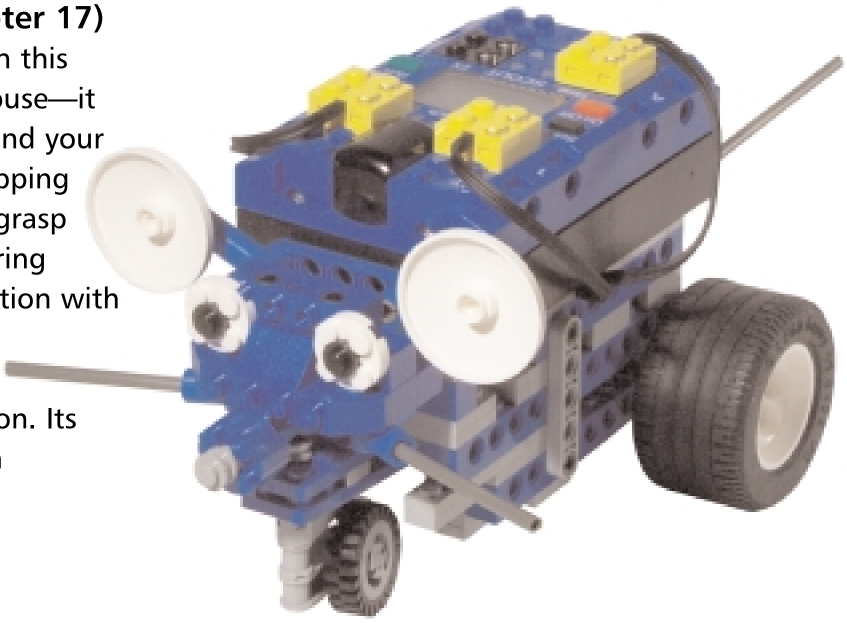


A Center of Gravity (COG)-Shifting Biped (Chapter 15)
In order to keep its balance, a biped robot needs to move its center of gravity over one foot before lifting the other. This two-legged walker has its RCX mounted on a sled, which slides over the supporting leg with each step.

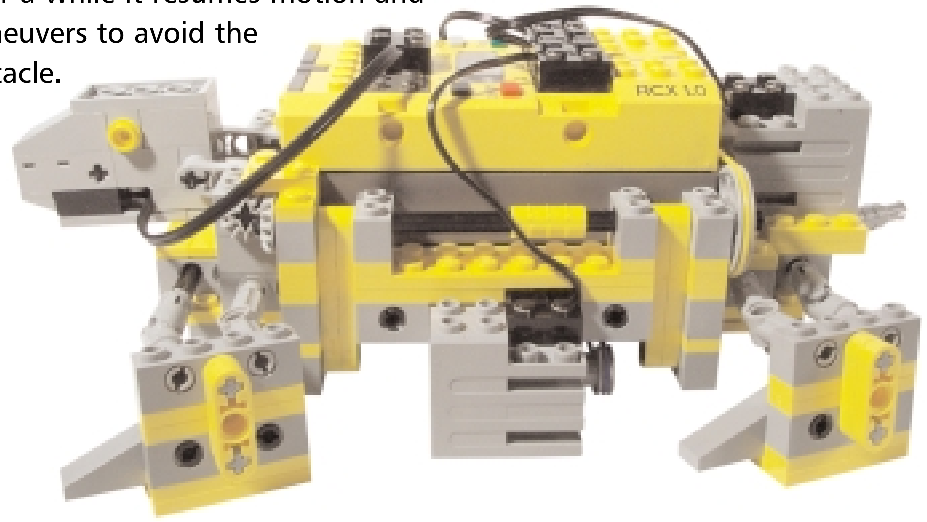
Skier (Chapter 16)
This simple robot is able to snowplow on a snowy slope. The left ski pole has a wheel attached to a rotation sensor, through which the robot can evaluate its speed. A motor on the back operates the legs, making them more or less convergent and thus keeping its speed in the desired range.



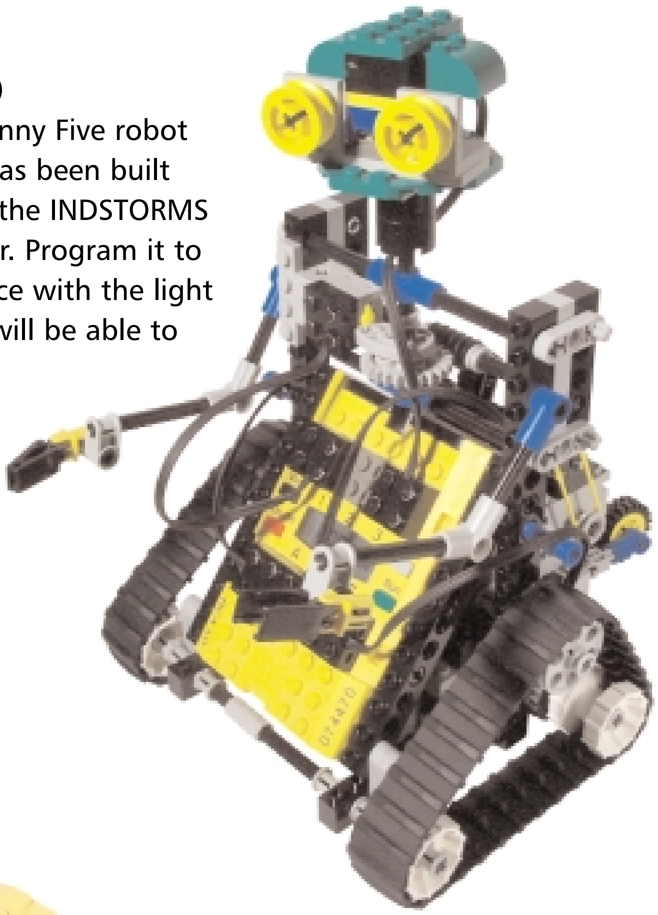
Mouse (Chapter 17)
Just try to catch this fast robotic mouse—it will speed around your room, only stopping (briefly) if you grasp its tail! Its steering drive configuration with a caster wheel gives it a fast responsive action. Its entire head is a bumper.

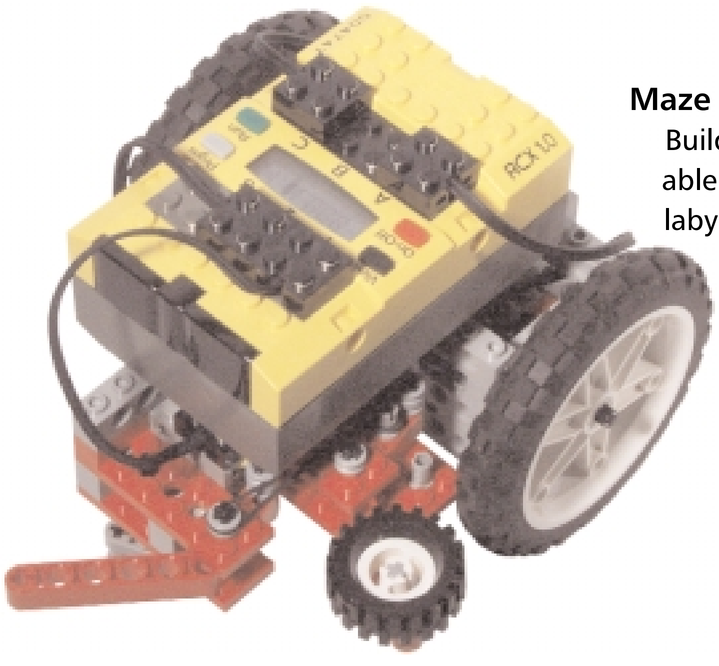


Turtle (Chapter 17)
This is a slow walking turtle with a sensor in its nose. When it detects an obstacle, it stops and retracts its head. After a while it resumes motion and maneuvers to avoid the obstacle.

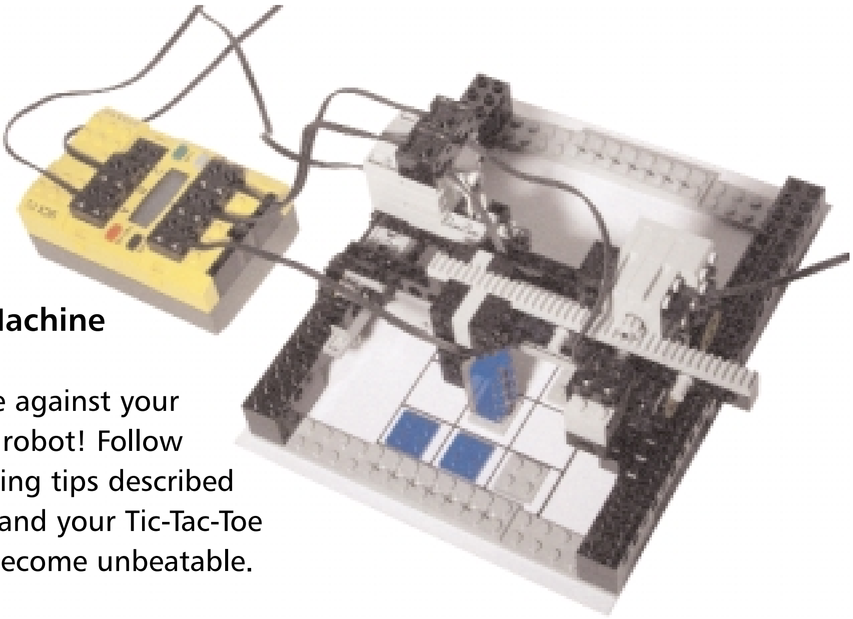


Johnny Five (Chapter 18)
This reproduction of the Johnny Five robot from the *Short Circuit* film has been built from the parts contained in the INDSTORMS kit plus one additional motor. Program it to find the strongest light source with the light sensor in its head, and you will be able to drive it with a flashlight.

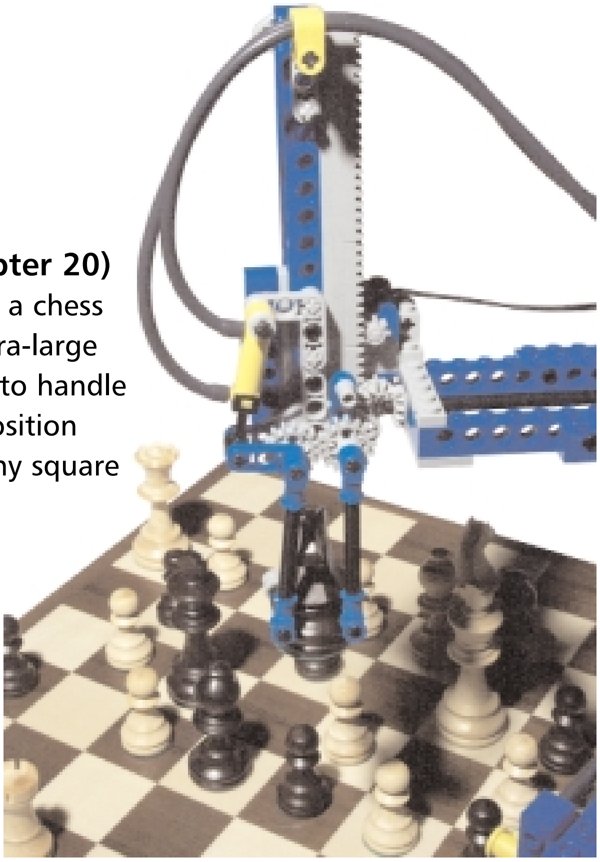




Maze Runner (Chapter 19)
Build a MINDSTORMS robot able to find its way out of a labyrinth. This Maze Runner has been designed to run through a maze following the left wall out to the exit.

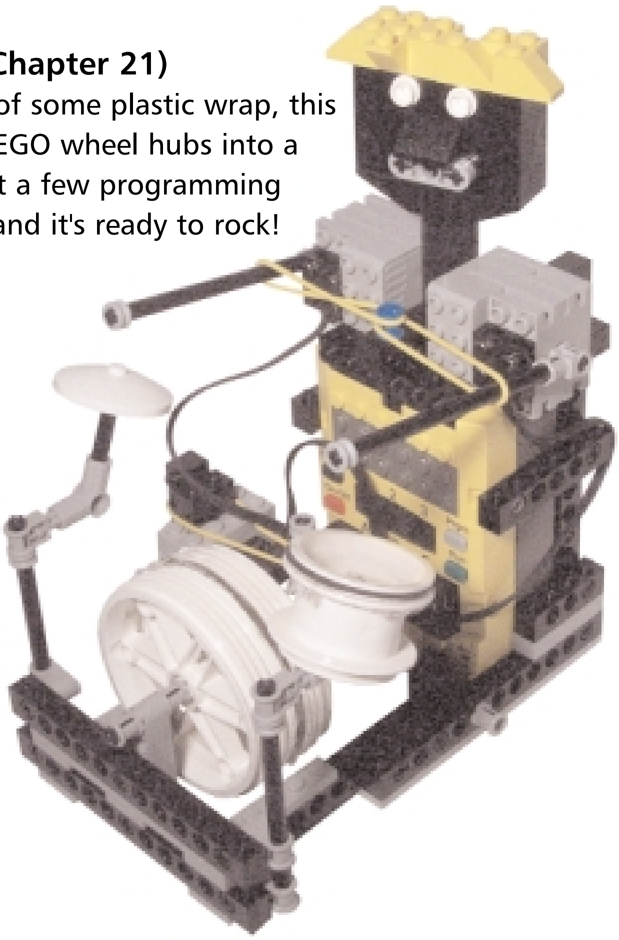


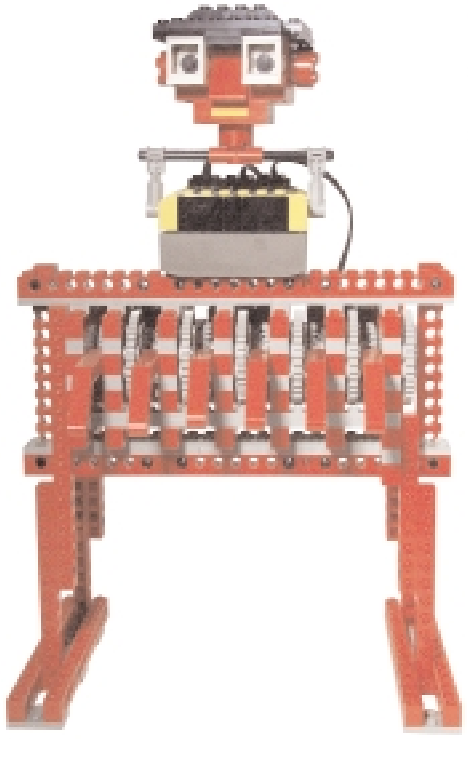
Tic-Tac-Toe Machine (Chapter 20)
Play Tic-Tac-Toe against your MINDSTORMS robot! Follow the programming tips described in Chapter 20 and your Tic-Tac-Toe machine will become unbeatable.



Broad Blue (Chapter 20)
Are you looking for a chess opponent? This extra-large robotic arm is able to handle chess pieces and position them precisely in any square of the chessboard.

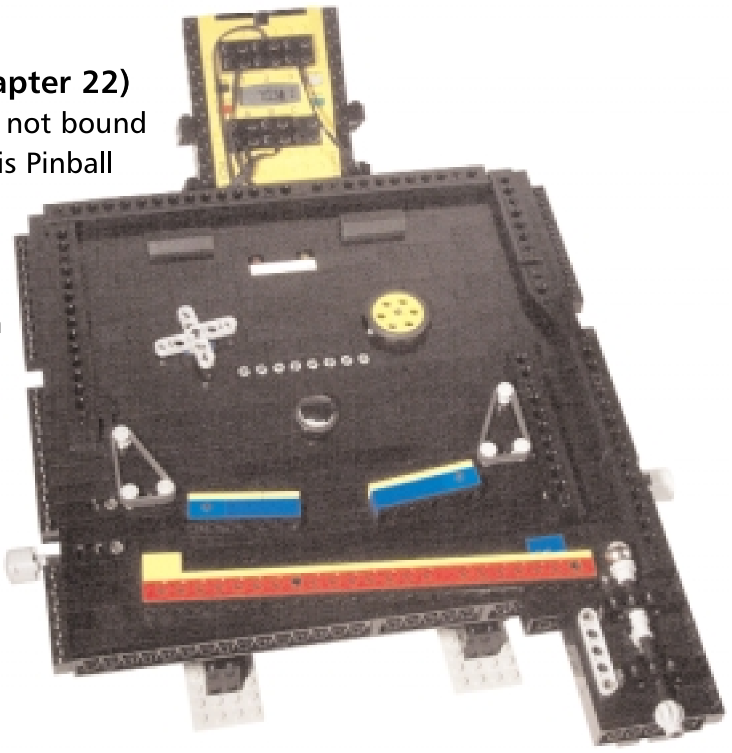
Drummer (Chapter 21)
With the aid of some plastic wrap, this robot turns LEGO wheel hubs into a drum set. Just a few programming instructions, and it's ready to rock!



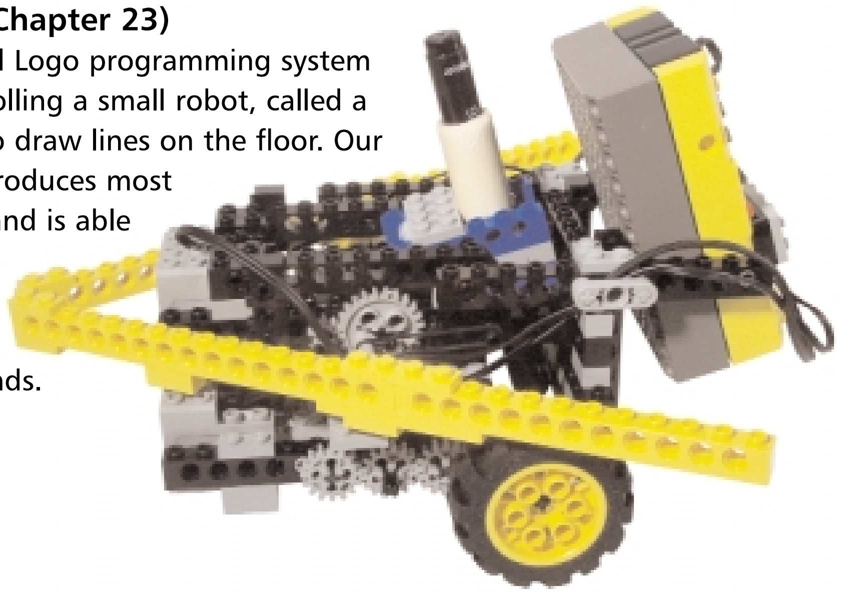


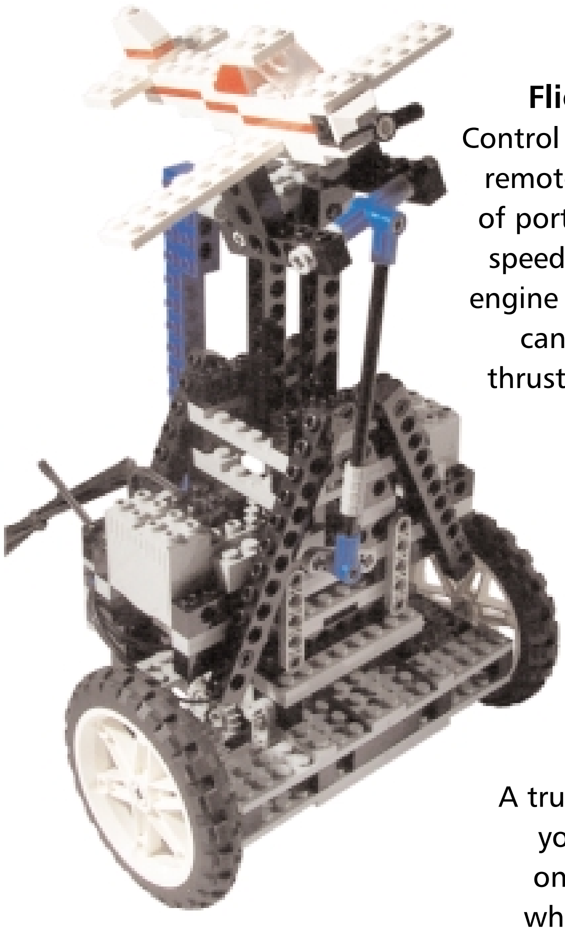
Piano Player (Chapter 21)
Our Piano Player has been designed to play on a real keyboard. It is capable of playing notes and chords on six consecutive keys.

Pinball Machine (Chapter 22)
The MINDSTORMS kit is not bound to build only robots. This Pinball machine demonstrates that there's plenty of opportunities for projects which—though not considered pure robotics—are indeed a lot of fun to imagine, build, and play with.



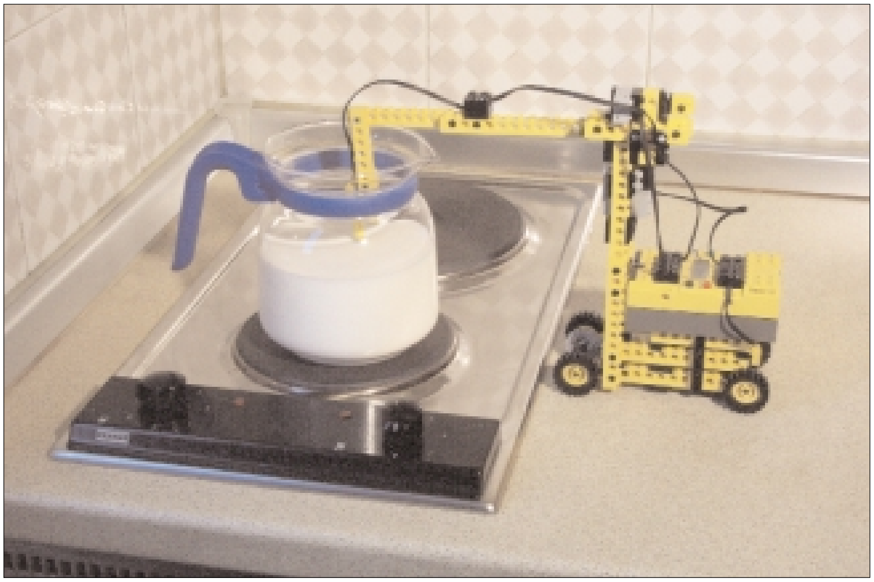
Logo Turtle (Chapter 23)
The educational Logo programming system aimed at controlling a small robot, called a turtle, is able to draw lines on the floor. Our Logo turtle reproduces most of its features and is able to draw precise geometric shapes from simple commands.





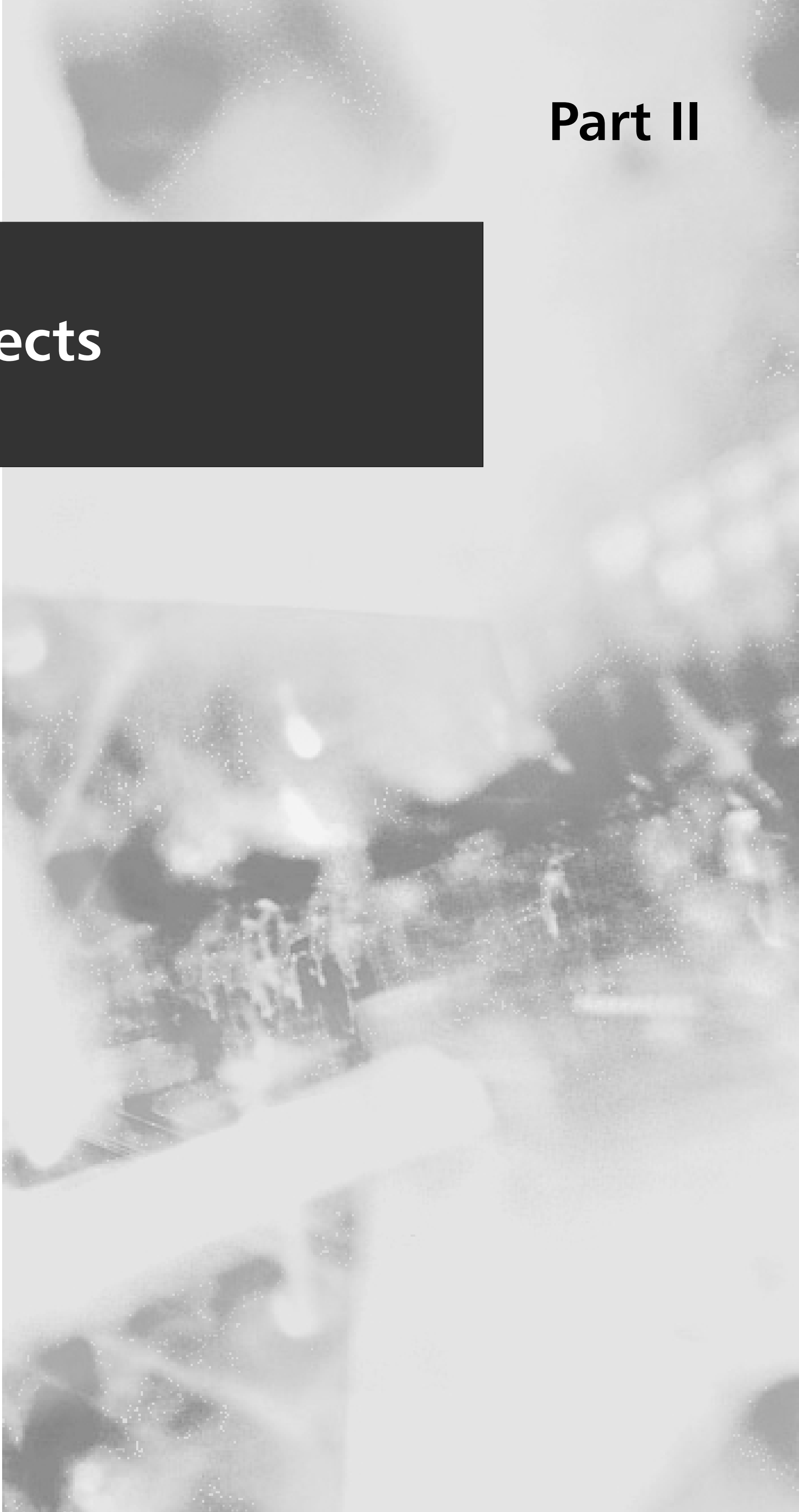
Flight Simulator (Chapter 24)
Control this Flight Simulator through a remote console that works like a sort of portable cockpit—the RCX displays speed and altitude and produces the engine sound—or a stall warning. You can experiment with the effects of thrust, lift, drag, acceleration, speed, heading, and altitude!

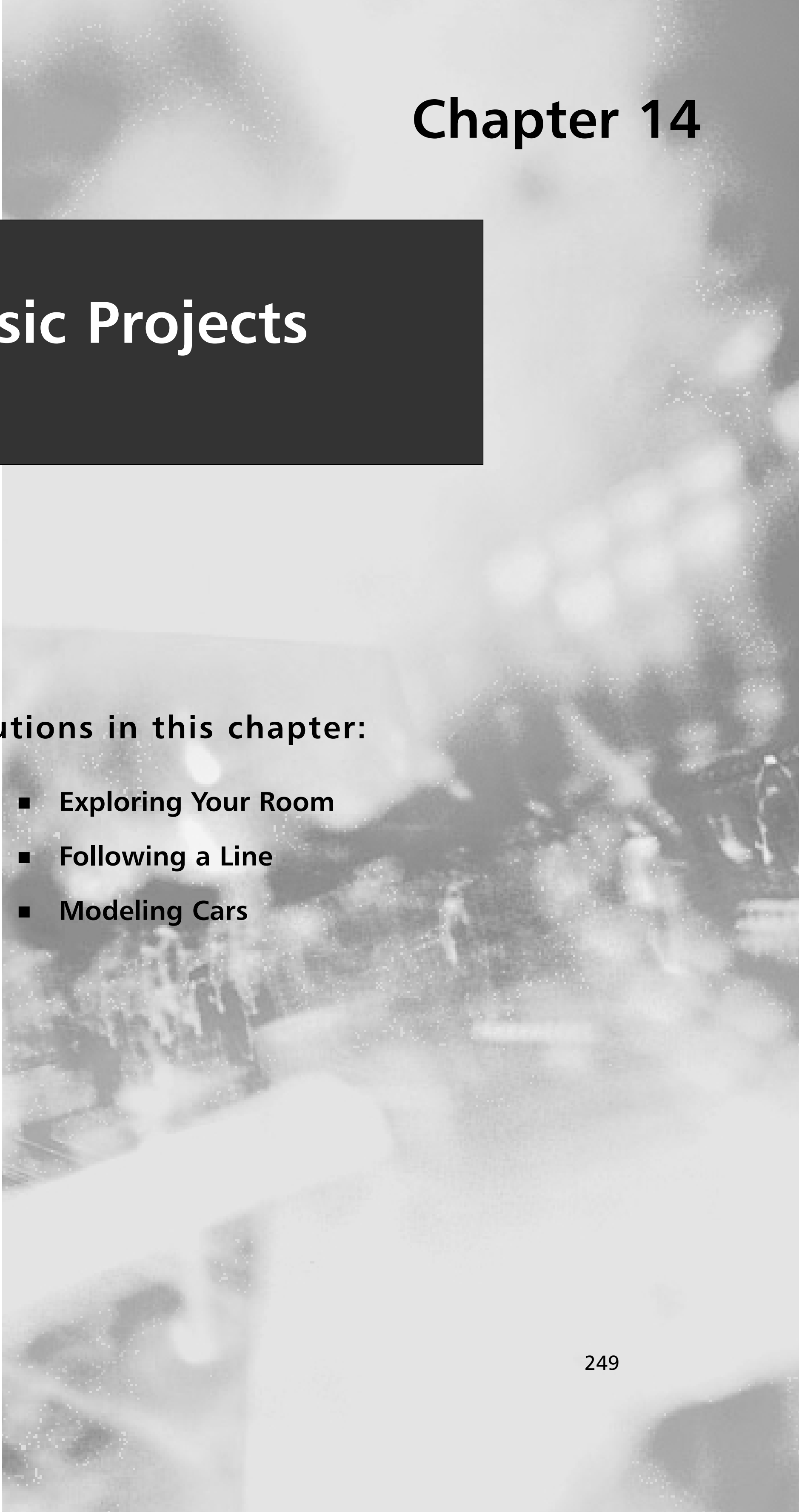
Milk Guard (Chapter 25)
A truly useful robot! Now you can warm your milk leaving the pan unattended on the stove. This robot will warn you when it reaches the right temperature, preventing it from boiling over.



Part II

Projects





Chapter 14

Classic Projects

Solutions in this chapter:

- **Exploring Your Room**
- **Following a Line**
- **Modeling Cars**

Introduction

From this chapter on, we will explore several example projects that could be the inspiration for many others of your own creation. As we already explained, the spirit of the book is not to provide you with step-by-step instructions, but rather to give you a foundation of information and let your imagination and creativity do the rest. For this reason, you will find some pictures of each model, some text that describes their distinguishing characteristics, and tricks that could be useful for other projects. Of course, we don't expect each detail to be visible in the pictures. It isn't important that your models look exactly like ours!

Another point we want to bring to your attention is that there is no reason to read the project chapters in Part II in order. Feel free to jump to the project that attracts you most, since they aren't ordered according to their level of difficulty.

In this chapter, we'll show you some projects that could be considered "classic," because almost everybody with a MINDSTORMS kit tries them sooner or later. Though you might not find them exciting, working with them is a good way to build up some solid experience and learn tricks that will prove useful in more complex projects. If this is among your first forays into robotics, we strongly suggest you dedicate some time to them.

All the robots appearing in this chapter have been built solely from the RIS 1.5 equipment. Only in describing some of the possible additions to the robots do we suggest extra parts.

Exploring Your Room

Well, actually "exploring" your room is too strong a term for what we are proposing here, it's more like *surviving* your room—your robot and your furniture could take some hits! The task here is to build a robot with the basic ability to move around, detect obstacles, and change its route accordingly.

For simplicity of design, and for the robot's ability to turn in place, we suggest you make this robot from a differential drive architecture, like the one shown in Figure 14.1.

We deliberately chose a gear ratio that makes the robot rather slow: 1:9, obtained from two 1:3 stages (Figure 14.2). This ensures that if you make some error in the code and the robot fails to properly detect the obstacle, it won't collide with it at too high a speed. Never expect everything to go well on the first try—because it won't!

Figure 14.1 Start with a Simple Differential Drive

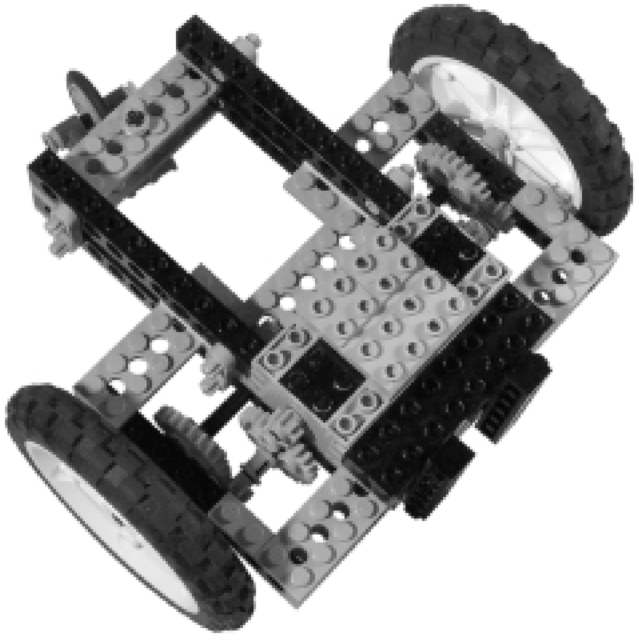
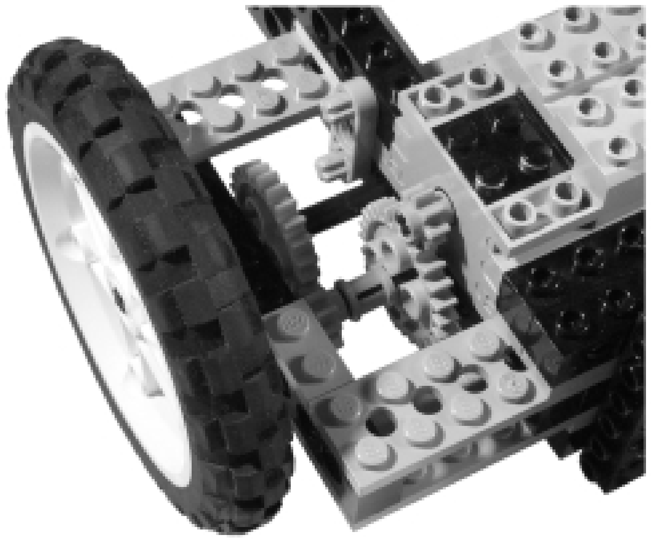


Figure 14.2 Detail of the Two-Stage Gear Train



When you feel satisfied with your software and your robot runs safely around your room, you can always try a faster ratio. Substituting the second 1:3 gearing with two 16t gears will give you an overall 1:3 ratio, making your robot about three times faster.

This robot has been designed to host the RCX behind the motors, and without the weight of the RCX in that position it actually does a wheelie (flips up)!

Another thing we'd like you to notice is that we made the robot rather large, keeping the main wheels far from the body of the robot. There's a reason for this, too: In a differential drive, the distance between the drive wheels affects the turning speed of the robot, because the wheels have to cover a longer distance during turns. The further the wheels are from the midpoint, the slower the turns. Since you're going to control turns through timing, slow turns are a desirable property which means finer movement control.

The caster wheel is the same kind we showed in Chapter 8. Now add the RCX and a couple of bumpers that are normally closed, like those in Chapter 4 (see Figure 14.3), and you're ready to go—well, ready to program the robot, anyway. Check out Figure 14.4 to see what the completed robot looks like.

Figure 14.3 Detail of the Bumper

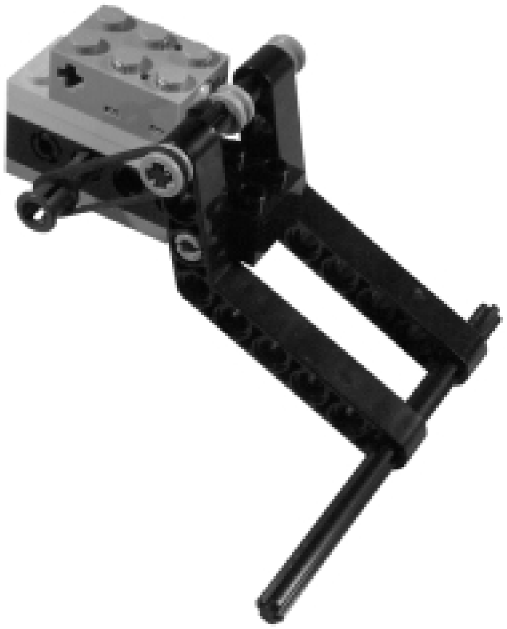
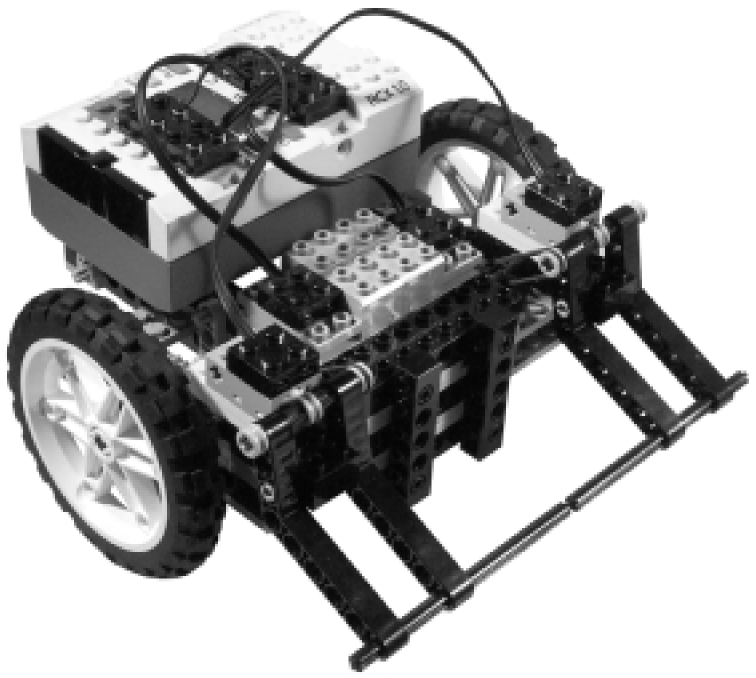


Figure 14.4 The Robot, Complete with RCX and Bumpers



The program itself is very simple: Go straight until one of the touch sensors opens. When that happens, reverse for a few fractions of a second, then turn in place, right or left depending on which bumper found the obstacle. Finally, resume straight motion.

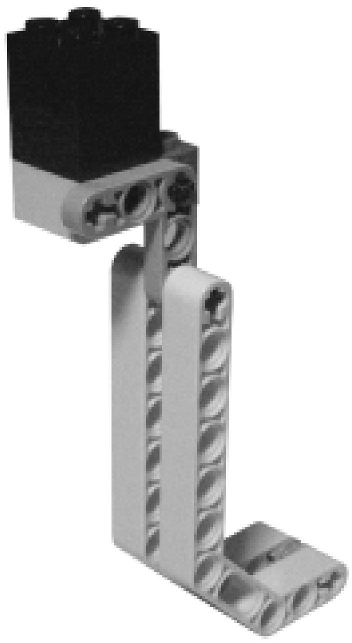
Experiment with different timing for turns, until you are happy with the result. You might also use some random values for turns to make the behavior of your robot a bit less predictable and thus more interesting. If you feel at ease with the programming, you can add more intelligence to your creature—for example, to make it capable of realizing when it's stuck in a cul-de-sac. This can be achieved by monitoring the number of collisions in a given time, or the average time elapsed between the last *n* collisions, and then adopting a more radical behavior (like turning 180°).

Detecting Edges

If your room has a flight of stairs going down, you can equip the robot with a kind of detector to sense the edge and avoid a bad fall. Normally, you would use a touch sensor for this, connecting it to a feeler flush with the ground (see Figure 14.5 for a detail). When the feeler in front of the robot drops, you have detected an edge.

Unless you have a third touch sensor, you are forced to use the light sensor. It's time to look back at some of the tricks explained in Chapter 4 and see if you find something useful. A light sensor can actually emulate a touch sensor: You have to place movable parts of different colors in front of it, so that when contact is made, the parts move, and the color of the brick in front of the light sensor changes.

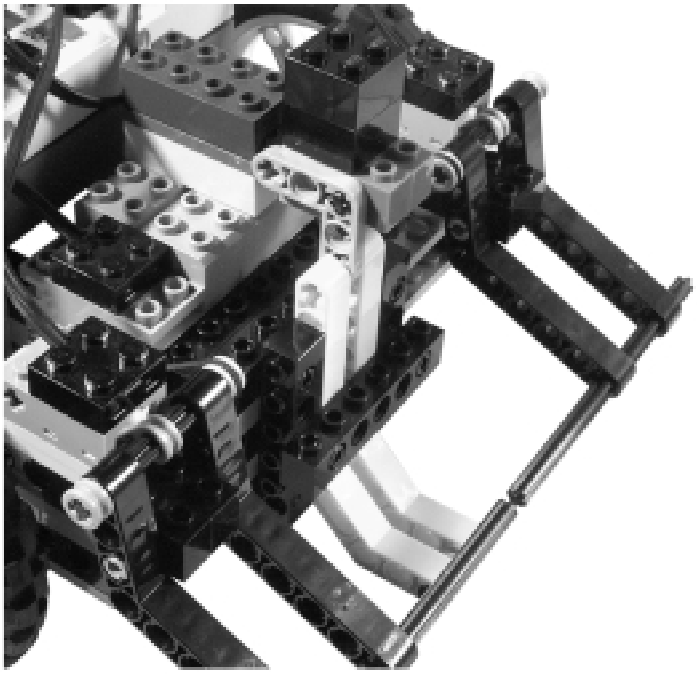
Figure 14.5 Edge Detection System Detail



We kept the edge sensor behind the bumpers, so that in most cases it doesn't interfere in the obstacle detection (Figure 14.6).

Unfortunately, this system doesn't cover all possible scenarios, because your robot could approach the edge at an angle that allows a wheel over the edge before detection occurs. You can improve upon the design and avoid this by providing the robot with two left- and right-edge sensors, but you'll probably have to give up the double bumper and go with a single sensor bumper.

Figure 14.6 The Edge Detection System Installed on the Robot



Using a different approach, you could write the software to make the robot very cautious, turning slightly left and right from time to time to see if there's a dangerous precipice around.

Variations on Obstacle Detection

If you own a couple of rotation sensors, you can experiment with indirect obstacle detection. Connect them to the main wheels, and program the robot to monitor their count while in motion. If both the motors are on forward, but the count doesn't increase, the robot knows an obstacle has blocked it. As a positive side effect, the rotation sensors allow you to use the same platform for experimenting with navigation, applying some of the concepts about dead reckoning explained in Chapter 13.

You can also implement indirect obstacle detection using a *drag sensor*. The idea requires that your robot keep a mobile part in touch with the ground, and that the friction that this part exerts against the floor surface when the robot moves activates a touch sensor. For example, you can use the friction of a rubber tire to oppose the force of a rubber band that keeps a touch sensor closed. When the robot moves, the friction of the tire on the floor overcomes the force of the rubber band and opens the touch sensor; as soon as the robot stops—or has gotten blocked by an obstacle—the friction disappears and the touch sensor closes.

Following a Line

The line-following theme is often mentioned in Part I of the book, as we think it is a very useful indicator of how different techniques can improve the behavior of a robot. The time has come to give it an official place, and face the topic in its entirety. Let's review what we have already said about line following:

- You must actually follow the edge between the tape and the floor, reading an average value between dark and bright, so when you read too dark or too bright you know which direction to turn to find the route back (Chapter 4).
- If you want to keep your software and robot as general as possible, you should use some kind of self-calibration process before the actual following begins. Calibration consists of taking readings of light and dark areas on the pad before actually starting line following. This lets your robot adjust its parameters to the actual lighting conditions at the time it runs, which are almost certainly different from the conditions it was designed in (Chapter 6).
- Some platforms can benefit from the introduction of a small quantity of hysteresis to reduce the number of corrections and get a higher efficiency. We explained in Chapter 12 that hysteresis widens the gray area between light and dark, which keeps the robot from spending too much time on course correction instead of moving forward!

To turn theory into practice and experiment with line following, you can use the same differential drive as in the previous project in this chapter. Remove the bumpers and mount a light sensor facing down, as shown in Figure 14.7.

The light sensor is stacked on a 2 x 4 black brick, which is attached to the structure with two black pegs placed into its tubes at the bottom. In fact, the diameter of those tubes is very close to one of the holes in the beams, so the pegs and axles fit very well (isn't LEGO wonderful?) The distance of the sensor from the pad is very important (Figure 14.8). Our experience teaches us that for the best results this distance should fall in the range of 1mm to 5mm (0.04 to 0.2 inches).

Figure 14.7 The Differential Drive Equipped for Line Following

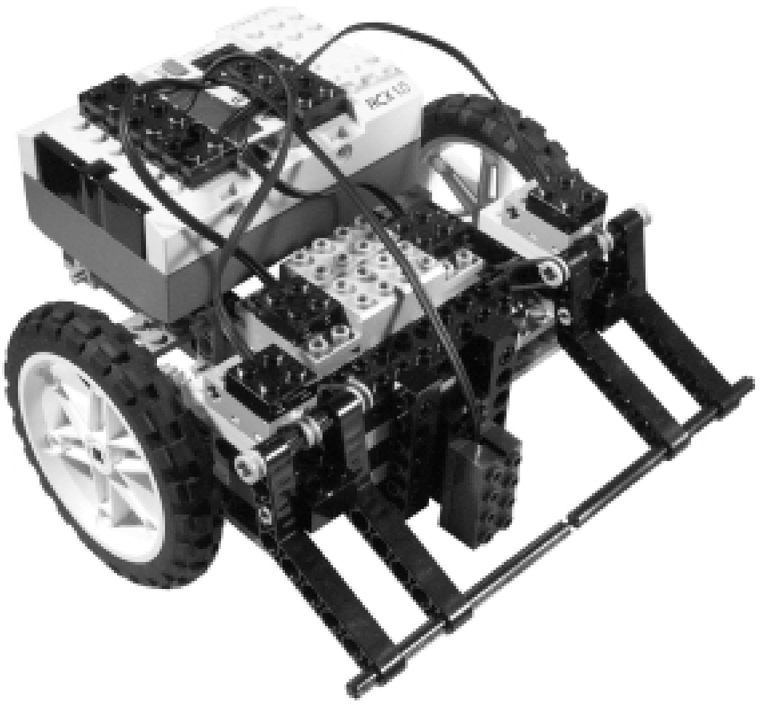
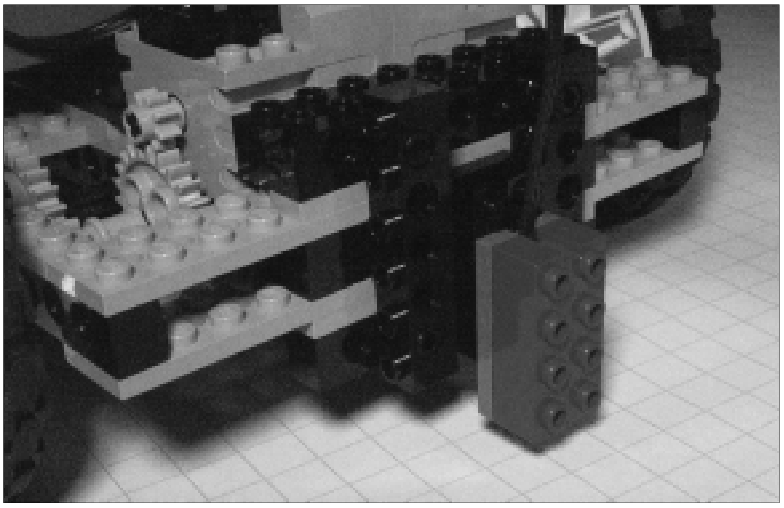


Figure 14.8 Set the Proper Distance between the Sensor and the Pad



Now that you've finished, you're ready to program and test your robot for line following. If you want to, you can use the NQC code from Chapter 6. It shouldn't be difficult to adapt it to the language of your choice.

Suppose you have succeeded in the task of programming for line following, and you feel quite happy with the result. You have good reason to, but you should also wonder, as always, if there is anything else you can do to make it better. What could "better" mean in this case? Probably "faster," along with little to no errors. This is what standard line-following competitions are about: going from one end of a line to the other in the shortest time, and at the highest speed.

Observe carefully your differential drive in action. When it turns in place to adjust the course, it makes no progress along the line. This approach gives your robot the ability to follow a winding line closely with very tight turns, but it isn't very efficient. A first step could involve changing the course adjustment algorithm to a better one—for example, making the robot turn with one wheel stopped and the other in motion, instead of turning with one going forward and the other reversed. Try this technique, and you'll see that your robot progresses much faster.

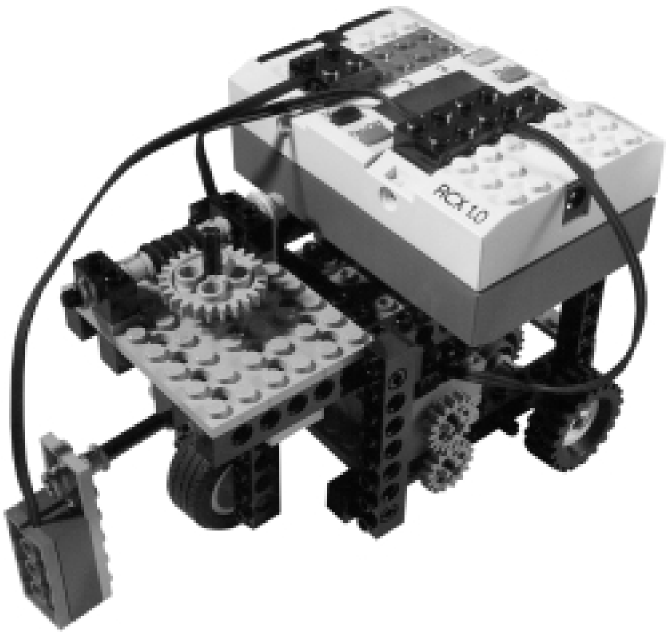
We haven't yet mentioned the most obvious improvement—increasing the speed of your robot! Try different gearings until you find the fastest setup that still allows your software to keep the course.

Nothing more you can think of? Imagine yourself standing over a line, straddling it with one leg on the left side and the other on the right. Now, gazing at the line, you advance one foot or the other trying to keep your eyes centered above the line. Do you feel a bit stupid? We would. This isn't actually what you would do to follow a line in a real situation. You're not a differential drive. You would rather walk as usual, putting one foot before the other, simply changing your direction without changing your speed.

That's the key: You need something that changes its direction without affecting the speed. Looking back at Chapter 8 with a different eye you will discover that the steering drive, the tricycle drive, and the synchro drive actually share this property of allowing changes in direction without changing the speed of their driven wheels. The synchro is probably too complex in gearing to be really efficient, so choose between a traditional steering and a tricycle drive. They are almost equivalent except on very tight turns. If you exclude turns with a very short radius from your path, proceed with the steering drive architecture that is simpler to implement.

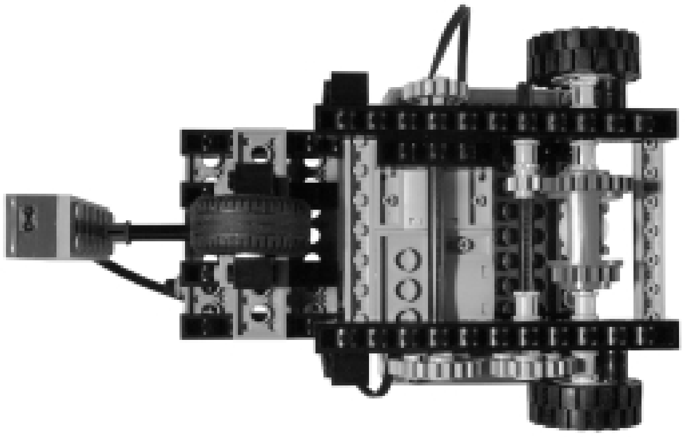
It's not imperative that a steering drive have four wheels, so ours will actually be a tricycle, because this makes the platform easier to build. Figure 14.9 shows our version of a steering line follower.

Figure 14.9 A Steering Line Follower



Let's dissect it to understand some of the choices we made, starting with the drive gearing; you might think this is a bit more complex than necessary (Figure 14.10).

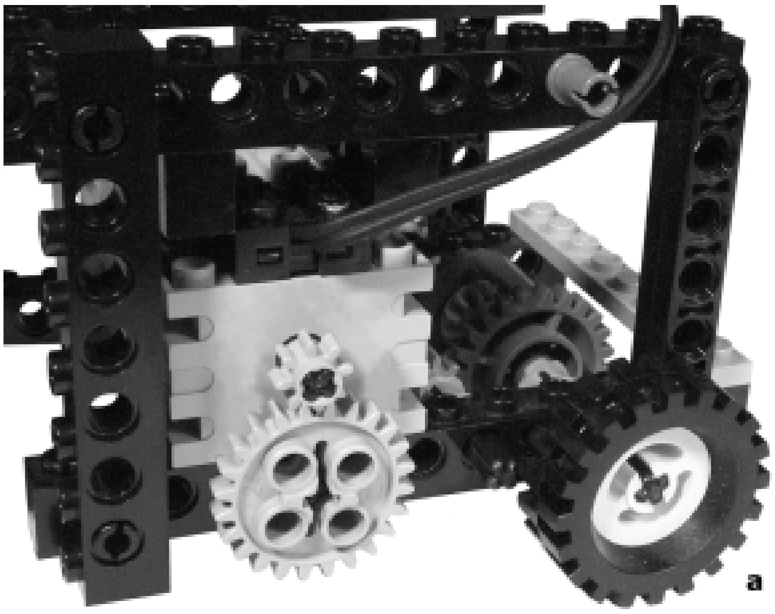
Figure 14.10 Bottom View: The Gearings

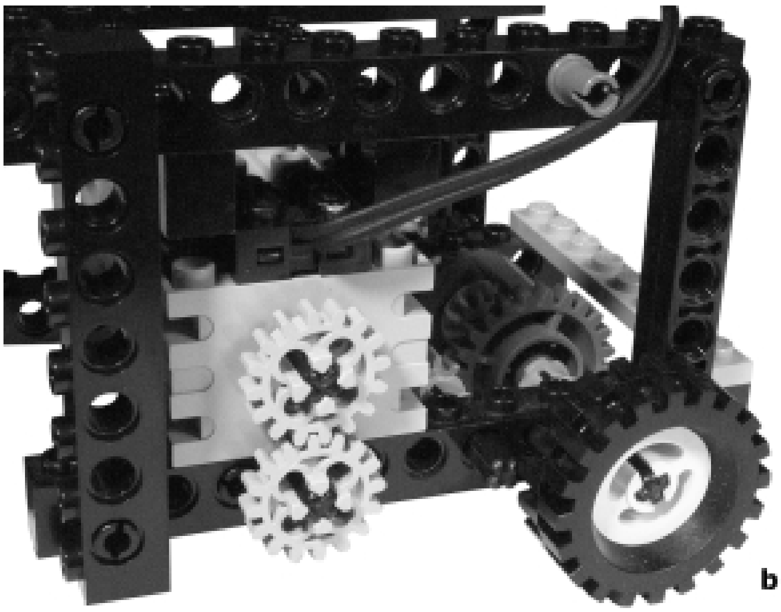


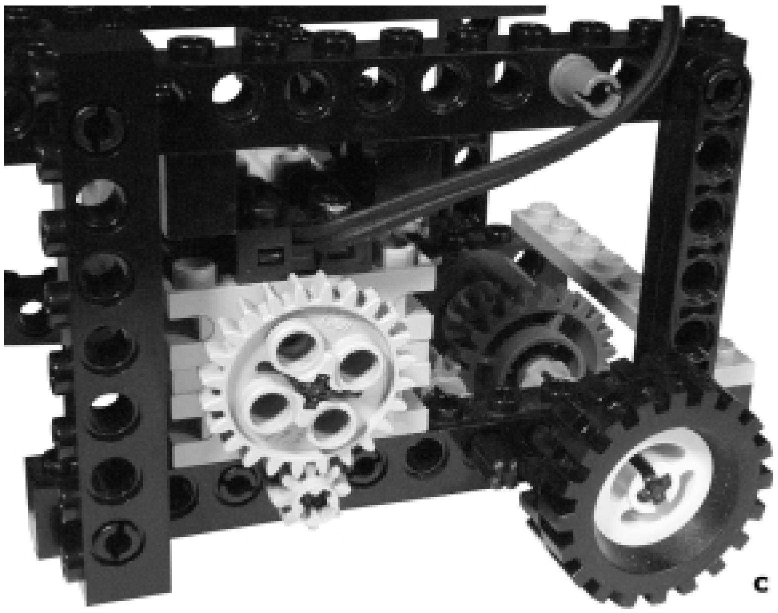
While it is possible to drive the differential from the motor without so many additional gears, we were looking for an easy way to change the gear ratio without having to take the robot apart. While you build the robot, you don't know yet at what maximum speed it will be able to keep following the line. Our solution brings a pair of gears on the outside of the model at a distance that allows at least three combinations, so it will be easy to experiment with different speeds (Figure 14.11, version a, b, and c):

a. 8t to 24t (1:3)
b. 16t to 16t (1:1)
c. 24t to 8t (3:1)

Figure 14.11 These Gears Are Easy to Replace with Different Combinations

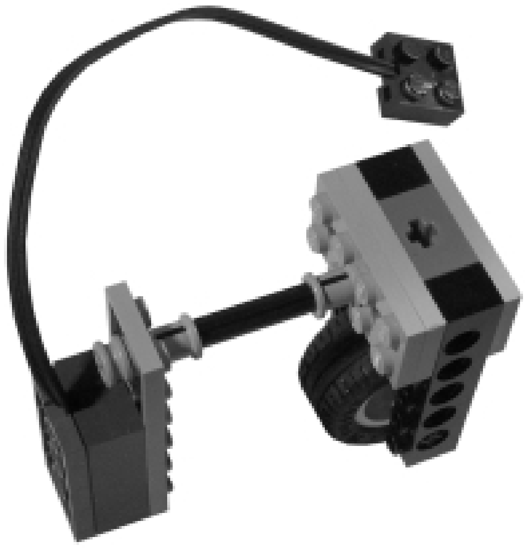



b


c

Now turn your attention to the front assembly. It's very simple, and there are just a few things worth mentioning. There is a green 1 x 2 brick with an axle hole that makes the fork drivable by the steering assembly, and another perpendicular axle that holds the light sensor at some distance in front of the wheel (Figure 14.12).

Figure 14.12 The Front Wheel Fork Assembly



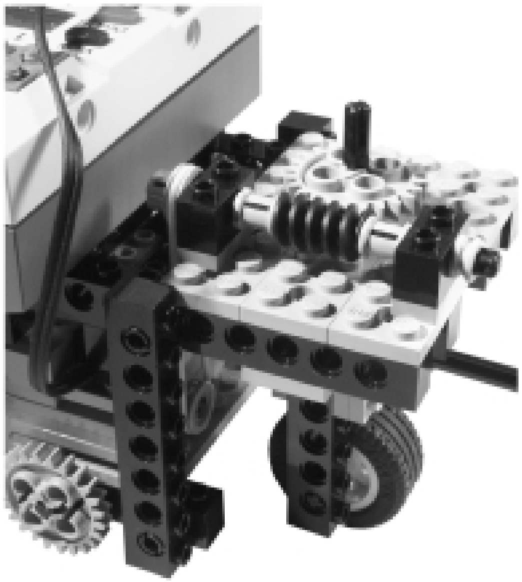
Notice that we used the indented side of the bushings to attach the axle to the plates: It perfectly matches the area among the four studs, and its friction is enough to allow the axle to carry a small weight.

The front fork is driven by a steering assembly based on a worm gear, a 24t and two pulleys. As you learned in Chapter 2, the pulley-belt systems prevent any stall situation if the robot doesn't control the steering wheel properly when the software is not yet fully tested and debugged (Figure 14.13).

Now it's time to program and test the robot. If you wrote the line-following program for the differential drive, it will work for this robot with a few minor modifications. During the run, the drive motor will always stay on, while the robot will adjust its course using only the steering motor, either in forward or reverse direction.

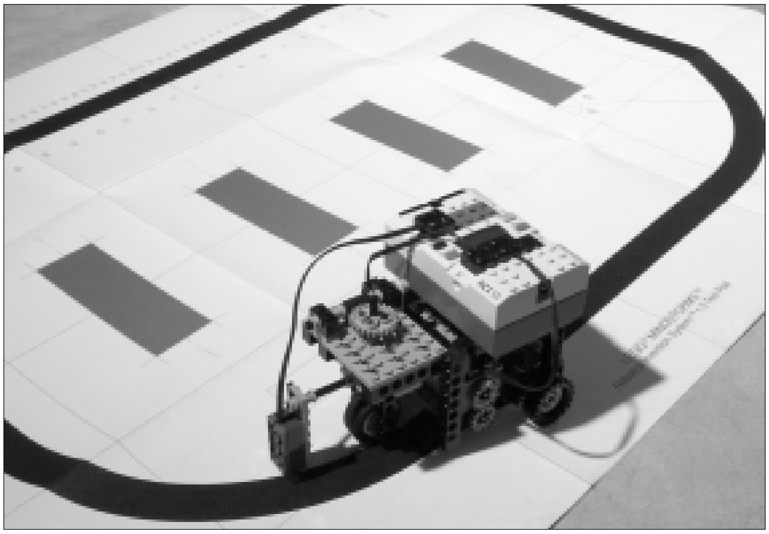
As for the calibration procedure, we suggest that with the robot still and placed on the borderline, you make it rotate the front wheel slightly left and right to read the minimum and maximum light values, then compute the average and position the light sensor onto that. This architecture needs very small hysteresis, or none at all.

Figure 14.13 The Steering Assembly



Start at slow speed, mounting an 8t gear onto the motor shaft, and when your robot is able to follow the line properly, try to change the ratio and increase the speed. Our own version (see Figure 14.14), programmed in NQC, was able to follow the line with the two 16t gears, running at about 20cm/s (0.65ft/s).

Figure 14.14 The Steering Line Follower in Action



The minimum radius that the robot can follow depends on a combination of the forward speed of the robot, how quickly the turn drive motor can move the steering wheel, and how far in front of the robot the light sensor is. We encourage you to experiment with these variables and see how the robot behaves when following lines with turns of different radius.

Further Optimization of Line Following

You should be happy with this line follower, it runs very smoothly compared to the differential drive configuration. Nevertheless, you might wonder if there's anything you can do to make it run at an even higher speed.

Changing the gear ratio is not enough. There is a large margin for an increase in speed—it's not difficult to set up a LEGO robot to run at about 2 m/s (6.5 ft/s). The problem is keeping the robot *on the line*.

When targeting excellence, the finer details are vital. There is a big difference between building a robot that "works" and a robot that is *optimized* for a given task. You already switched from a differential drive to a steering drive, and this change of architecture has proved a significant improvement, but now you have to dig into the particulars if you want to gain some further speed. In working with line following, one of the key factors is to make the steering fast and accurate. To achieve this, you can reduce the *backlash* between the 24t and the worm gear, for example, keeping the steering gently pulled back from one side with a rubber band. Use a long and soft rubber band, because you don't want to introduce too much friction, you want to simply keep the teeth of the 24t in contact with the worm gear, and always from the same side (Figure 14.15).

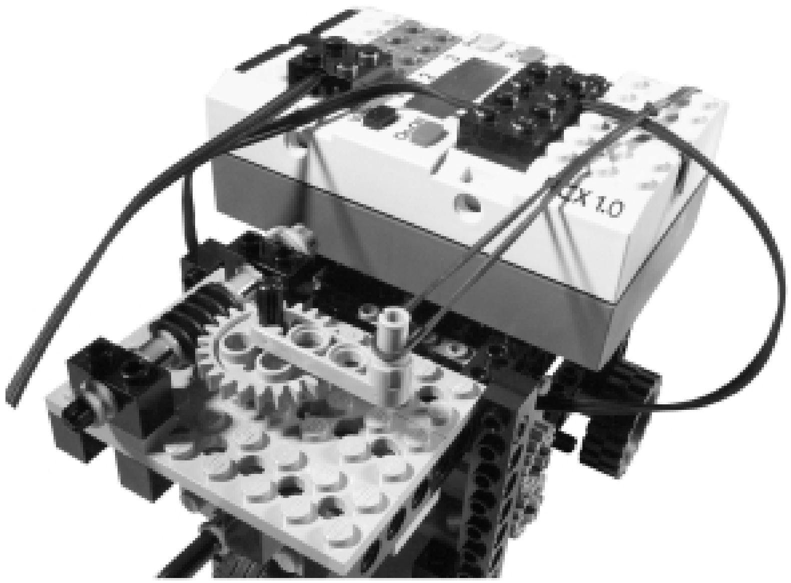
At this point, speed limitations are probably due to the rotation speed of the steering assembly. This pulley, belt, and worm gear system is safe and accurate, but a bit slow. We leave you the task of developing a faster steering assembly. It's not very difficult: Try connecting different pulley sizes to the steering mechanism.

Now you will meet the last barrier: the reaction time of the software. If you used the standard firmware, either with RCX Code, NQC, or other tools, the time it needs to interpret the instructions becomes relevant to the performance of your robot. Another very critical factor is the sampling frequency of the sensors, which is much higher in most replacement firmware than in the original LEGO one.

For these reasons, you have to switch to legOS, leJOS, or pbForth if you want to overcome this limitation and get the fastest reactivity from the software.

With a setup like the one described here, and using legOS, Paolo Masetti won a line following contest with a robot that ran at about 70 cm/s (2.3 ft/s)!

Figure 14.15 Using a Rubber Band to Reduce Gear Slack



Modeling Cars

For some reason, many people find the building of a robotic car a very attractive project, and we have more than a few friends that made this their very first experiment with robotics. This is more a mechanical task than anything else, but indeed offers some interesting topics of discussion and deserves its place among the "classic" MINDSTORM projects. The robot we're going to describe has been designed to allow basic obstacle detection while driving around the room. However, you can easily adapt it to other tasks, like line following, or enhance its abilities with edge detection as we did for the differential drive.

We started with the simple car chassis shown in Chapter 8, slightly modified to include a sensor that detects the position of the steering and another one for the bumper (Figure 14.16).

Starting again from the driving wheels, you'll notice that the gearing here introduces a small multiplication—a 24t against the 16t side of the differential gear makes a 1.5:1 ratio (Figure 14.17). As the differential drive of the model in the first part of the chapter was deliberately slow, we designed this vehicle to be deliberately fast so you can experiment with the problems that arise when detecting obstacles at (relatively) high speeds.

Figure 14.16 A Simple Steering Drive Robot

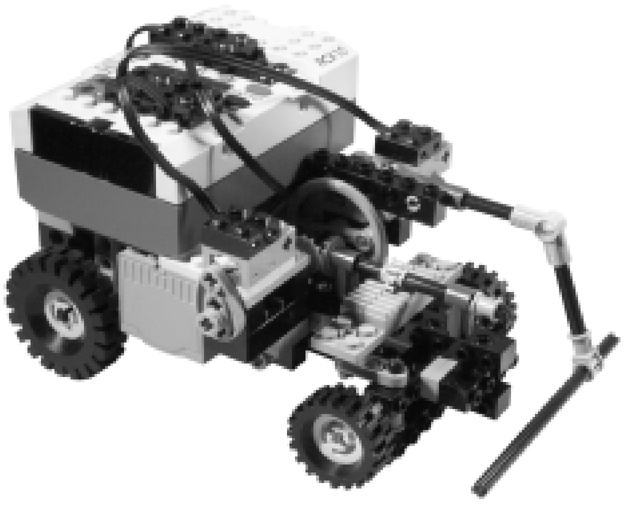
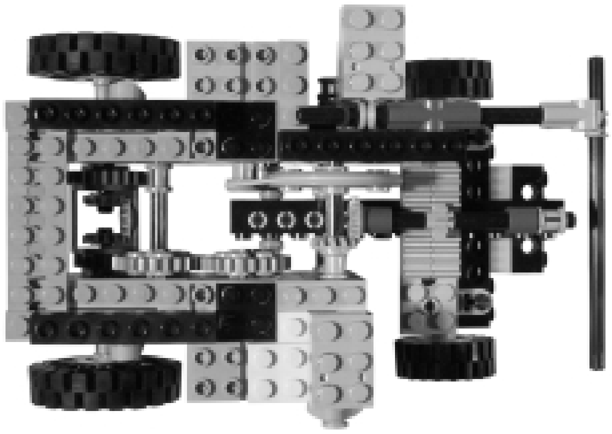


Figure 14.17 Top View (with RCX Removed)



Notice also in Figure 14.17 that we placed the intermediate 24t gear at a height that does not correspond to the standard grid. (If you look carefully, you'll see the 24t gears don't mesh as closely as usual.) You can see this more clearly in Figure 14.18, which shows the rear view of the vehicle.

As we explained in Chapter 8, the construction of a steering drive using parts from the MINDSTORMS kit requires a little trick. In our model, we built the entire front section of the car with the beams turned *on their side*. This has some advantages: It makes a solid support for the front wheel axles, and provides a smooth surface over which the racked plate can slide (see Figure 14.19).

Figure 14.18 Rear View

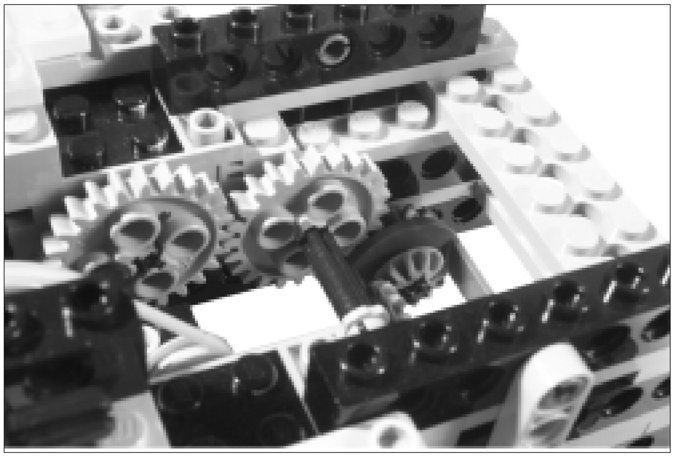
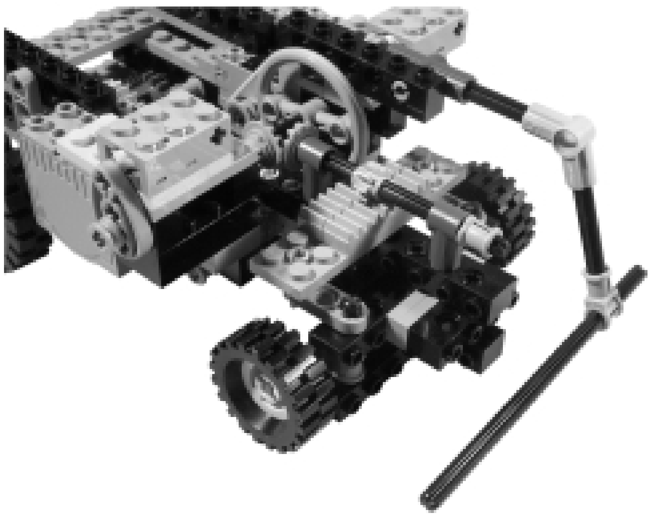
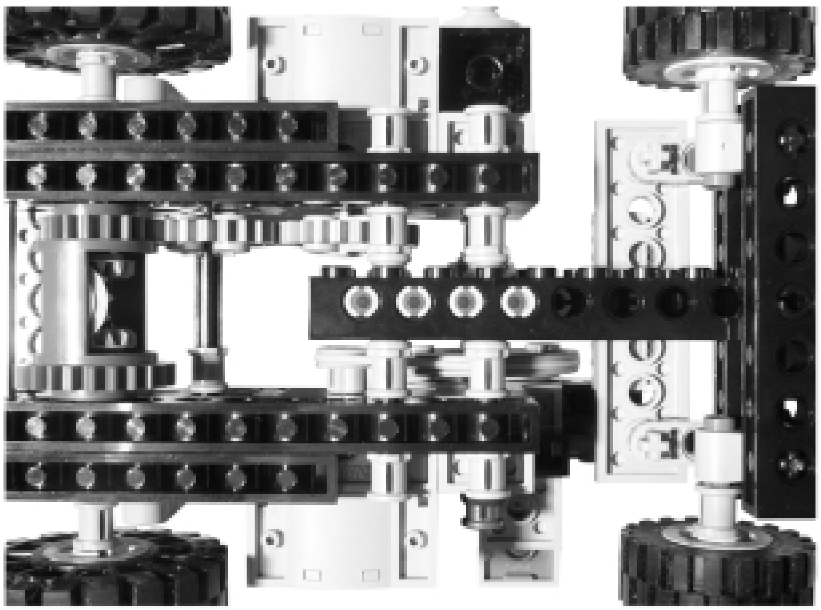


Figure 14.19 Detail of the Rack and Pinion Steering Assembly



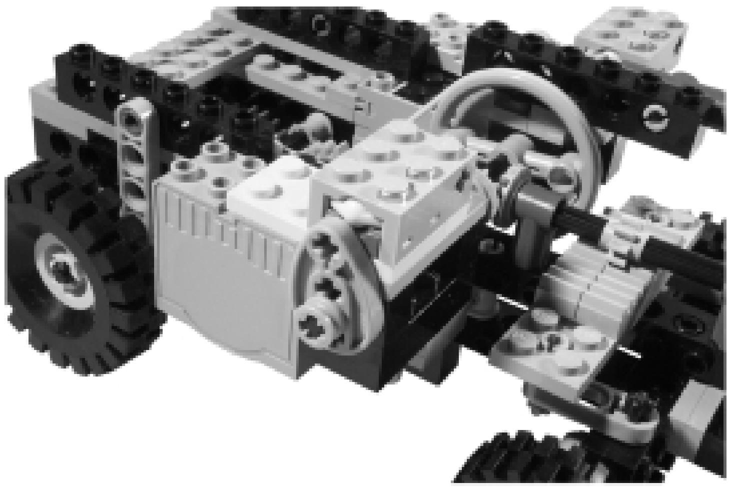
To connect the front and rear sections, we used two yellow connectors with two pegs per side and a crossed hole in the middle, named the *3L Double Pin*. You can see the details in Figure 14.20.

Figure 14.20 Bottom View: Front and Rear Section Connection Detail



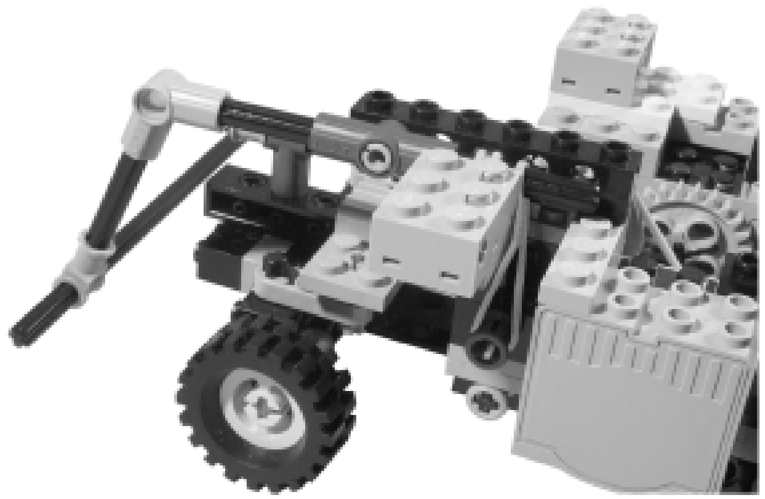
In comparison with the version of this vehicle shown in Chapter 8, here we've added two sensors. The first serves the purpose of noting the position of the steering, or more precisely put, the touch sensor closes when the wheels are centered (Figure 14.21). It's only the tip of the cam that operates the sensor; this gives a narrow band of center indication and makes it easy to keep the car going straight.

Figure 14.21 The Touch Sensor that Monitors the Steering



The second sensor is operated by the bumper. Having just one touch sensor left, we decided to equip the vehicle with a single bumper, again one that is normally closed (Figure 14.22).

Figure 14.22 The Bumper



To program this robot, we suggest this simple sequence:

1. The robot goes straight, monitoring the bumper in a tight loop so it stops as soon as it detects an obstacle. If you want effective braking, you can also briefly reverse the main motor.
2. To avoid the obstacle, the robot maneuvers like real cars do. It turns the steering left (or right) and goes back a bit, then turns right (or left) and goes forward another short distance before finally centering the steering and resuming straight motion.

Your program will include three subroutines **Steer_left**, **Steer_right**, and **Center_steer**. The first two assume the steering is in its central position, and simply switch the steering motor on for a few hundredths of a second in the proper direction. The **Center_steer** subroutine needs to steer left or right until the touch sensor closes. The direction of the steering motor depends on which steering routine has been called: You have to reverse the direction. The following NQC code fragment suggests a possible implementation, which assumes that the steering motor is connected to output port B and the steering touch sensor to input port 1:

```
void Steer_left()
{
  OnFwd(OUT_B);
  Wait(10);
  Off(OUT_B);
}

void Steer_right()
{
  OnRev(OUT_B);
  Wait(10);
  Off(OUT_B);
}

void Center_steer()
{
  Toggle(OUT_B);        // invert the direction of the motor
  On(OUT_B);
  while (SENSOR_1==0); // waits for the sensor closes
  Off(OUT_B);
}
```

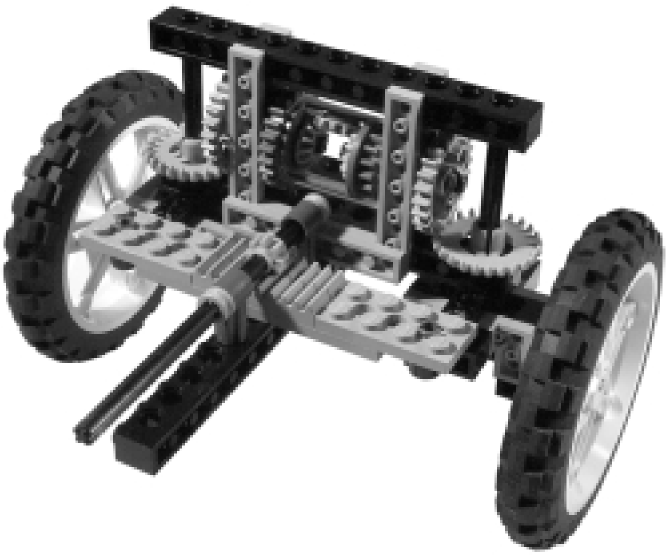
The **Toggle(OUT_B)** command in the **Center_steer** routine instructs the RCX to invert the turning direction of the motor connected to out port B, whichever it was. For example, after **Steer_right**, the direction of the motor was "Reverse;" a call to **Center_steer** will toggle the direction to "Forward."

Front-Wheel and Four-Wheel Drives

If you feel inclined towards tough challenges, you can try to make a front-wheel drive vehicle. The main difficulty is transferring the drive to the steering wheels, in which case you have to face problems similar to those we discussed in Chapter 8 about the synchro drive, though in this case the wheels don't need to rotate 360° and you can feel satisfied with something like +/-45° from the straight position.

If you're limited to using only MINDSTORMS parts, your life won't be made easy. Nevertheless, you can achieve your goal. In Figure 14.23, you see an assembly which, though not very compact or elegant, solves the problem.

Figure 14.23 A MINDSTORMS-Only Front-Wheel Drive Assembly

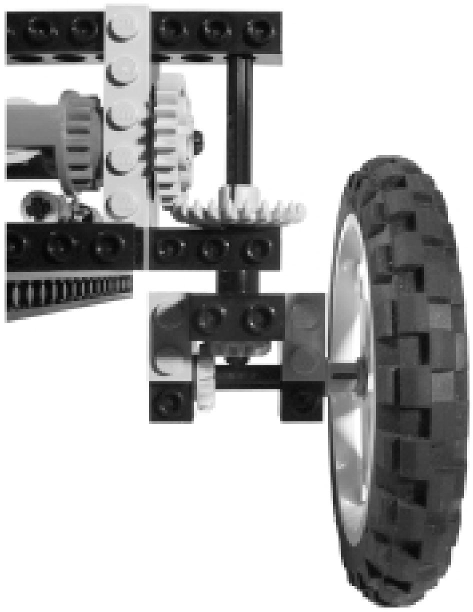


The trick we used here is to send the motion to the wheels along their pivoting axles, exactly like in a synchro drive. And the wheel assembly also works like the one used for our synchro of Chapter 8 (Figure 14.24).

NOTE

This section describes an alternative to using gears to transfer drive motion to the wheels by using universal joints. Universal joints suffer from the side effect of changing the rotational velocity of the output shaft as the angle between the axles increases. That's why big trucks have long drive shafts: The angles don't change that much. Modern front-wheel drive cars use a more complicated version of the universal joint called the constant-velocity (CV) joint.

Figure 14.24 Detail of the Wheel Assembly



Years ago, the LEGO company released a classic model, the 8880 Supercar, that featured four-wheel drive and four-wheel steering systems (and a gear shift and many other amazing things like full suspension and an eight-cylinder motor). That model, still considered by collectors to be one of the best LEGO TECHNIC models ever, used some very special parts to drive and steer all four wheels. Unfortunately, those parts are hard to find, but even using more common extra parts, you can still design an effective front-wheel drive.

Our design is based on an original concept by Fabio Sali. The key component is the LEGO *universal joint* located inside the steering assembly, which hosts two axles that connect the differential to the wheel (Figure 14.25).

The universal joint is able to transfer motion through two angled axles. It cannot even get close to 90°, but for a vehicle with a maximum steering angle of about 45°, it works well. Keep in mind, it's very important to keep the joint perfectly aligned along the pivoting axle.

Sali's design also has the merit of introducing some Ackerman correction in the steering geometry (see Chapter 8). As you can see in Figure 14.26, when steering, the inner wheel turns more than the outer one, achieving a shorter radius.

Figure 14.25 A Front-Wheel Drive Based on the Universal Joint

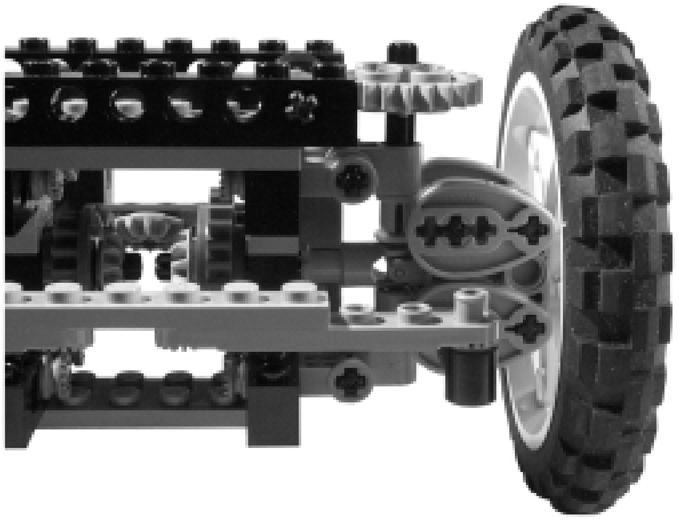
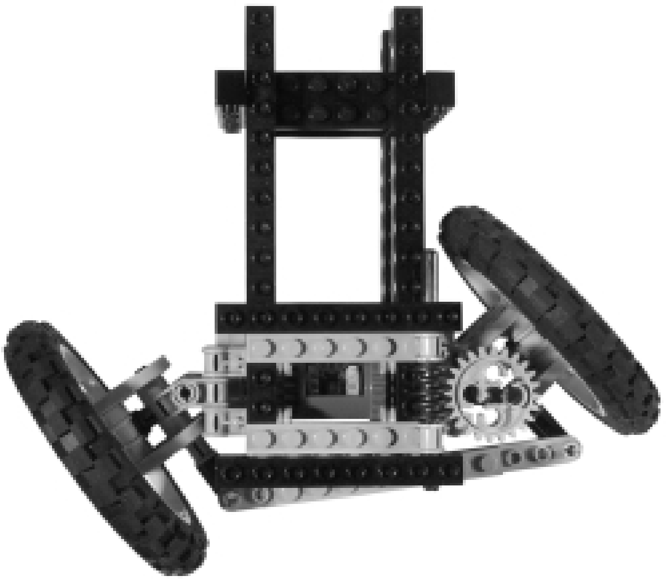


Figure 14.26 This Design Features Ackerman Correction



Notice also that the arm connecting the wheels does not remain parallel to the front side of the chassis. For this reason, it's not possible to employ the simple rack and pinion scheme we used before. You should substitute the jointed arm for the straight and rigid arm carrying the rack. We used a different approach, and mounted a 24t gear over the left wheel steering assembly, driving it with a worm gear (to be connected to the steering motor).

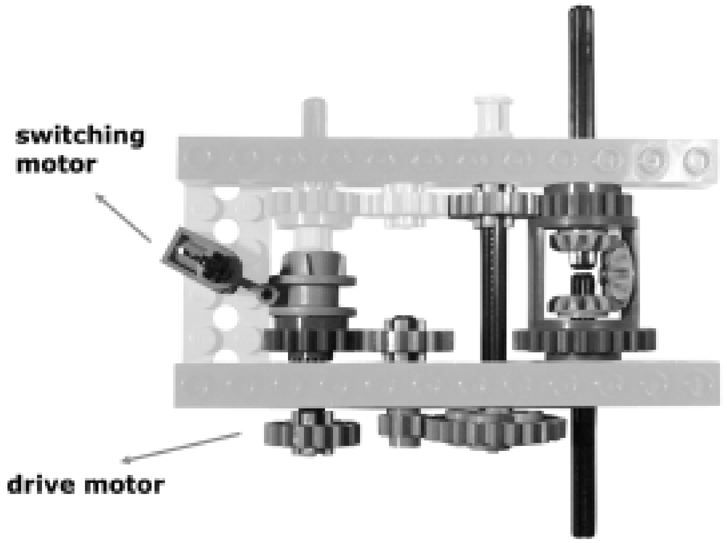
Once you've solved the front-wheel drive problem, you're more than half way to your goal of a four-wheel drive. What you must do next is build a traditional rear-wheel drive system with a differential gear, then connect the front and rear differentials with a third differential, whose body is powered by the main motor.

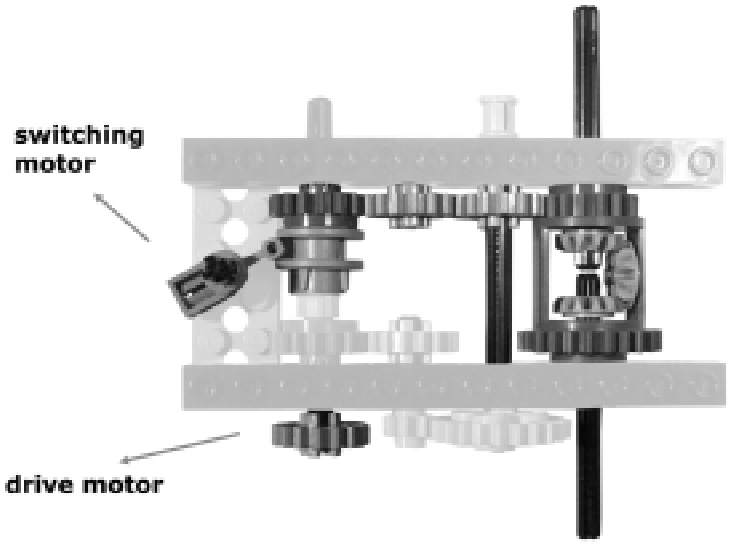
Switching Gears

If you want, you can complete your robotic car with a gear switch. A simple two-speed gear switch is not difficult to build, and your car could use the first speed when starting, maneuvering, and climbing slopes, then the second when traveling on level terrain.

If you have the special parts referred to as the 16t gear with clutch, the transmission driving ring, and the transmission changeover catch, you can assemble a simple gear switch like that in Figure 14.27. The two dark gray 16t are normally free to rotate on their axles, since they don't have a crossed axle hole but rather a circular hole. The driving ring, on the other hand, rotates solidly with the axle. When the changeover catch pushes the ring partly inside the gear with clutch, the latter becomes engaged.

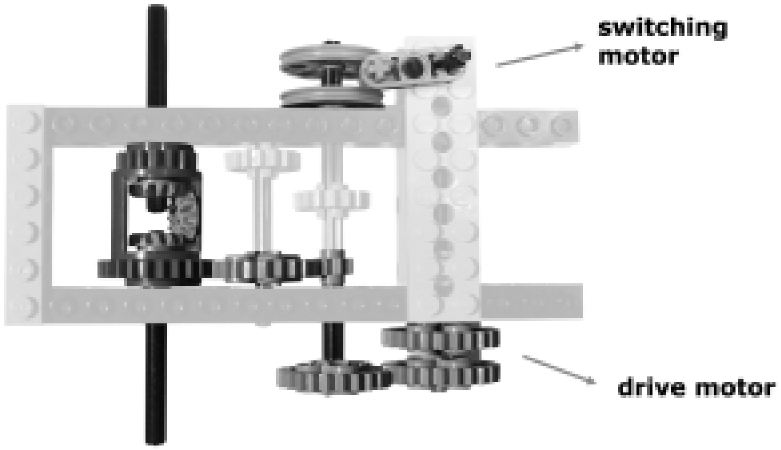
Figure 14.27 A Gear Switch Based on the Transmission Ring

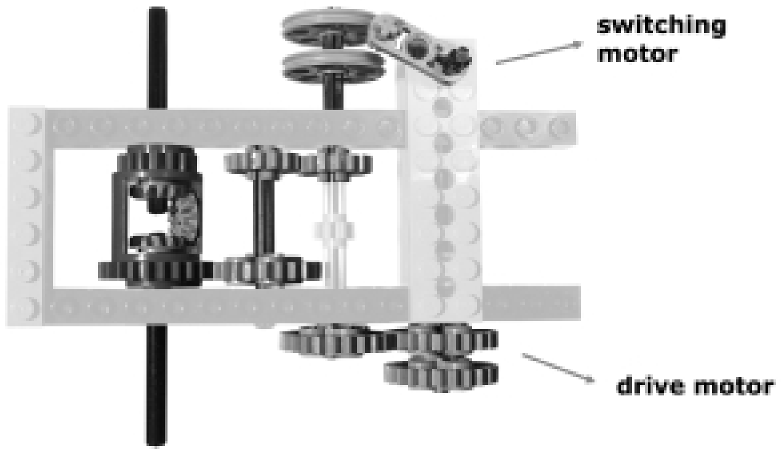




You can just as easily build your gear switch using more available components. The switch in Figure 14.28 resembles what we incorporated into our first robotic car. There is a movable shaft that can engage either an 8t to 24t pair (first speed) or a 16t to 16t (second speed).

Figure 14.28 A Gear Switch





The problem with this gear switch is that the gears are not very easy to engage and they tend to grind, which is exactly what happened in pioneer cars before the invention of the synchronized gear switch. Remember to keep the drive motor running while switching, as this helps the gears in finding their match.

Both the switches shown here require that a motor be operated, so your car will mount three motors for the three functions: drive, steer, and gear switch.

Using the Gear Switch

How would you use the gear switch? Your software will know when the vehicle is starting or maneuvering rather then traveling, and can consequently engage the proper speed. Our first robotic car was programmed to begin in first gear, and switch from first to second gear after a few seconds of straight motion. When blocked by an obstacle, it stopped and maneuvered to avoid it according to the following sequence: switch to first gear, steer left, go back for a short distance, stop again, center steering, resume forward motion.

You can make your robotic car even more sophisticated by equipping it with a sensor capable of detecting changes in the slope or in the quality of the terrain that may require the first speed. The simplest solution involves a rotation sensor, through which the program can monitor the actual speed of the vehicle and switch to first speed if something slows it down. This is an indirect approach: The robot knows nothing about the nature of the terrain, but detects that "something" is slowing it down. So, just as we suggested for the differential drive, this steering drive can also use the rotation sensor to detect obstacles—when the drive motor is powered but the wheels don't rotate, an obstacle has surely blocked your robot.

You can use a different approach, which does not require a rotation sensor, and build a *slope* sensor, something that informs your software that the car is climbing a slope. For example, you can make a sort of pendulum that releases a touch sensor when the car inclines more than a given angle, or, using a technique similar to the one described for the edge detector, make your pendulum alternate colored bricks in front of a light sensor.

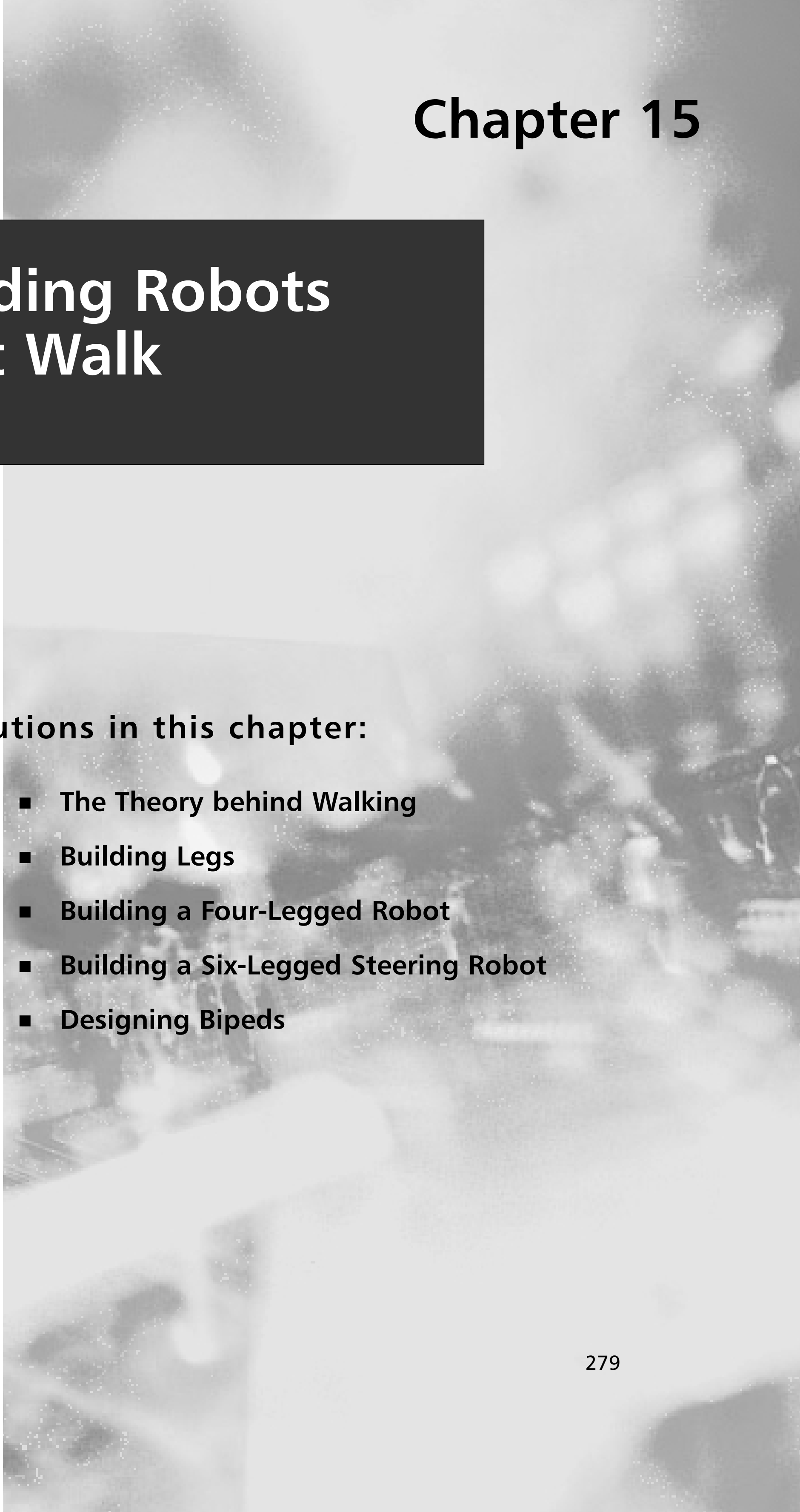
Summary

You may not have any interest in the topics covered in this chapter—you may think: What a bore. I'd like my robots to do more than just follow a black line or run around my room bouncing against obstacles…

You're right, there *are* more interesting activities you can program your robots for, but these classic tasks help lay the foundation for more complex projects in the future. They reveal that even apparently trivial projects conceal unexpected difficulties, and we're sure the time you spend experimenting with line following and simple navigation will indeed pay off in the end. In this chapter, you also had the opportunity to review many of the tricks learned in the first part of the book and see them at work. We applied the concepts of Chapter 5 to make solid structures and build two of the most important types of mobile configurations described in Chapter 8: the differential drive and the steering drive. Naturally, to build these configurations we used the principles of the first two chapters concerning the geometric relationships of LEGO parts and the proper use of gears. We recalled the techniques of Chapter 4 about making good bumpers, about using light sensors for line following, and even about emulating a touch sensor with a light sensor. From Chapter 6, we took the idea that good code should be as general as possible, and for this reason we suggested using a self-calibration routine to make your line-follower suitable under any light conditions. Even the math of Chapter 12 had its applications here: We recalled the idea that hysteresis can improve the efficiency of your robot in tasks like line following.

Like a thread sewing together most of the topics of Part I, the simple robots of this chapter demonstrated what we've stated on more than one occasion: Robotics involves many disciplines, and a good design cannot neglect any of them!

Another theme that pervades the chapter concerns the care you should put into the particulars: building a robot that works roughly as expected, compared to looking for optimal performance, are two very different approaches. Obviously, the concept behind the *design* of a robot is an important element in regards to its functioning; we explained how shifting from the differential drive to the steering drive can have an impressive effect on the resulting efficiency. But even after finding a satisfactory architecture, there is still much work to be done to optimize the single subsystems of the robot. The details, as often happens, make the difference. You will see this prove true in the last part of the book, where we will explore the world of robotic *contests*. Contests are a great incentive to pursue optimization, the kind of motivation that makes you spend all night rebuilding a working robot from scratch in search of that little improvement that will make all the difference!



Chapter 15

Building Robots That Walk

Solutions in this chapter:

- **The Theory behind Walking**
- **Building Legs**
- **Building a Four-Legged Robot**
- **Building a Six-Legged Steering Robot**
- **Designing Bipeds**

Introduction

So far in this book, we have discussed in depth many mobility configurations, all of them based upon one of the most important inventions of humankind: the wheel. In this chapter, we will try and emulate what nature invented long before the wheel to provide humankind with a mode of transportation—legs!

Legged robots are rather unpractical for all but some special applications, but there's much to learn in designing and building a walking robot, and the matter is both challenging and fascinating.

This chapter owes a lot to all the great designers who published their creations on the net, and patiently explained their choices through text and pictures, including Joe Nagata, Jin Sato, Kazuhiro Umeda, Miguel Agullo, and many others.

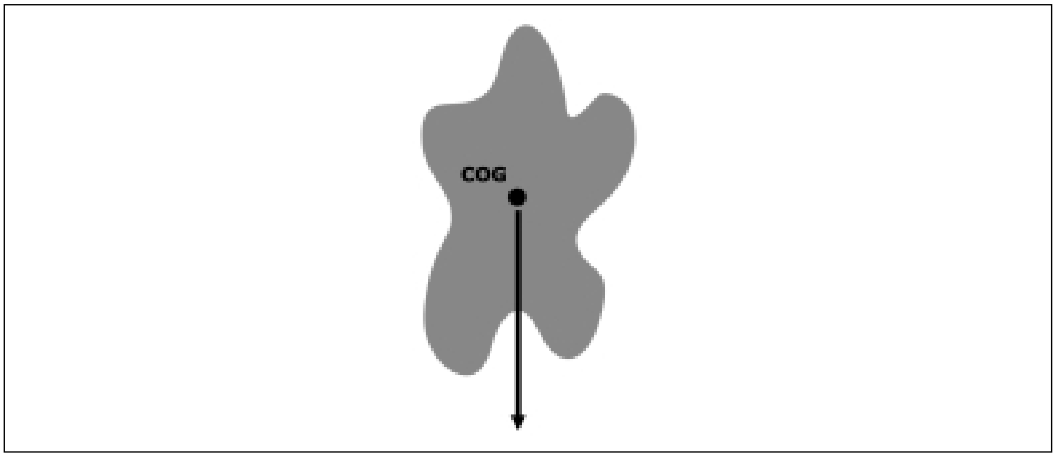
The Theory behind Walking

How can one define walking? It's the process of lifting a leg from the ground while one or more other legs support the body. When the leg has been lifted, it gets advanced and lowered back to the ground. From there the process continues with another leg, and so on.

The crucial point is this: What prevents a creature from falling down when a leg is lifted? To discover this, we need to introduce some basic concepts from a branch of physics called *statics*, which explains the laws of balance.

The weight of an object is the resulting effect of the force of gravity against the mass of the object. To describe a force, you need to determine three variables: its intensity, its direction, and its point of application. For example, if you want to move a piece of furniture in your room, the intensity is the amount of strength you must apply to make it move, the direction is the bearing of the course you're pushing it on, and the point of application is where you place your hands to apply the force. Returning to *gravity*, its intensity is proportional to the mass of the object. Its direction points vertically downward, but where is its application point? To answer this question, you should consider an object as being the sum of a very large number of very small particles, each one having its own mass. The gravity exerts a force upon every particle, and thus all of them can be considered a point of application. However, physics teaches that a combination of forces can be interpreted as a single force—called the *resultant*—which has its own intensity, direction, and point of application. The resultant of the force of gravity has an intensity which corresponds to the weight of the objects, a direction pointing downward, and a point of application called the *center of gravity* (COG) of the object (Figure 15.1).

Figure 15.1 The Center of Gravity of an Object



The force of gravity acts on any object and tries to move its COG as close as possible to the ground; this is why objects fall and shift until they reach a stable position. But what makes a position stable? Statics teaches that a body becomes stable when the vertical passing for its center of gravity falls inside its supporting base. The supporting base is the surface whose perimeter results from connecting the supporting points with straight lines, where the supporting point is any point on the object which is in contact with the ground or with any other stable object (like the floor of your room or your desk). For example, a book placed on a table has the whole surface of its cover in touch with the table, and that defines its supporting base. A table has four legs, each one having a small surface in touch with the floor: its supporting base is the area delimited by the legs, which includes points untouched by the table (Figure 15.2).

Every child learns this rule by experience when building towers of stacked blocks: while the COG remains within the supporting base, the tower is stable; as soon as it falls outside the base, the tower itself falls down (Figure 15.3).

Okay, now you know the rule, but where's the COG of an object? For objects that are symmetrical in shape and density, the COG coincides with their geometrical center, but in more complex objects the COG is not very easy to find, and it is not guaranteed to be *inside* the object. A table is again a good example: The COG of a typical table lies somewhere below its top, as demonstrated by the fact that it has more than just one stable position (Figure 15.4).

Figure 15.2 The Supporting Base of a Table

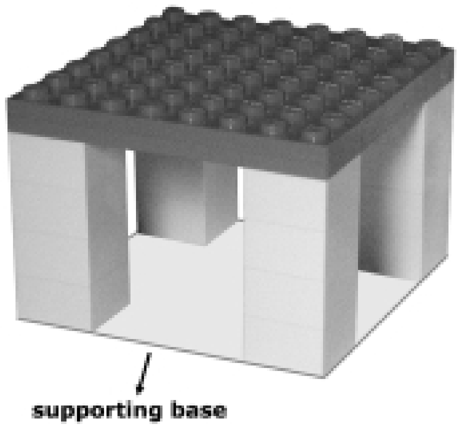


Figure 15.3 Stable and Unstable Piles of Bricks

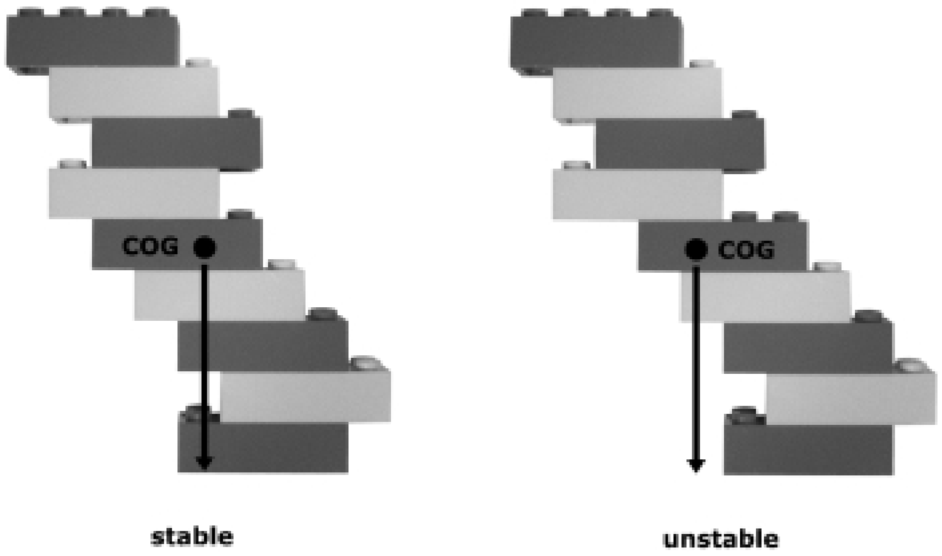
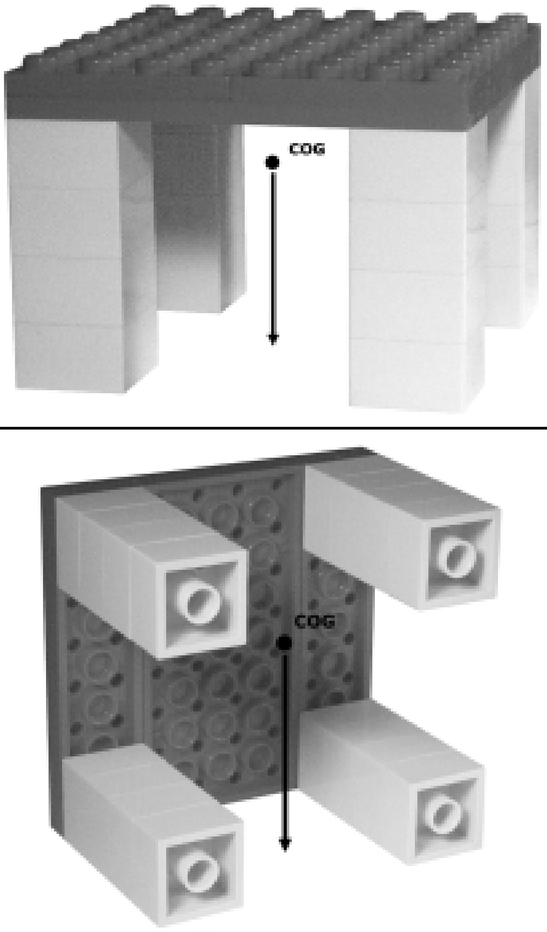


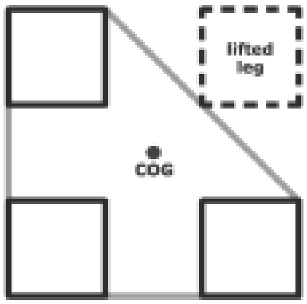
Figure 15.4 The COG of an Object May Lie Outside It



Fortunately, you don't need to find the actual position of the COG of your robots. You are actually interested in the position of the vertical line that passes through the COG, in order to see if it falls inside the supporting base. This is easier to find. If your robot is mainly symmetrical, this line will pass very close to its geometrical center. Thus, what you really need is to look at your robot from the top, to figure out if the COG falls over the supporting base delimited by the legs.

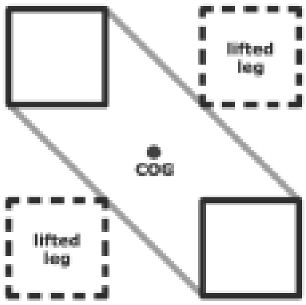
For example, in Figure 15.5 you see a scheme that represents a robot with four large legs (top view). One of the legs is lifted, and you see that the COG falls inside the surface delimited by the other three legs. Thus, the robot is stable.

Figure 15.5 A Four-Legged Robot with One Leg Lifted



The same robot can stay balanced even with just two legs, because the COG still falls inside the supporting base (Figure 15.6).

Figure 15.6 A Four-Legged Robot with Two Legs Lifted



When the robot advances the two lifted legs, part of its mass moves forward, and the COG moves forward a bit, too. But the large contact surfaces of the legs delimit a zone wide enough to make the COG fall within the boundaries (see Figure 15.7).

Using more than four legs, you don't need to rely on their size anymore. A six-legged robot, for example, can walk with very thin feet provided it always has at least three of them touching the ground (Figure 15.8).

On the contrary, when reducing the number of legs, things become more complicated. The making of two-legged (biped) robots requires a very careful design. A little trick is to build U-shaped legs that partly interlace, providing a large support for the robot (Figure 15.9). LEGO suggested a similar approach in one of its Idea Books (8891, back in 1991).

Figure 15.7 A Four-Legged Robot with Two Legs Lifted and Advanced

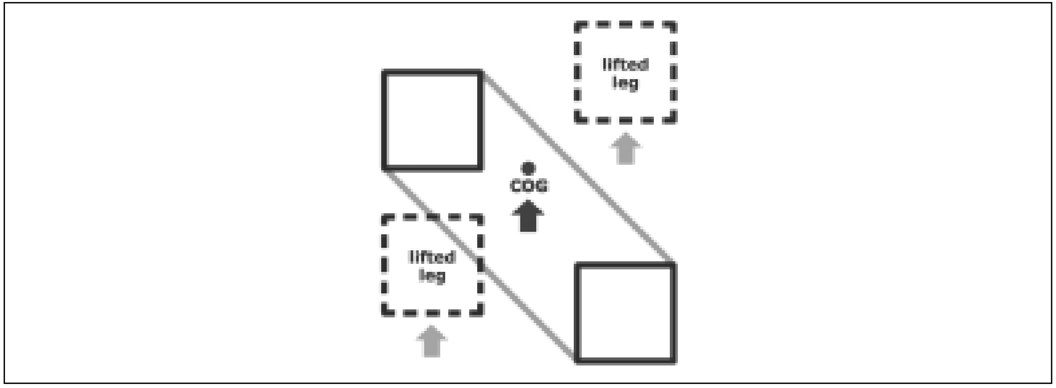


Figure 15.8 A Six-Legged Robot with Three Legs Lifted

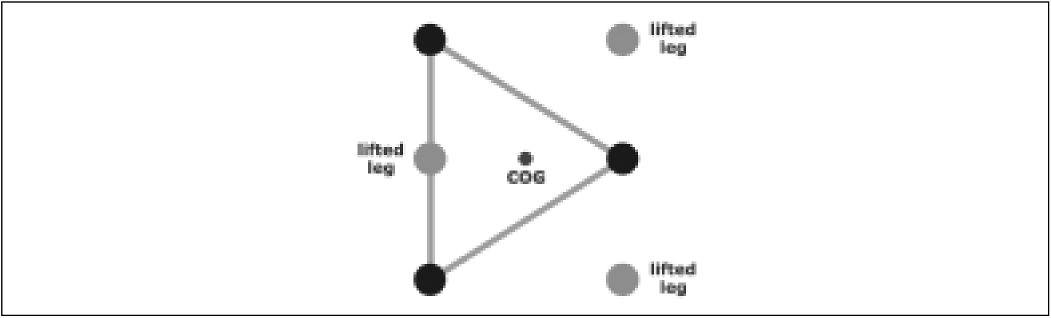
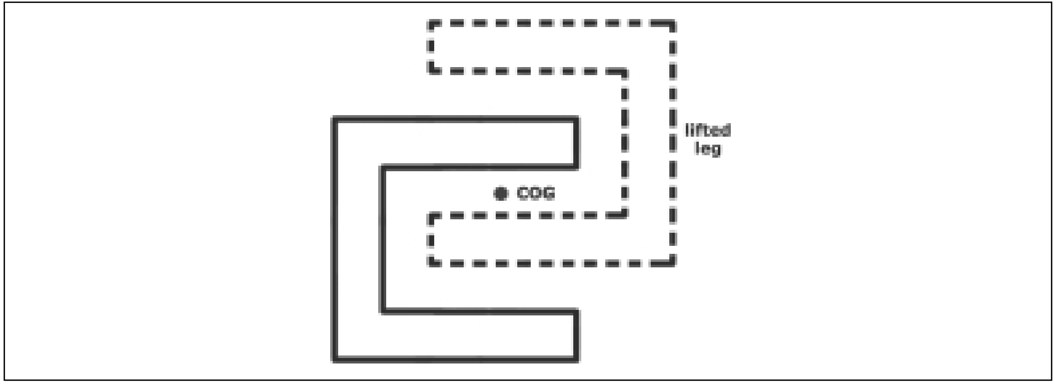


Figure 15.9 A Two-Legged Robot with Interlaced Legs



Though this works, it's a bit like cheating! If you want to emulate the way we human beings walk, you must understand what happens in the human body. Let's do a simple experiment. Stand still, being sure to distribute your weight evenly over your feet. Keep your arms at your side and keep all your muscles relaxed. Now slowly try and lift one leg: your body tends to fall to that side. While walking under normal conditions, you unwittingly move your COG over one foot before lifting the other. This gives you balance and stability and prevents you from falling.

This is the behavior you have to replicate to build a true biped robot. You have to shift its COG over one foot before lifting and advancing the other (Figures 15.10 and 15.11).

Figure 15.10 A Biped Robot Standing

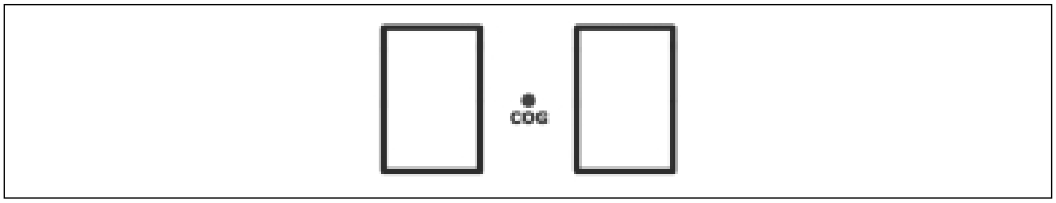
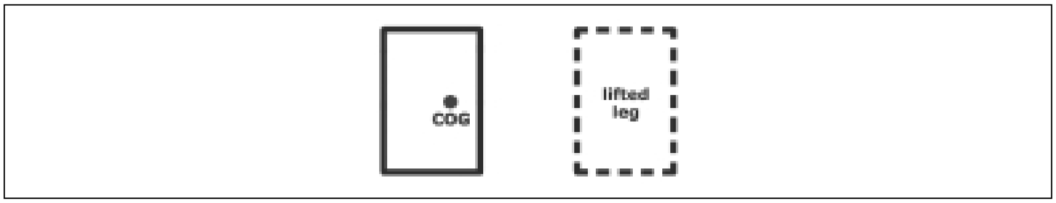


Figure 15.11 A Biped Robot Shifts Its COG over One Foot before Lifting the Other



Actually, the way human beings and animals walk follows not only the rules of statics but also those of dynamics, the branch of physics which deals with matter in motion. When a man runs, for example, he is in dynamic balance, producing forces that oppose gravity and temporarily violate the rules of statics. To understand how this happens, you can study how *you* walk, and also look carefully at how animals phase their walking (bipeds, four-legged animals, insects, and arachnids). For example, elephants and other very large animals only lift one leg when walking slowly, to keep the static COG within the triangle bounded by their remaining legs. Once the pace picks up, the opposing gait takes over, which is similar to the sequence we described in Figures 15.6 and 15.7. Most four-legged animals use this scheme when trotting. At further increases of speed, like in galloping, dynamic stability is more important than static: only one leg needs to contact the ground, and this allows the animal to cover more ground with every cycle.

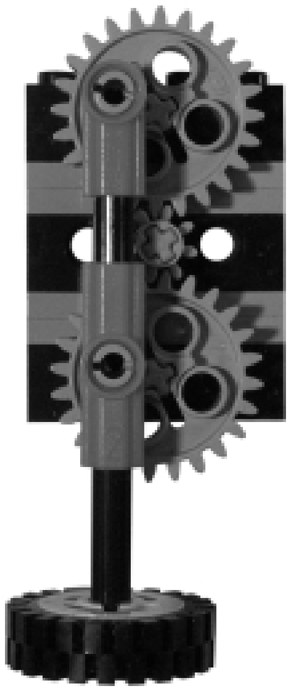
Building a robot that walks or runs using dynamic balance is a very complicated task, and for this reason in this chapter we will stay inside the comforting walls of statics.

Building Legs

Whatever kind of walking robot you're going to build, you must find a way to convert the rotary motion provided by the electric motors into the proper sequence of movements necessary for a leg to work. Animals and human beings use a very complex geometry operated by an impressive number of independent muscles. You must stick to the constraints imposed by the MINDSTORMS system, thus finding simpler solutions.

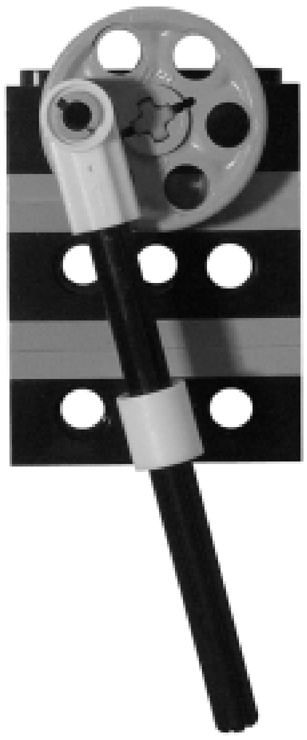
Figure 15.12 illustrates an initial idea: a leg mounted on two gear wheels of the same size, which are then connected in phase through a third gear. It's very important that the leg attaches to two corresponding holes of the gears, otherwise it won't work because the holes will change their spacing as the gears turn

Figure 15.12 This Leg Always Remains Vertical and Follows a Circumference



By driving any of the three gears, this simple leg will go up and down, forward and back, always in a circle. The leg always remains vertical. Figure 15.13 shows a slightly different approach, where only one point of the leg is attached to the wheel, and the leg itself slides freely into a rotating support (fulcrum).

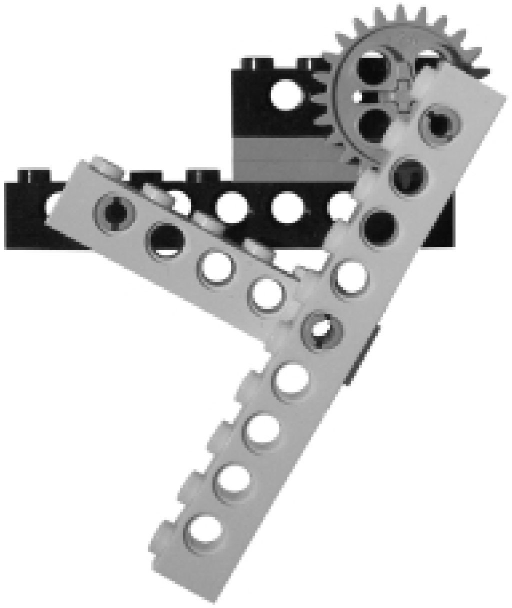
Figure 15.13 This Leg Describes an Ellipse



In this assembly, the terminal point of the leg describes an ellipse—a flattened circle—whose height is equal to the distance between the uppermost and lowermost positions of the point where the leg is attached to the wheel, and whose length is a function of the distance between the fulcrum and the wheel. The closer the fulcrum to the wheel, the longer the ellipse and, consequently, the stride of the leg. You can adjust this distance to make your robot take longer or shorter steps, affecting its speed. We invite you to experiment with this setup, changing the distance between the wheel and the fulcrum, to understand the effect on the stride. Later in the chapter, we'll use this feature to provide a legged robot with turning ability.

More complex leg geometries are also possible (see Figure 15.14). Designing legs is almost an art—it requires good intuition, and a lot of patience to test and improve your initial idea.

Figure 15.14 A Leg with a More Complex Geometry



Building a Four-Legged Robot

Let's start by building a robot with four legs in order to demonstrate the center of gravity principle explained in Figures 15.5 to 15.7. The architecture is very simple, and symmetrical: keep the COG as close as possible to the center (Figure 15.15). We built it solely from RIS parts.

Removing the RCX, you'll notice there's a single motor which, through two worm gears, provides motion to the front and rear leg assemblies (Figures 15.16 and 15.17). The other thing to notice is the phase of the legs: they are diagonally paired. The front left goes together with the rear right, while the front right accompanies the rear left, which implements the walking scheme shown in Figures 15.6 and 15.7.

Figure 15.15 Our Four-Legged Robot

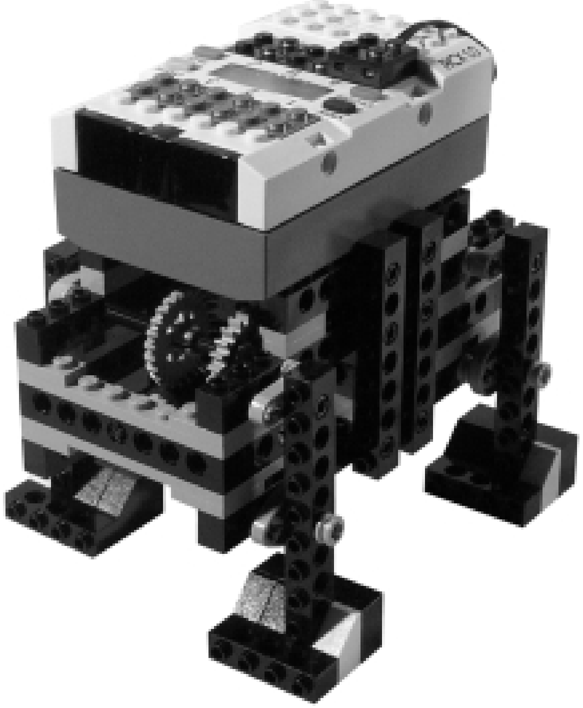


Figure 15.16 Top View (RCX Removed)

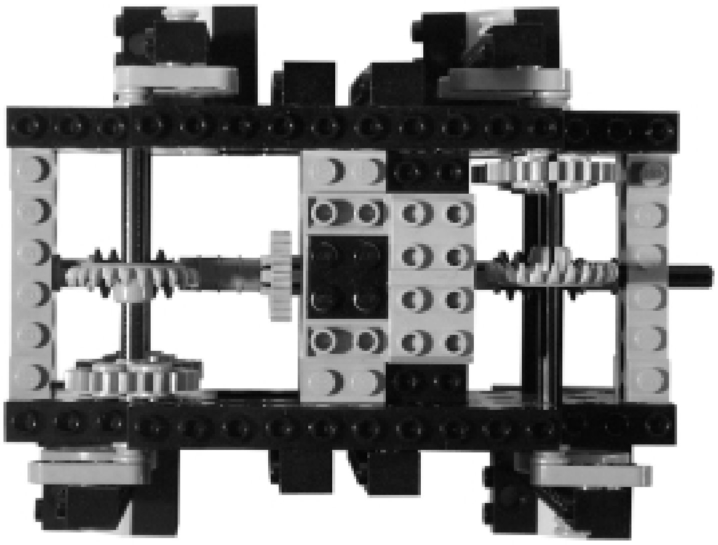
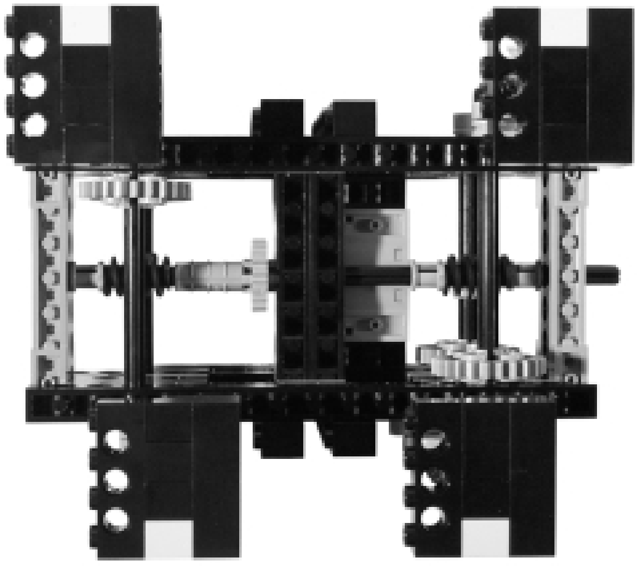


Figure 15.17 Bottom View



The legs follow the scheme of Figure 15.12, where just the gear wheels are inside the robot, their axles mounted on short 1 x 3 liftarms to which the legs are connected (Figure 15.18). We also used four cams because the MINDSTORMS kit includes only four liftarms.

When the robot walks, it lifts two legs diagonally opposed, while standing on the other two (Figure 15.19). Even when moving the legs, this robot always remains symmetrical, thus its COG doesn't change position.

There's actually not much this robot can do. It's easy to build, somewhat instructive, but it's only able to go straight forward or backward. You can mount two front and rear bumpers to make it reverse direction, but that's all you can expect from it. To provide your robot with directional control, unlocking all the opportunities that navigation affords, you need further sophistication. Let's move on and discuss some more challenging projects.

Figure 15.18 The Front Left Leg

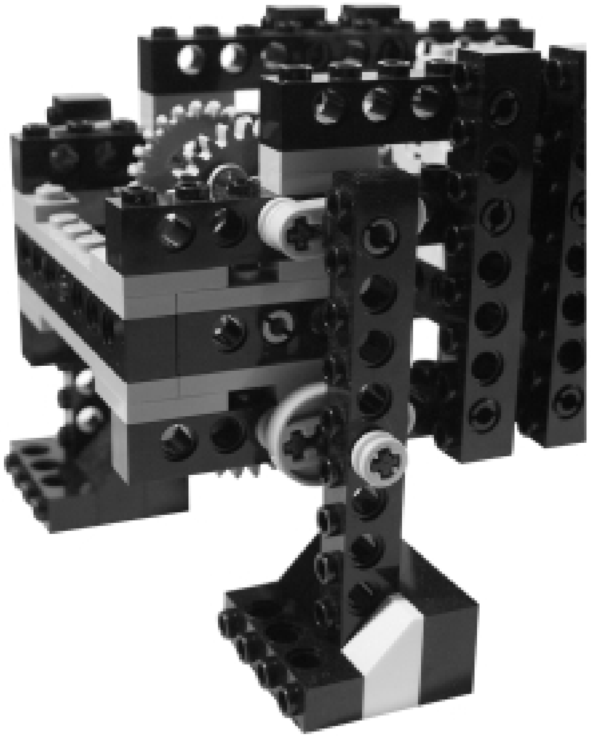
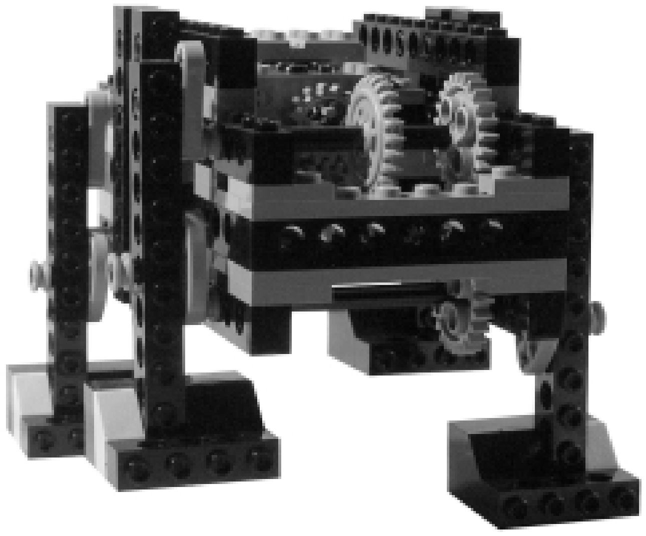


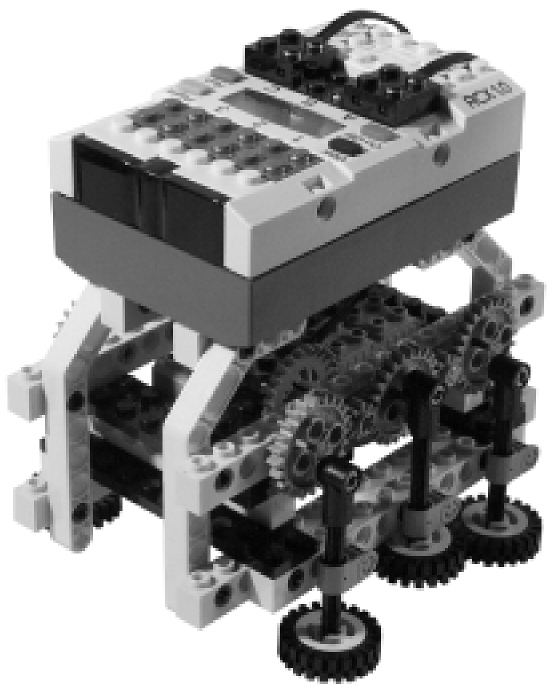
Figure 15.19 Front View, the Robot Stands on Two Legs



Building a Six-Legged Steering Robot

By increasing the number of legs, you can easily make a steering walker. The robot shown in Figure 15.20 (MINDSTORMS parts plus 2 x 24t gears) has six legs similar to that in Figure 15.13.

Figure 15.20 A Simple Six-Legged Robot



The left and right leg groups are powered by two independent motors (Figure 15.21), and in each group the wheels are phased so as to have the mid one raised when the front and rear legs are lowered (Figure 15.22).

This robot turns and walks. You can make it turn by stopping or reversing one of the motors as if it were a skid steer drive. But it's affected by a serious problem: stability. The two groups of legs are not synchronized. Because of this, only the central legs are down at certain times. Since two legs are not enough for a stable balance, the robot tilts forward or backward a bit, ensuring the additional legs make contact. As a result, its walking is rather irregular and jolting.

Figure 15.21 Top View (RCX Removed)

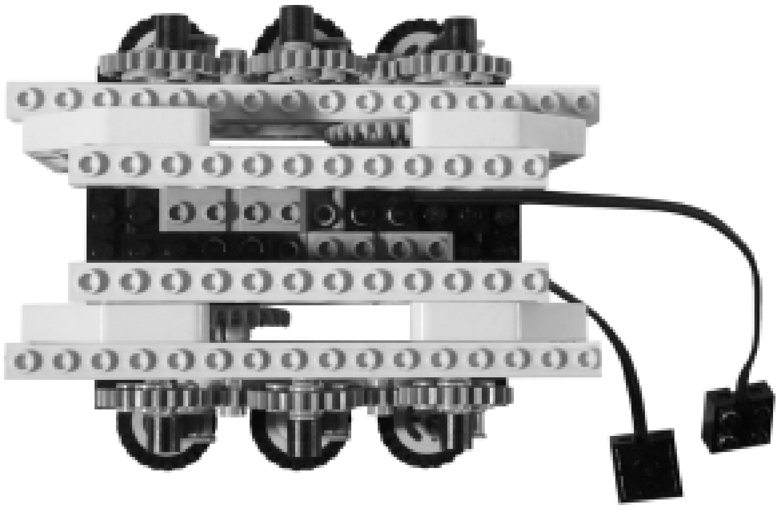
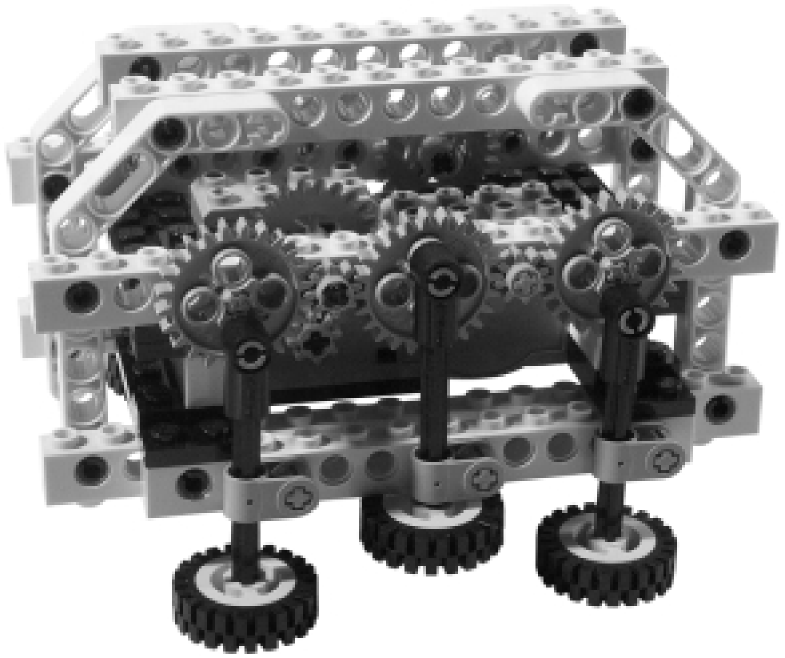


Figure 15.22 The Left Leg Group

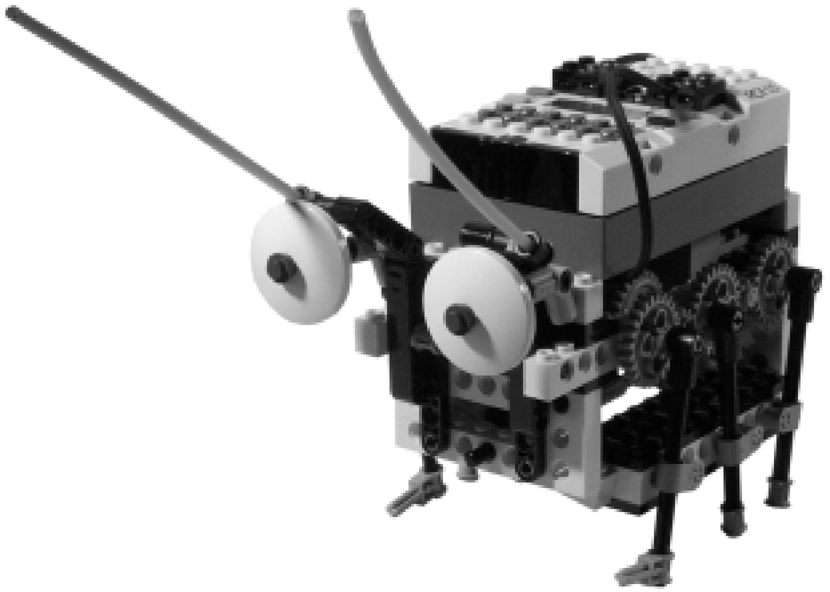


What could you do to smooth the walking motion? Using two sensors to detect the position of the legs, you could keep the two groups in sync so that one side goes on the mid leg only when the other one has two legs down.

There is another approach, more on the hardware side, that requires you to vary the geometry of the legs to make them change their stride. The left and right leg groups are connected together, and powered by a single motor, so that the robot is always supported by a triangle like that shown in Figure 15.8. To change the stride of the legs, you have to change the distance of their fulcrums from the gear wheels.

The robot in Figure 15.23 uses this technique. (You can replicate it using MINDSTORMS parts plus 2 x 24t gears.)

Figure 15.23 A More Sophisticated Steering Walker



All the legs are powered by a single motor, while the second motor controls the leg geometry (Figure 15.24).

It's crucial that you connect the six legs in phase, so that each side has the mid leg raised when the other two are down, and the left side has the mid leg down while the right one has its mid one up (see Figure 15.25).

The fulcrums of the legs are attached to a swinging chassis that the second motor can incline on one side or the other (Figure 15.26). The stride of the legs becomes shorter at the side where the fulcrums have been lowered, and longer at the other, thus making the robot turn.

Figure 15.24 Top View (RCX Removed)

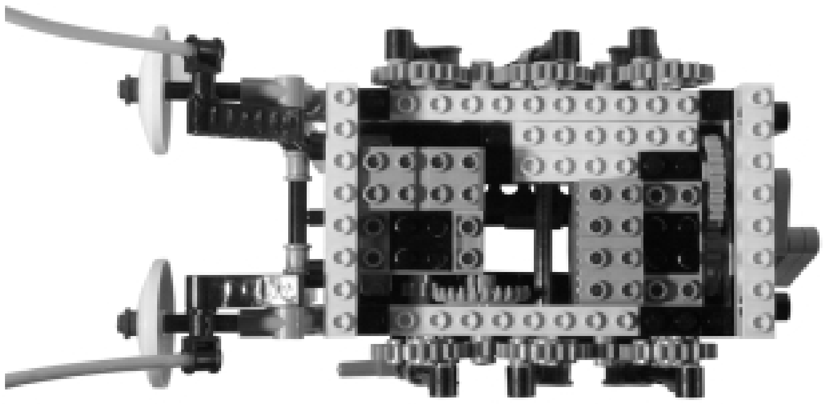


Figure 15.25 Three Legs Are Always in Contact with the Ground (Side View)

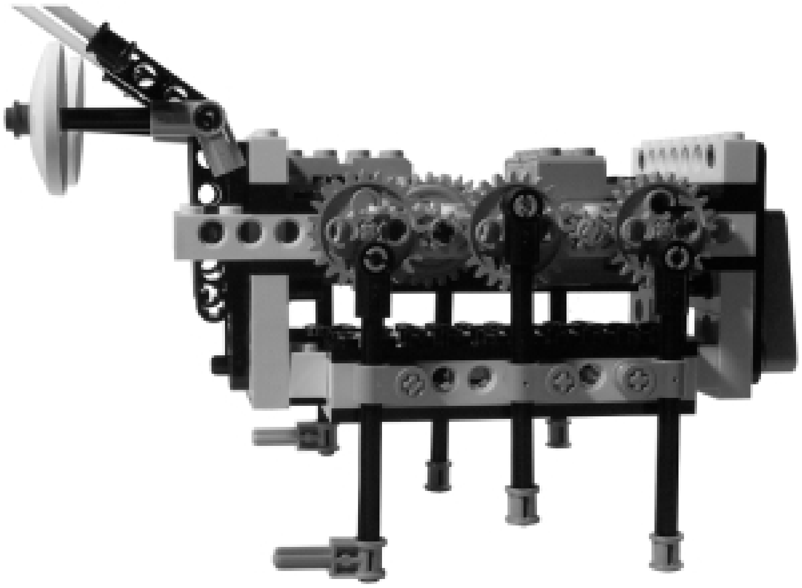
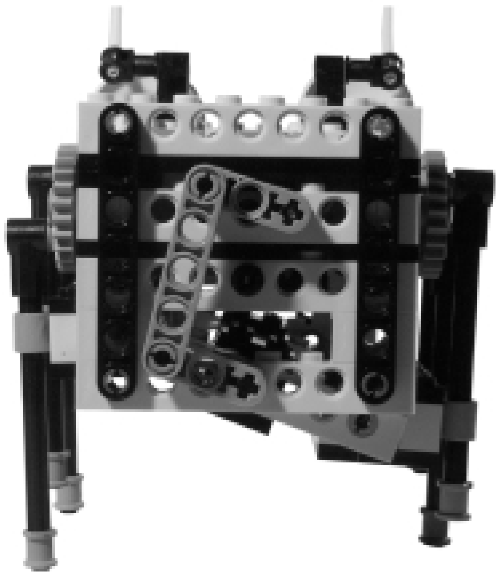
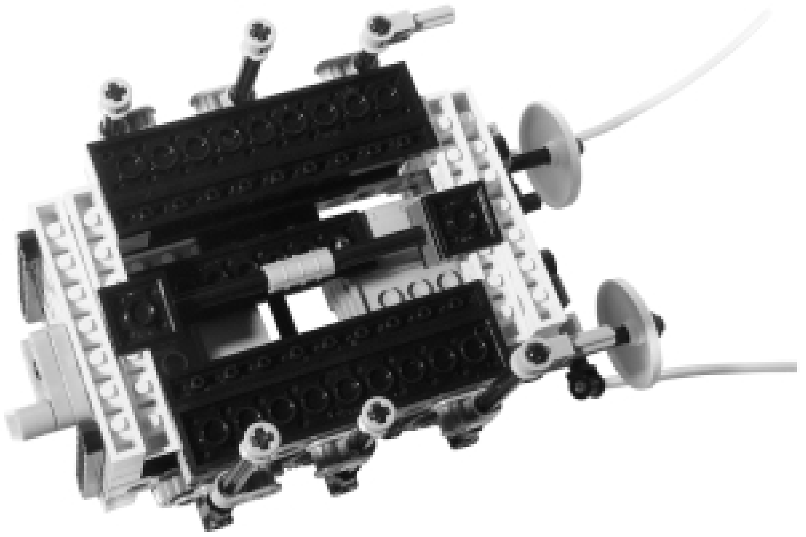


Figure 15.26 Rear View



The front and rear sides of the swinging chassis are operated through a long joined axle and two 1 x 2 bricks with an axle hole (Figure 15.27).

Figure 15.27 Bottom View

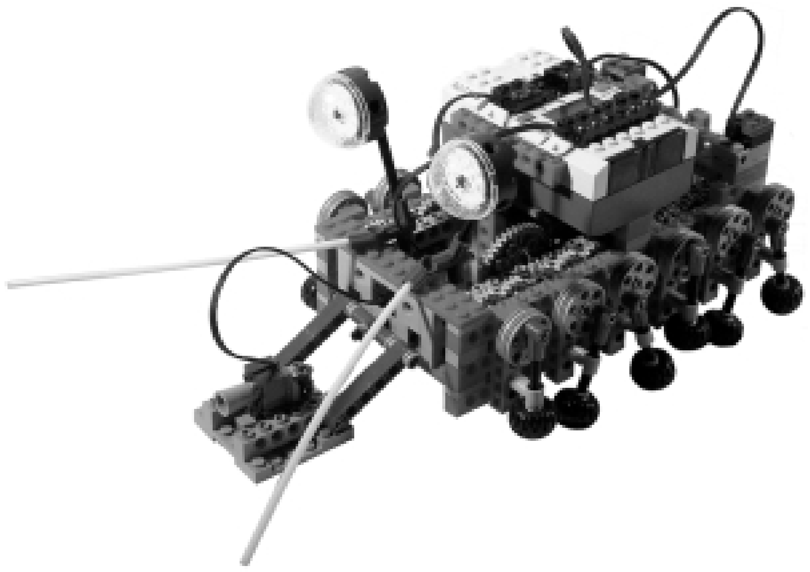


This robot needs no sensors to control its motion. When you want to make it turn, switch the motor on for a few seconds to change the geometry, then brake it to hold it in position. Recall that floating the motor might allow the swing chassis to turn. To resume straight motion, let the motor float and the swinging chassis will return to its central positions after a few steps.

The limit of this architecture is that the robot will turn only with a very large radius. It will be able to follow a line only if this doesn't make tight angles.

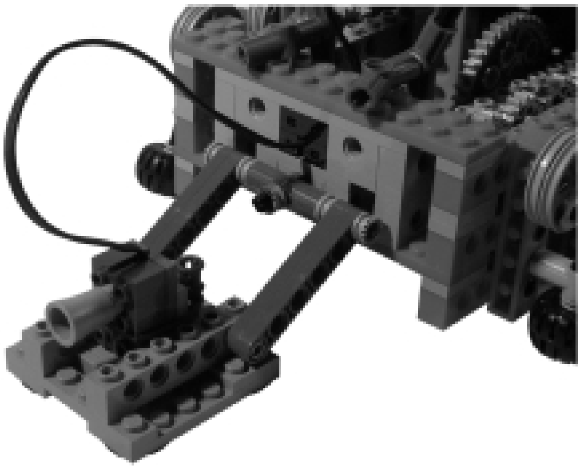
To give high maneuverability to your robot, you must remain with a skid-steer type drive, possibly increasing the number of legs to improve stability. We designed a 12-legged walker to attend a line following contest in Rome. Being that always had three legs down at each side it was very stable, and was able to follow the black line through right angles, too (see Figure 15.28).

Figure 15.28 Dodi, Our 12-Legged Line Follower



To minimize interference from ambient light, and to be sure the light sensor was always at the same distance from the line, we placed the sensor on a sort of sled pushed by the robot (see Figure 15.29).

Figure 15.29 The Light Sensor Was Always at the Same Height from the Ground



Designing Bipeds

Biped robots are among the most challenging projects you can ever face. In a biped, the position of any single part, any single gram of mass, is critical to a stable balance. If you replicate the designs that follow, you will see that all of them walk very smoothly, but you will discover also that you can't add additional parts anywhere in their body and not feel the pain!

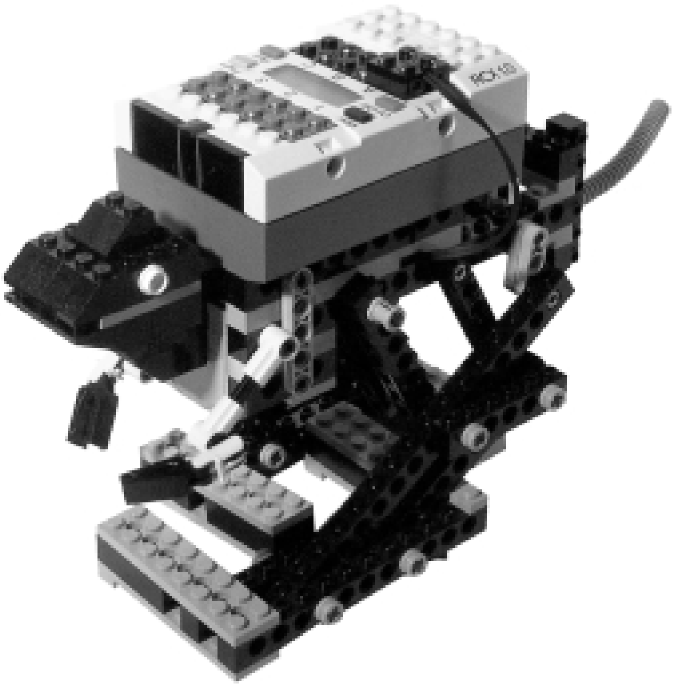
We will go through the approaches described in the The Theory Behind Walking section at the beginning of the chapter: interlacing legs (Figure 15.9) and COG shifting (Figure 15.11). For this latter category, we will explore two techniques: The first relies on an independent mechanism which moves a mass from one side of the robot to the other to change the position of its COG, while the second requires that the whole body of the robot bend at the ankles to move the COG over a foot.

At the end of the section, we will give you some tips about the next step in the challenge: the making of a biped robot capable of turning!

Interlacing Legs

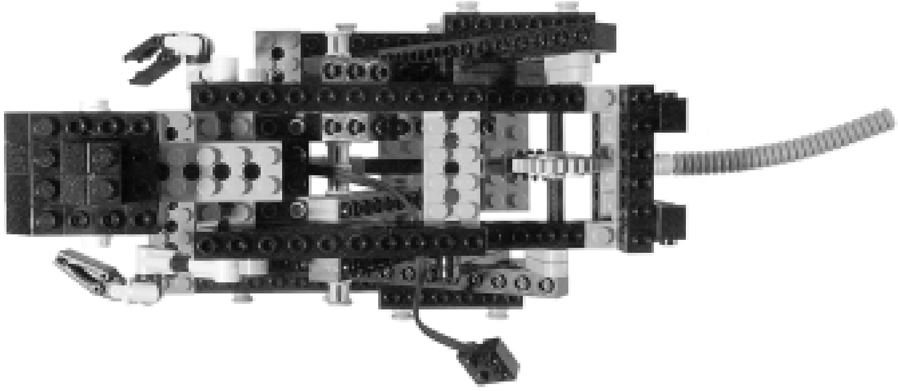
Let's start with a biped based on the technique shown in Figure 15.9 using interlacing legs. The feet must be U-shaped and large enough to support the weight of the whole robot (Figure 15.30, MINDSTORMS parts only).

Figure 15.30 A Biped with Interlacing Legs



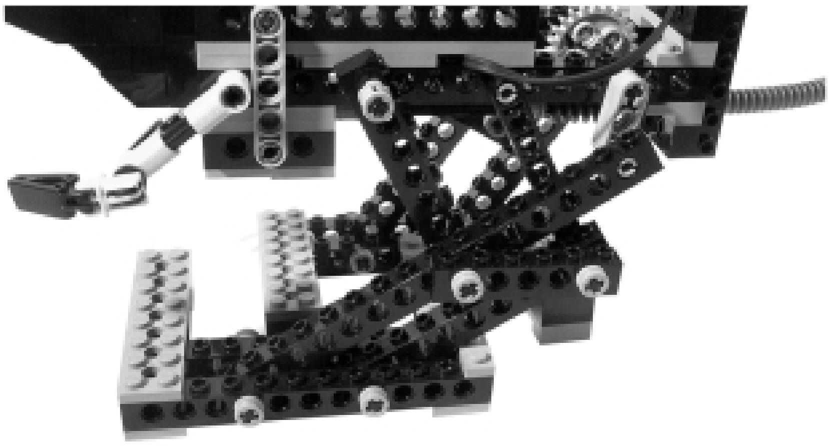
This robot uses a simple gearing, only a 24t and a worm gear. The axle of the 24t connects to two opposed liftarms that operate the legs. The motor shaft has been prolonged with a 12 axle that juts out from the back and to which we attached a short decorative tail (Figure 15.31).

Figure 15.31 Top View (RCX Removed)



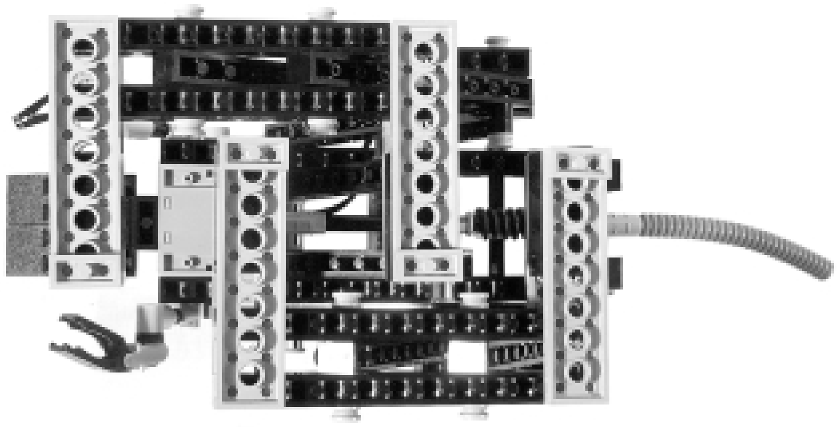
The leg geometry is very similar to the one in Figure 15.14, but here we used a double series of parallel beams in order to form two connected parallelograms, an upper (body to knee) and a lower (knee to foot) one (Figure 15.32). Making this allows the foot to always remain parallel to the body, and thus the body always parallel to the ground.

Figure 15.32 The Left Leg



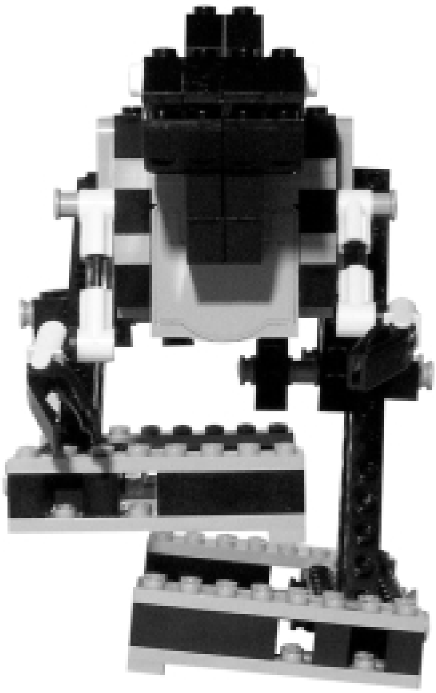
Looking at the robot from the bottom, it's easy to see how the feet interlace with each other (Figure 15.33).

Figure 15.33 Bottom View



The feet aren't really flat on the ground. We placed a 1 x 2 plate under each 2 x 8 plate to keep the inner part of the feet a bit raised. This compensates for the slackness of the leg that otherwise would make the robot lean at the side of the lifted leg, causing the COG to move beyond the base and make the robot fall (Figure 15.34).

Figure 15.34 Front View

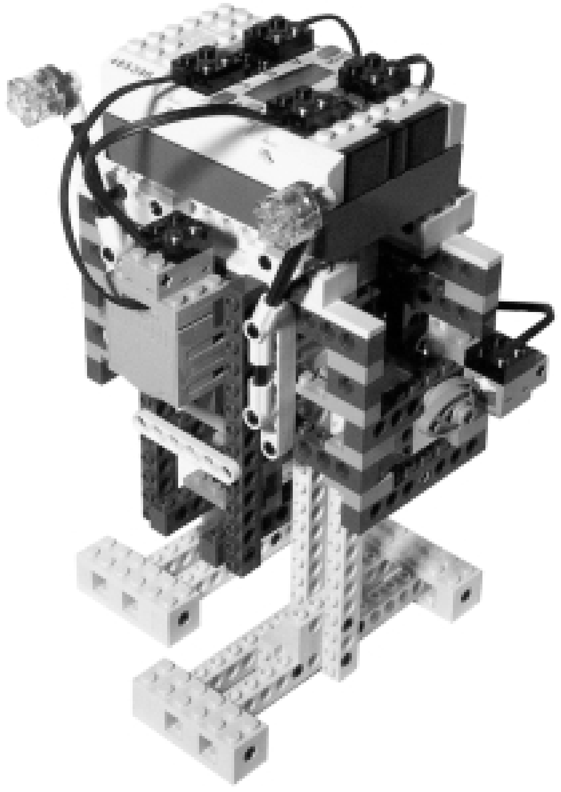


Don't forget to add some decorative parts to your walker! We used some of the parts left over in the MINDSTORMS box to provide the robot with a dinosaur-like appearance. In Figures 15.30 and 15.31, you can clearly distinguish its fierce-looking head and the short front legs.

COG Shifting

At the beginning of 1999, we built our first COG-shifting biped, S6, challenged by the widespread belief that a COG shifting robot was very hard, if not impossible, to build with LEGO. Though not an exact replica of S6, the robot in Figure 15.35 works on the same principle. (It requires some extra beams and liftarms of various sizes, along with four 1 x 8 tiles.)

Figure 15.35 A COG-Shifting Biped



Though the feet are quite large and solid, they are well-separated in the middle (by a distance of two studs). The RCX is installed on a sled, and as the robot walks it moves side to side to act as ballast. Before lifting a leg, the robot shifts the RCX over the opposite leg, and doing so moves its COG over the base of a single foot (see Figure 15.36).

The legs are operated using the same geometry employed in Figure 15.12, but this time realized in a more solid design. In Figure 15.37, notice that there's a touch sensor on the side of the robot, closed by a pair of opposing cams when the feet are both on the ground. A second touch sensor, located at the top front of the robot, detects the movements of the sled: Two black pegs close it when the sled is either at its left or right limit.

Figure 15.36 The Robot Standing on One Leg

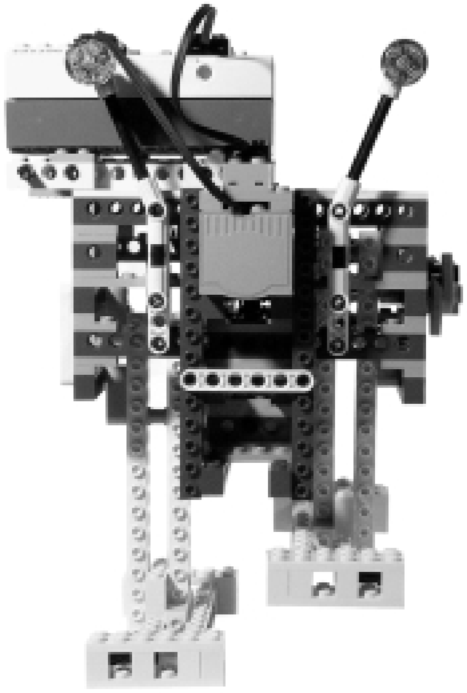
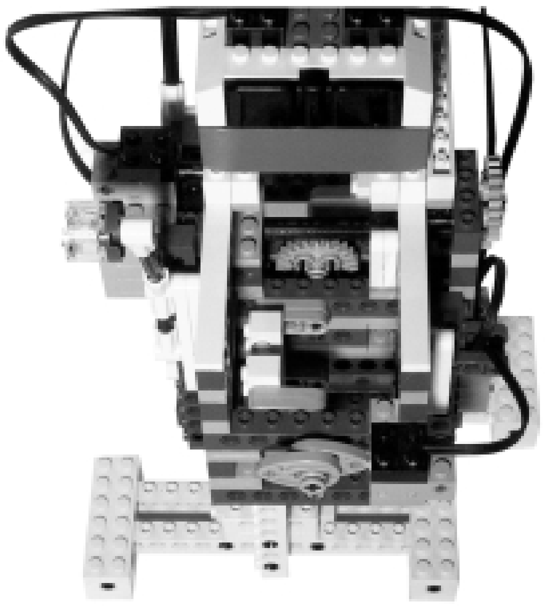


Figure 15.37 Left Side View



In the walkers seen so far there have been no need for sensors, but this robot relies on them to keep its synchronization. Starting with both legs down with the right one advanced, the sequence is:

1. Slide right the RCX until the sled sensor closes.
2. Advance the legs. Their sensor was closed, wait for it to open and close again.
3. Slide left the RCX until the sensor closes.
4. Advance the legs again.

The sled with the RCX moves through a (upside-down) rack and pinion assembly (Figure 15.38), while the legs use a second motor located at the very bottom of the robot (Figure 15.39).

Figure 15.38 Rear View

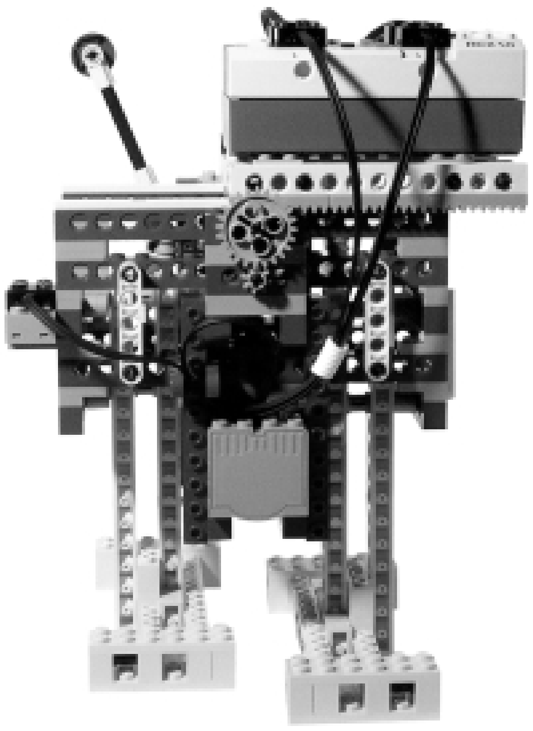
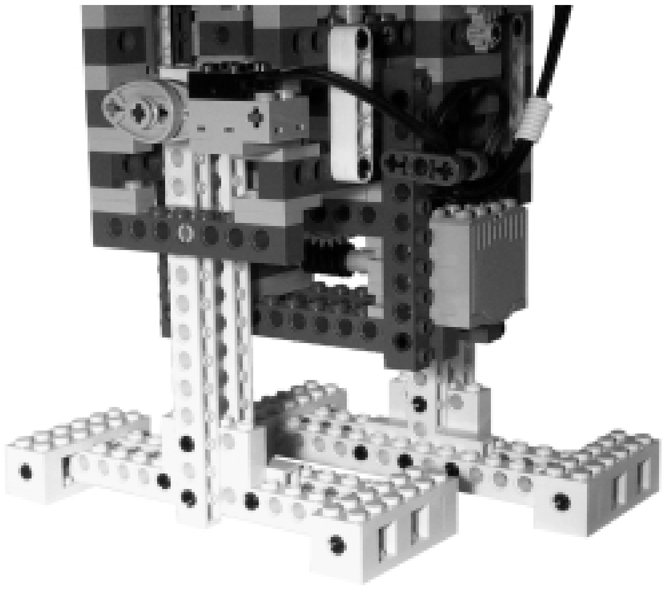


Figure 15.39 Detail of the Motor that Operates the Legs



Ankle Bending

There's another, completely different approach to shifting the COG of a walker: bending its ankle sideways in order to carry the COG over the foot. The robot in Figure 15.40 uses this technique. You'll notice that the right leg inclines outside and that the RCX rests over the foot.

We used the leg designed by Miguel Agullo for his very nice Hammerhead. The key component of the ankle is a *crankshaft* that manages the bending of the leg over the foot (Figure 15.41). It looks pretty funny as it walks, lurching from side to side.

The hips are free to swing back and forth, and their supporting axle serves to also transfer motion to the ankle with a technique similar to what we described for the making of synchro drives or front-wheel drive cars (Figure 15.42). A second axle at the rear provides motion to the legs, through two crankshafts and two liftarms.

Figure 15.40 An Ankle-Bending Walker

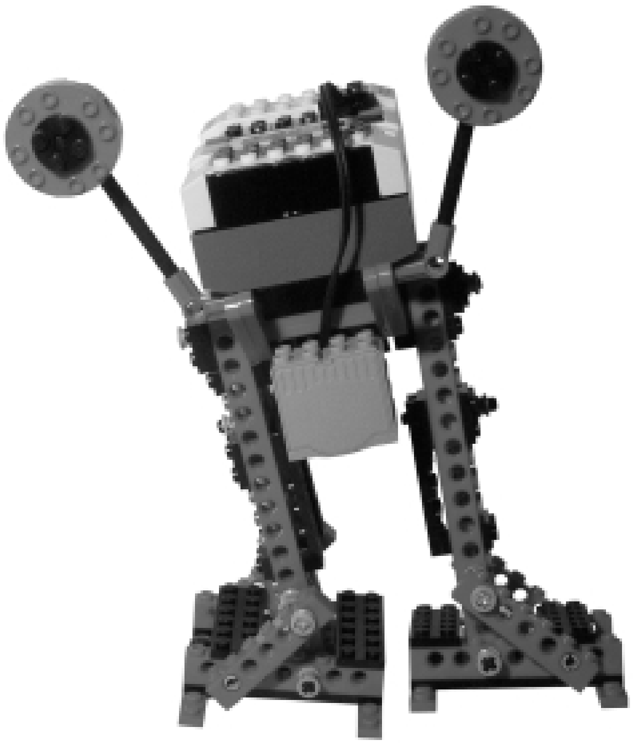


Figure 15.41 Detail of One Foot

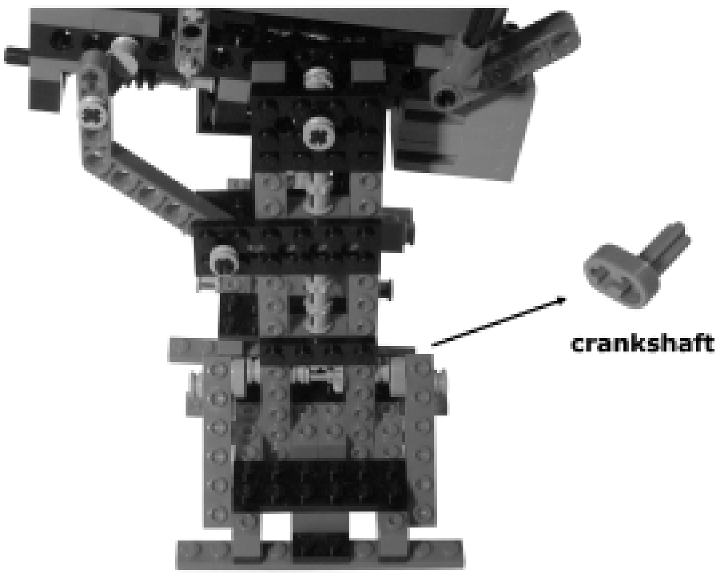
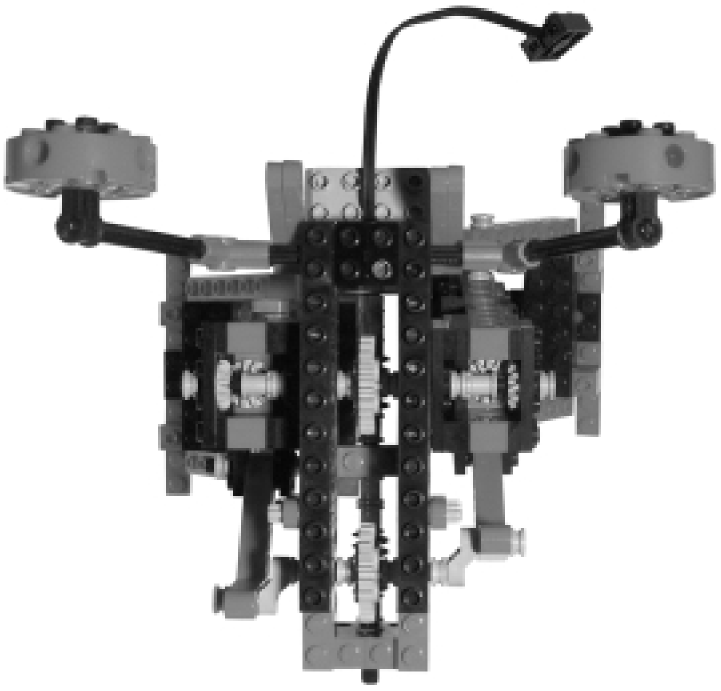


Figure 15.42 Top View (RCX Removed)



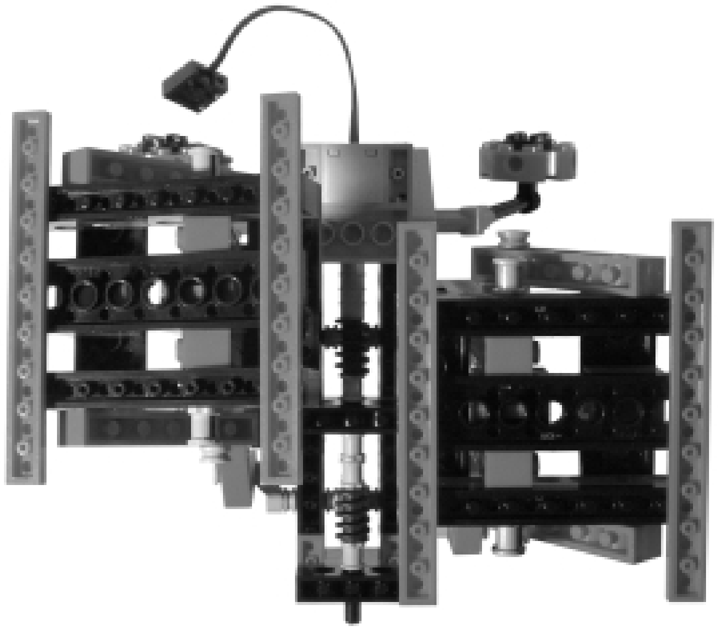
This walker uses a single motor and two worm gear and 24t pairs (Figure 15.43).

The difficulty of this model lies in finding the proper synchronization between ankle and leg movements. If you decide to give this technique a try, we suggest you follow Miguel's detailed instructions on his site (see Appendix A) to replicate his geometry.

WARNING

This ankle-bending model is not completely symmetrical, and can walk forward only. Its COG lies in front of the hip joint, so the robot tends to lean forward and transfer its weight from one leg to the other that is advancing. If you run it backward, it will fall.

Figure 15.43 Bottom View



Making Bipeds Turn

Is it possible to make a biped robot turn? It is, though it's definitely not an easy task. Once again we invite you to experiment for yourself to find a working strategy. Observe your feet while you walk slowly, taking short steps and going straight: One foot is ahead of the other, but they remain parallel, as if they were running on tracks. Now try to change direction, but only take the very first step. If you look at your feet again, you notice that they are no longer parallel: the foot that's ahead is pointing in the new direction.

How can you emulate this behavior in your robot? By using an architecture similar to that of Figure 15.34, but jointed in the middle so the legs can slightly converge or diverge. You will need a third motor to control the parallelism between the legs, and probably a third sensor to detect the straight position.

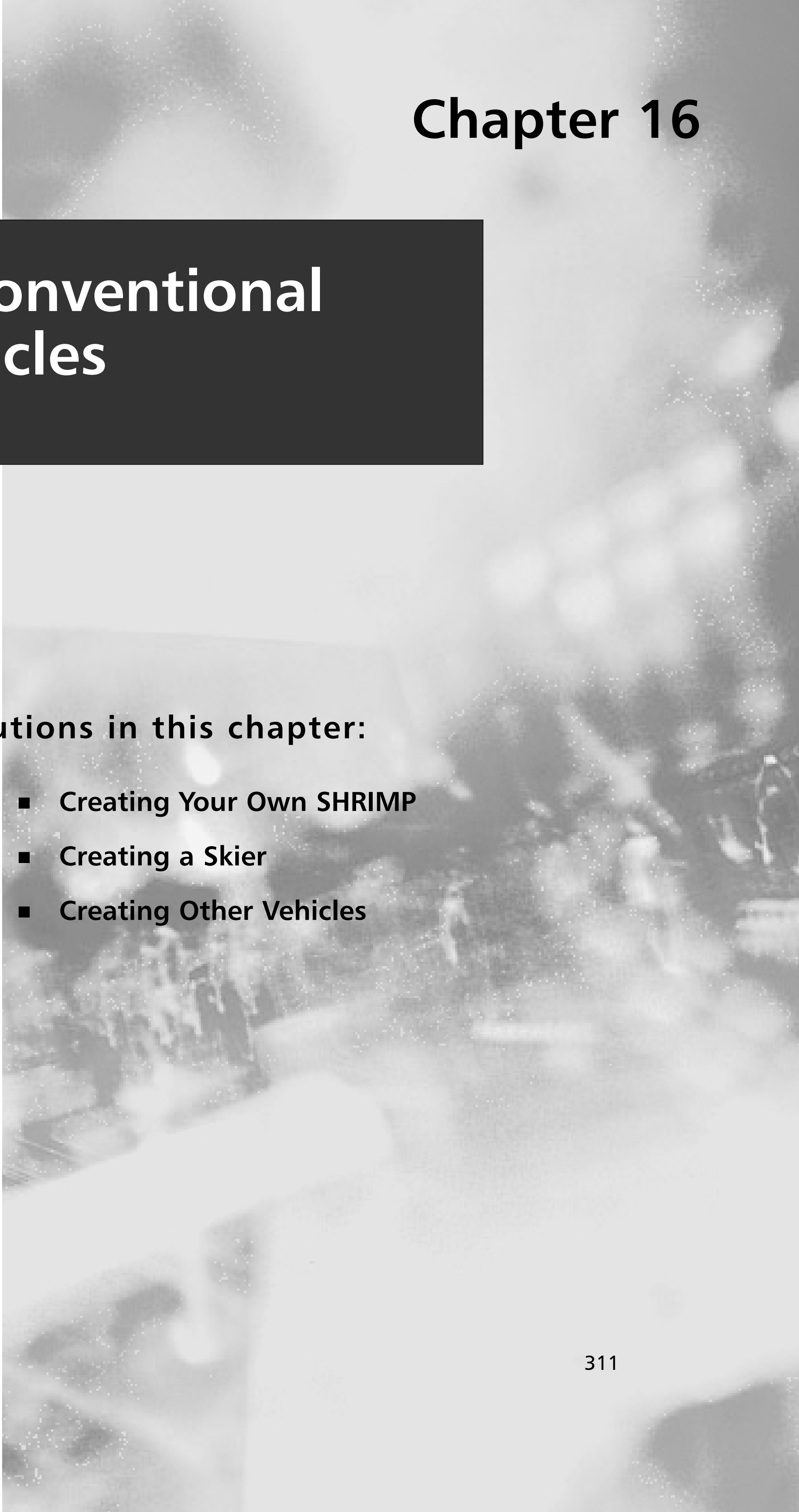
The list of Web sites in Appendix A includes resources that will help you in investigating this technique.

Summary

In this long chapter, we covered, hopefully, all the most important aspects about the making of robotic walkers. Along the way we discussed some important concepts, such as the *center of gravity*, that will prove useful in many other applications.

If you had the impression that we talked a lot about mechanics and not much about software, you're right. The task of balancing the weight over the legs is by itself so demanding that not much space remains to make your robot perform other actions. Although we showed some possible basic behaviors like line following, more complex tasks like grabbing objects are usually beyond the scope of walking robots due to the changes brought about regarding their delicate balance. Precise navigation is also not very suited to walkers, because the natural tendency of their legs to skid a bit on the floor makes them somewhat unreliable for positioning.

However, all this shouldn't keep you from experimenting with walkers. The pure reward of seeing them move compensates for all the effort put into building them. And who knows, in your enthusiasm you could develop some new solutions, or maybe design something as complex as a *running* robot!



Chapter 16

Unconventional Vehicles

Solutions in this chapter:

- **Creating Your Own SHRIMP**
- **Creating a Skier**
- **Creating Other Vehicles**

Introduction

In the previous chapters, we discussed the two most common propulsion systems, wheels and legs, and looked into the many details regarding possible implementation schemes. Actually we didn't exhaust all the possibilities, so we are going to describe a few more robotic vehicles suited to very special tasks.

What mainly affects the mobility of a vehicle is the nature of the terrain that it has to move over. The scale of the robot, also, has a strong influence on the size of the obstacles it can overcome: A pebble two inches high is nothing for a wheel 20 inches in diameter, but it's insurmountable for a differential drive with wheels of only 3.5 inches (the largest contained in the MINDSTORMS kit). Scaling your robot up, however, is not always a practical option. In the specific case of LEGO, you're limited to the size of the available parts and their mechanical properties, and just like with real-life robots, they, too, often face constraints when it comes to weight and size.

The two robots of this chapter, a SHRIMP rover and a skier, are completely different in nature, but share the fact that they are designed for special surfaces: rough terrain for the first model, and snow for the second.

Creating Your Own SHRIMP

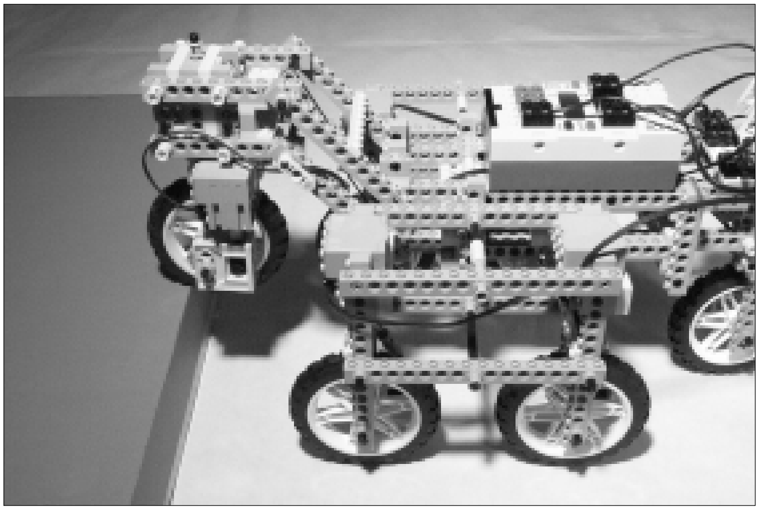
The original SHRIMP is a high-mobility wheeled rover designed by the Autonomous Systems Lab based in Lausanne, Switzerland. It features six wheels powered by independent motors: one front wheel mounted on an articulated fork; one rear wheel directly connected to the body; four wheels mounted on two lateral swinging bogies. (A *bogie* is a wheeled assembly that pivots one or more axles.) It performs amazingly well on many surfaces and against many kinds of obstacles. It's able to overcome obstacles as high as its wheels, even if they take the form of a stairway.

During summer 2000, we built our first LEGO version of SHRIMP, which had capabilities very similar to that of the original robot that inspired its design. The version described here is our first attempt at a turning SHRIMP; the first one, like the original, was not designed to turn.

Unfortunately, this project requires a lot of extra parts: seven motors, six gearboxes, ten universal joints, two polarity switches… not to mention a couple dozen of 1 x 16 beams. Later on in the chapter, we will give you some suggestions to help reduce the requirements, but the project remains rather demanding.

Let's start by looking at the SHRIMP in action to understand how it works. While the first wheel climbs the obstacle (Figure 16.1), the wheel assembly remains vertical, attached to the body with two parallel pairs of beams. This parallelogram geometry is the key to all of the SHRIMP abilities: The beams that connect the wheel assembly to the body convert the push from the other five wheels into vertical lift, while the front wheel itself follows the shape of the obstacle.

Figure 16.1 The SHRIMP Front Wheel Climbs a Step



When the first wheel is up, it's the bogies' turn to climb (Figure 16.2). They rely on the same principle: The bogie is a parallelogram attached to the body in the midpoints of its horizontal sides. When the bogie approaches the obstacle, those horizontal beams act like levers, with their fulcrums on the second wheel of the bogie. The load is applied in the midpoint of the levers, thus the first wheel has to lift only half of the weight applied to the bogie.

In Figure 16.3, the bogies are over the step, and pull up the rear wheel.

SHRIMP has an incredible ability to adapt to very complex terrain configurations. Some of its wheels may descend while others climb. Nevertheless, the body remains stable (Figure 16.4).

To turn properly, that is, with no skidding, the SHRIMP should rotate a minimum of four wheels. Do you remember the rule? If we extend imaginary lines through the axles of all the wheels, *they must meet at a single point.* This would be perfect, but very complex to build and control.

Figure 16.2 The First Wheels of the Bogies Climb the Step

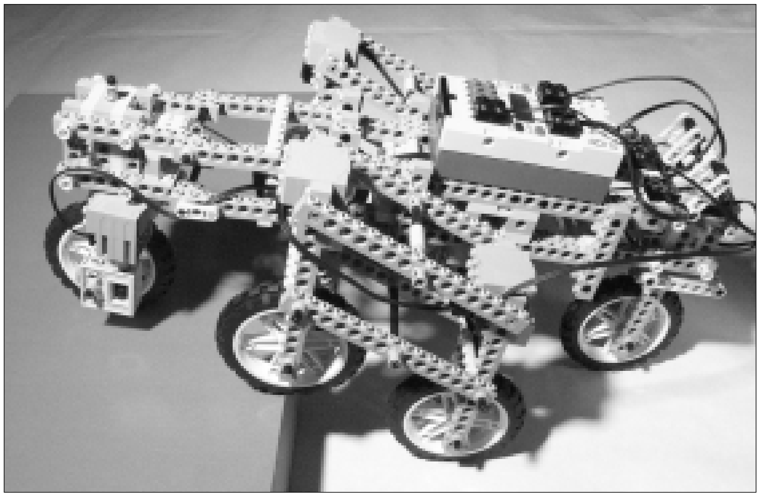


Figure 16.3 The Bogies Are Up and the Rear Wheel Climbs

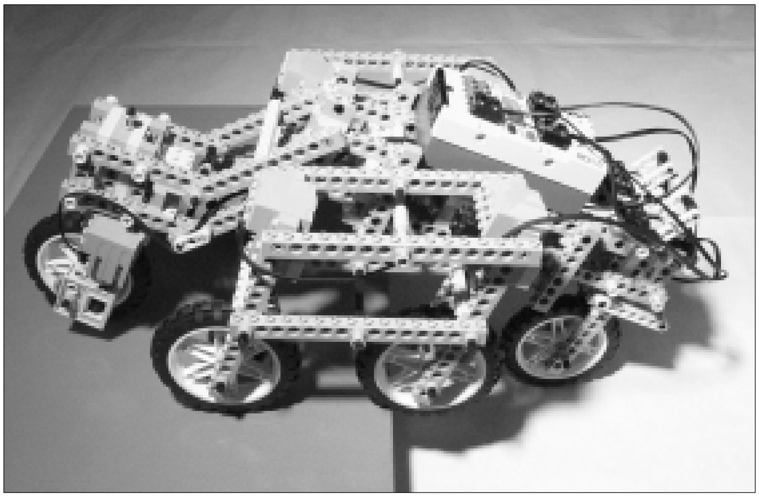
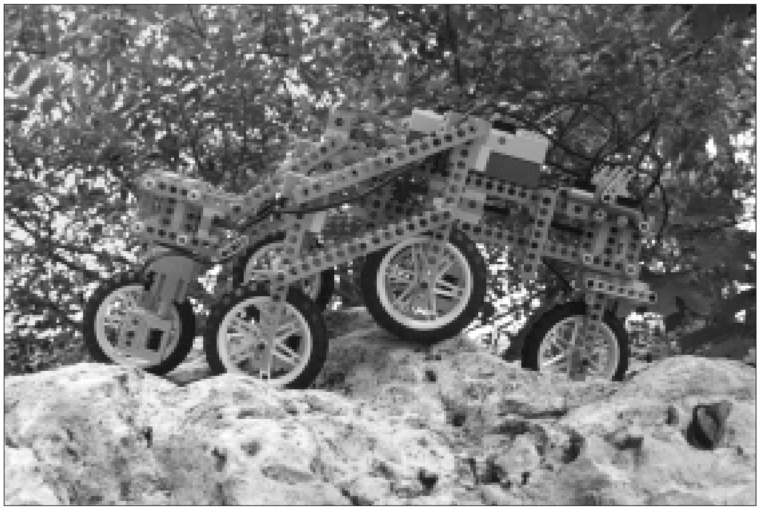


Figure 16.4 The SHRIMP Traversing a Rough Terrain



In our turning SHRIMP, we adopted a simplified scheme: the front and rear wheels turn, while the bogie wheels behave like a skid-steer. In other words, the inner ones stop while the outer ones continue to run. This is an approximate solution that introduces some slippage, but that in practice works very well on most terrains.

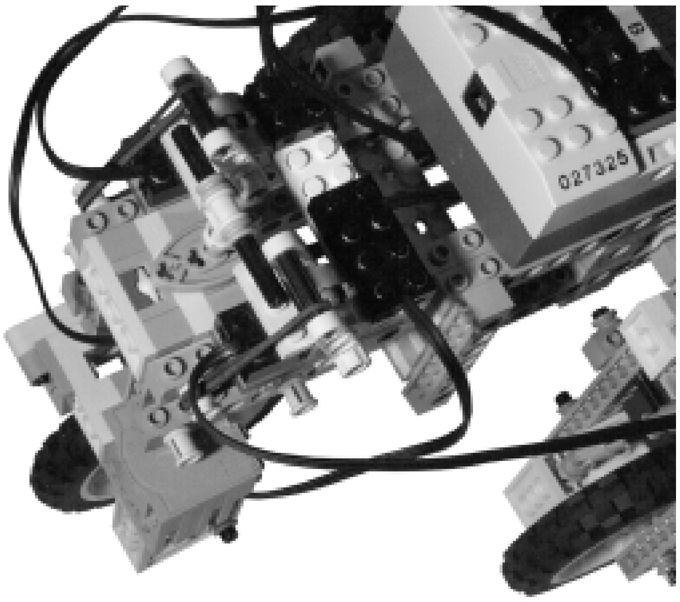
Though we reduced the complexity, there are still too many motors to control for a single RCX:

- The front and rear wheels form a group. Their motors need to always be powered when the robot is in motion.
- A motor turns the front and rear wheels.
- The wheels of the left bogie always run except when the robot turns left.
- The wheels of the right bogie always run except when the robot turns right.

We really wanted to avoid a second RCX, mostly so as not to add further weight to the robot. After long experimenting, and many useful tips from the LUGNET friends, we came out with a solution that saves not only the fourth motor port, but the third one also!

Our design requires two polarity switches, here used as simple on/off switches. The idea is that the steering system controls those switches too, and when the front and rear wheels turn, the inner bogie stops as a result. Figure 16.5 should help to explain the concept.

Figure 16.5 The SHRIMP Steering Control System



Two rubber bands keep the polarity switches gently pulled back in the on position. The pivoting axle of the rear wheel mounts a traverse axle, then, when turned, pushes the inner switch to the off position (the left switch controls the left bogie).

In the same picture, notice the touch sensor that detects the neutral position of the steering system, the only sensor used on this robot.

A single motor operates the entire steering system. It's placed in the bottom part of the body, and connects to the main steering axle through a pulley-belt-worm-24t geartrain (see Figure 16.6).

The main steer axle is a long joined axle that turns both the front and rear wheels. The rear steer assembly is rigidly attached to the body, while the front one forms one side of the parallelogram described previously. For this reason the steer axle requires two universal joints positioned precisely where the swinging beams connect to the body and to the steer assembly (see Figure 16.7).

Figure 16.6 Bottom View: The SHRIMP Steering Motor

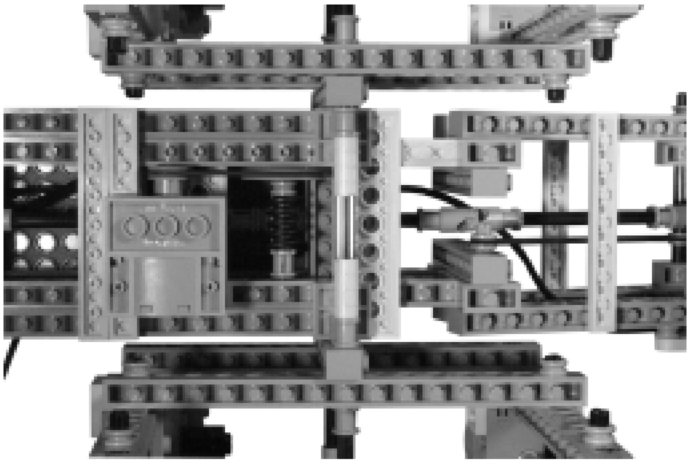
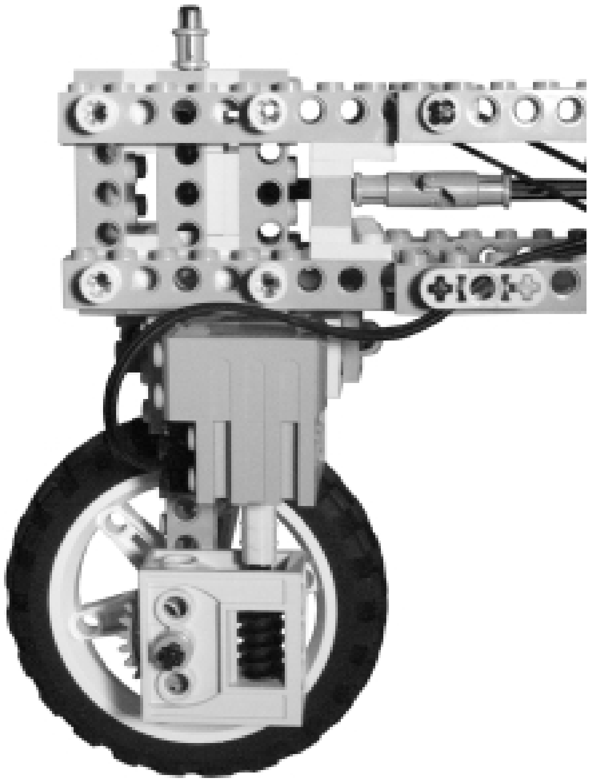
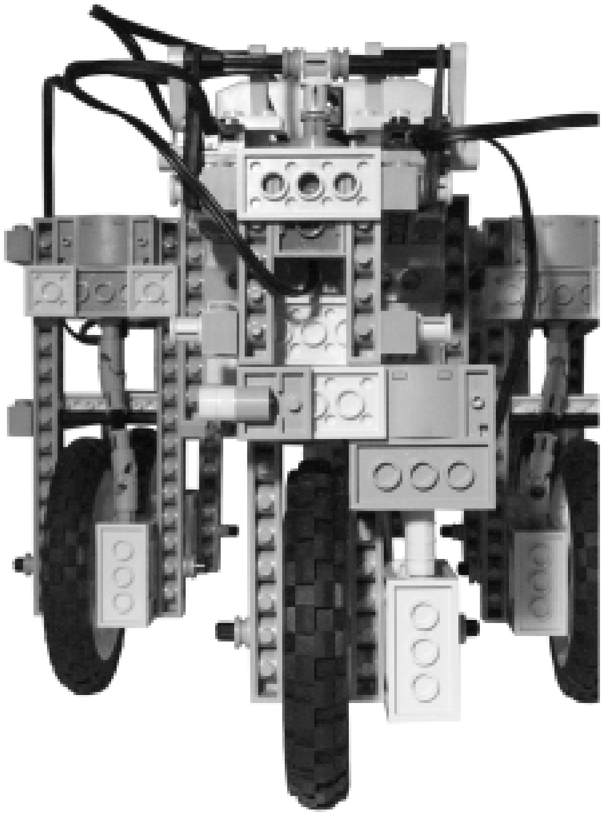


Figure 16.7 The SHRIMP Front Wheel, Side View



The long steer axle ends with bevel gear pairs on both sides, which transfer motion to the pivoting axles (see Figure 16.8).

Figure 16.8 The SHRIMP Rear Wheel, Rear View



The bogies mount four identical wheel groups, where a motor powers the wheel through a joined axle and a gearbox (see Figure 16.9).

In our first SHRIMP, the front and rear wheels were identical to the side ones, while in this version, the motor has been moved down to make the assembly more compact and leave the space above for the steering mechanism (Figure 16.10).

Figure 16.9 Close-Up of a SHRIMP Side Wheel

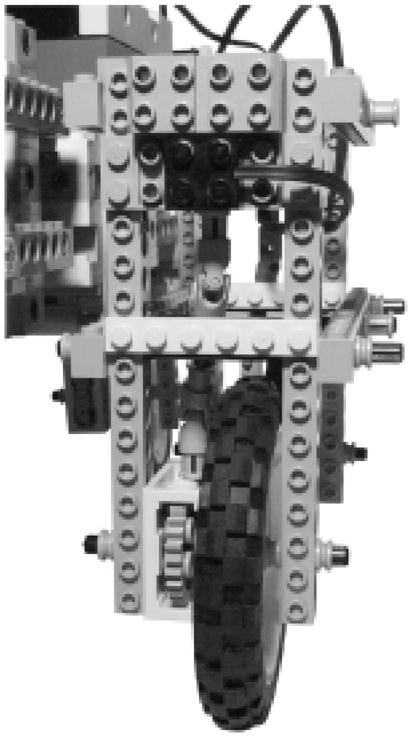
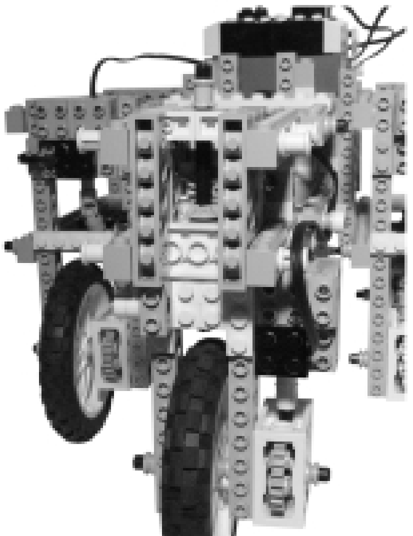
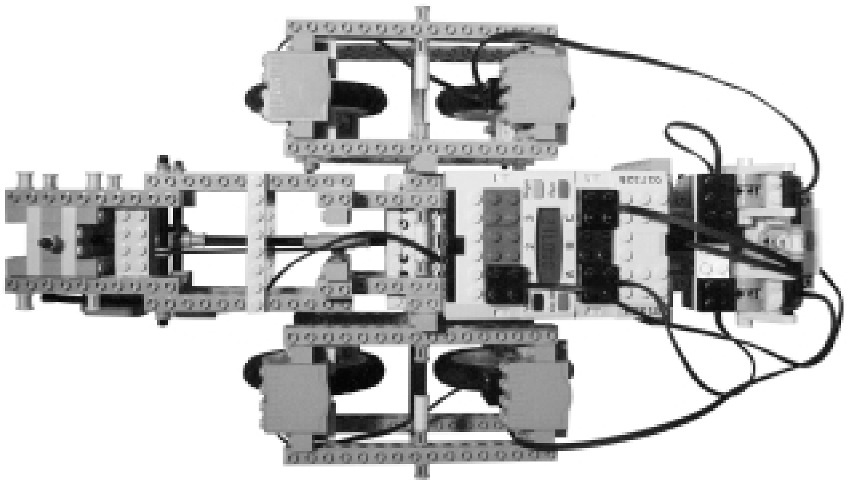


Figure 16.10 The SHRIMP Front Wheel



The RCX stays on the top, just behind the point where the bogies connect to the body (see Figure 16.11).

Figure 16.11 The SHRIMP Top View



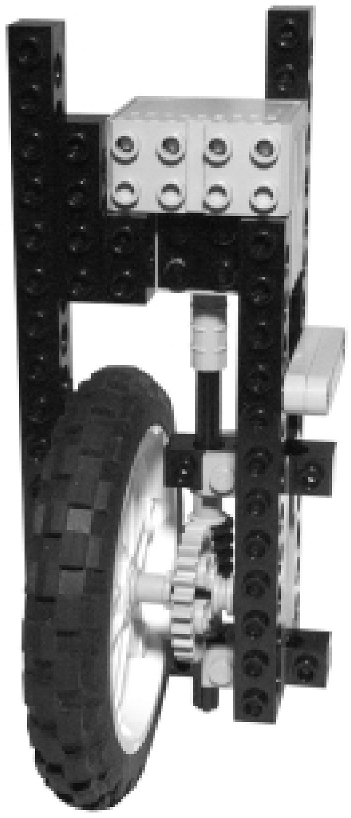
Building a SHRIMP

If you want to create your own SHRIMP, but don't have all the parts we used, the LEGO inventory offers many possible substitutes:

- **Gearboxes** A gearbox is a convenient way to match a worm gear to a 24t. But as you've seen in this book, there are many other assembly solutions, they're just a bit less compact.
- **Universal joints** Those that power the wheels are easily avoidable with a different construction. For example, our setup for the front wheel doesn't use them, and you can replicate this for all the wheels. In Figure 16.12, we show a wheel with no universal joints and no gearbox.
- **Polarity switches** You can use the free port of the RCX to control one bogie, and connect the other to the same port that drives the front and rear wheels. No polarity switches are needed for this configuration, but your SHRIMP would only turn in one direction.
- **Motors** A nonsteering SHRIMP also saves one motor. We didn't try, but we are sure it's possible to use a single motor instead of two in each bogie. In theory, with a lot of gearing, you can power your SHRIMP

with a single central motor: transport motion to the bogies through their supporting axles, and to the front and rear wheels with a system similar to the one we used for the steering setup. Such a SHRIMP will suffer from a lack of power and therefore climbing ability, due to the reduced number of motors and increased friction.

Figure 16.12 A Wheel Assembly



Creating a Skier

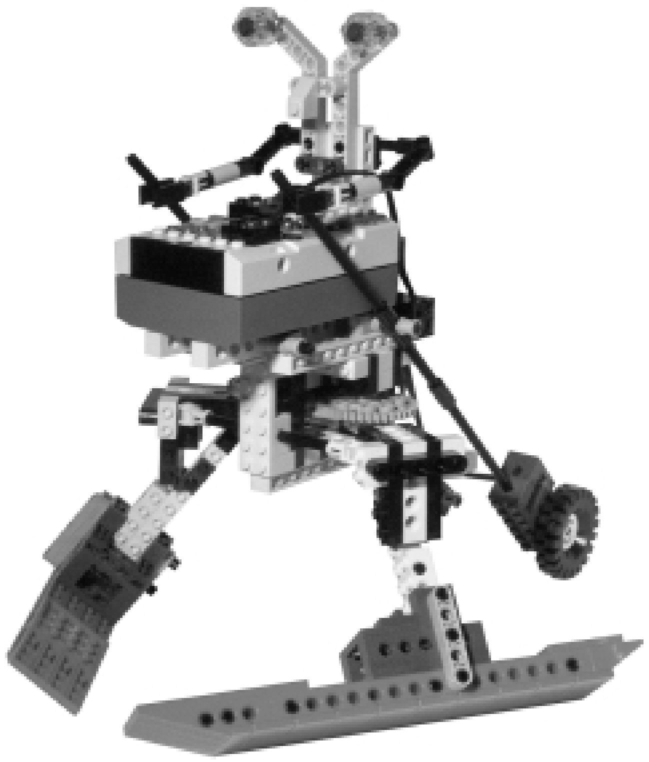
What we find most interesting in this project is not just the fact that this "skibot" robot can be used in the snow, but that without propulsion it descends snowy slopes like a true skier (well, almost!). It uses a technique known as *snowplowing*, due to the V-shape of the skis, often used by human skiers. In snowplowing of the human variety, the skier angles his toes inward in order to put the tips of his skis together and simultaneously dig the inside edges of the skis into the snow. To reduce speed, the skier pushes out the tails of his skis, increasing their angle to make a wider V; to increase speed, the skier draws the tails nearer, making the V narrower.

Our robotic skier is based on the same principle. It mounts onto a pair of skis, and while descending a slope, it varies the angle of the skis to increase or decrease the resistance and maintain a roughly constant speed. It uses only one motor and one sensor: The motor is on the back and operates the legs, making them more or less convergent, thus keeping the speed in the desired range; the sensor is a rotation, attached to a wheel at the end of the left ski pole, and serves to measure the speed.

Another interesting feature of this robot is that its geometry and the position of its center of gravity make it always point toward the direction of the maximum slope. There's no need to shift weight to control direction, this happens automatically because the motion along the longitudinal axis of the robot is the one that offers the lower resistance.

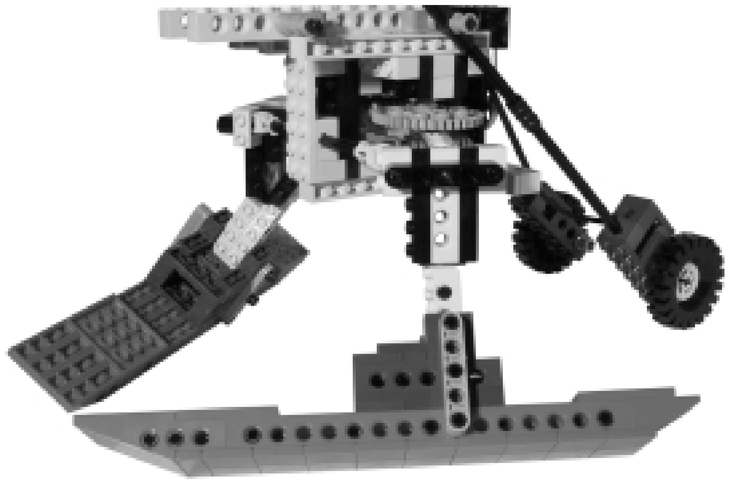
A general view of our skier can be seen in Figure 16.13.

Figure 16.13 The Skier



To build the skis you need some extra beams and, more important, many tiles (see Figure 16.14). We used 36 2 x 2 and two 1 x 4 tiles (available as spare parts at the LEGO Online Shop). If you are open to employing non-LEGO parts, you can build the skis from other materials, like strips of plastic available in hobby shops.

Figure 16.14 Side View of the Skier



The legs are not vertical, but rather are inclined outward. This is very important. For a human skier, it's what keeps the skis resting on their inner edges and producing the necessary resistance to gravity when in the convergent (snowplow) position (see Figure 16.15).

We achieved this effect by using some hinges and forming the legs from the diagonal of a perfect right triangle. If you don't have hinges, other possible solutions exist, like the one shown in Figure 16.16.

Each leg is rigidly attached to a 40t gear. The pictures don't show them, but we used the extra crossed holes of the gears to place pin-axle connectors into. The two gears meet a worm gear in the middle of the assembly, which receive motion from the motor and controls the convergence of the legs (Figure 16.17).

Looking at the bottom, you notice a longitudinal beam that locks the structures, and a transverse axle and beam that serve as boundaries to the movement of the legs (Figure 16.18). There are no limit switches. If the RCX tries to close or open the legs more than what's allowed, the belt will slip on the pulley.

Figure 16.15 Front View of the Skier

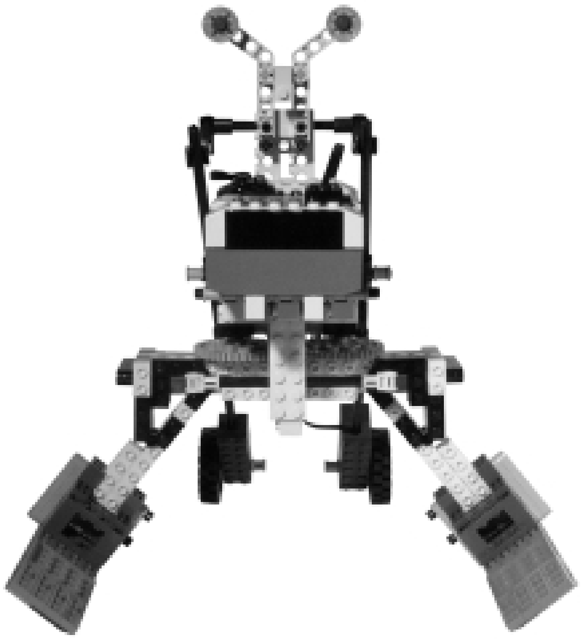


Figure 16.16 Alternative Leg Geometry

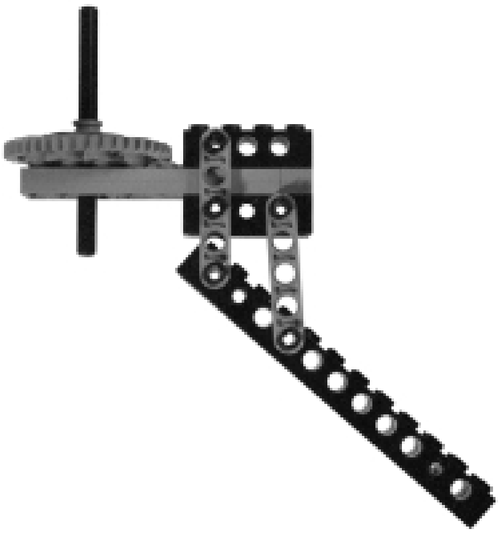


Figure 16.17 Rear View of the Skier

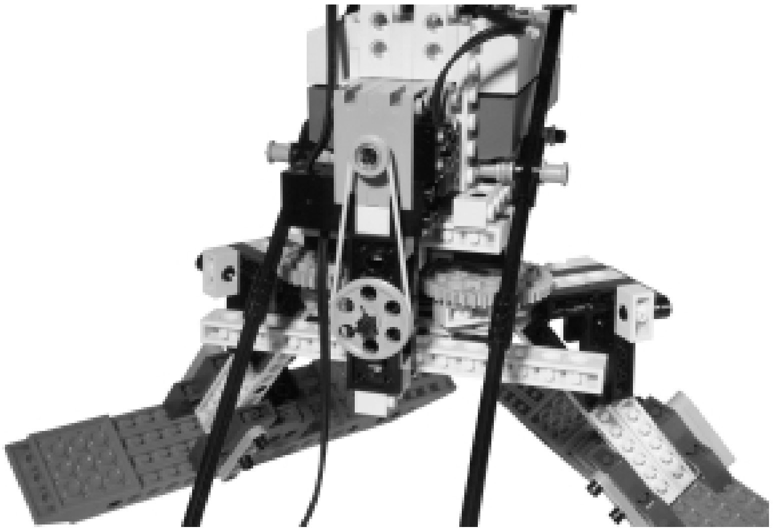
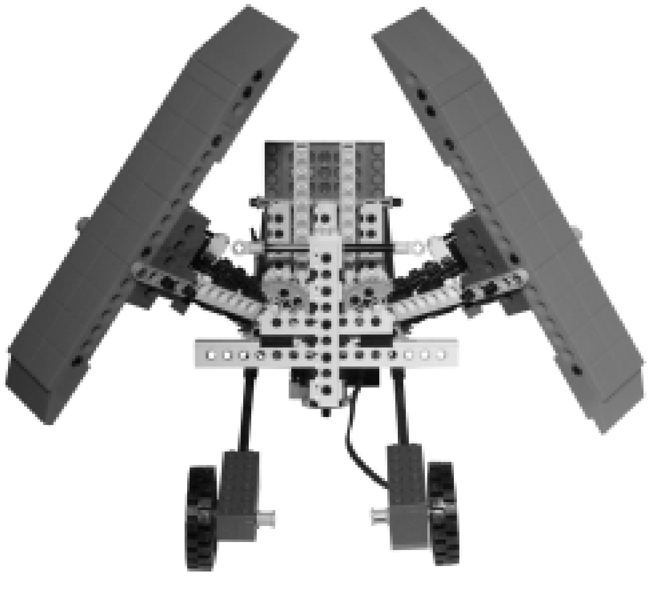
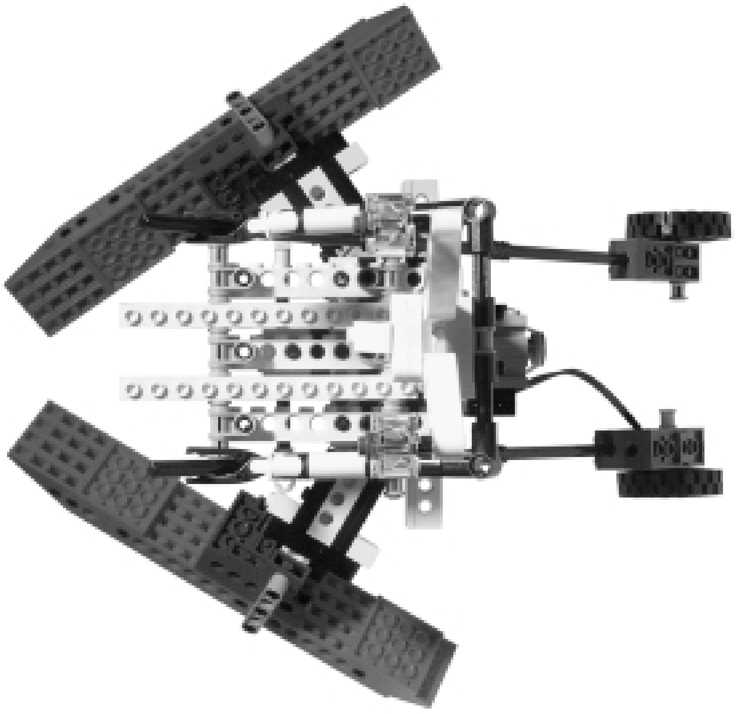


Figure 16.18 Bottom View of the Skier



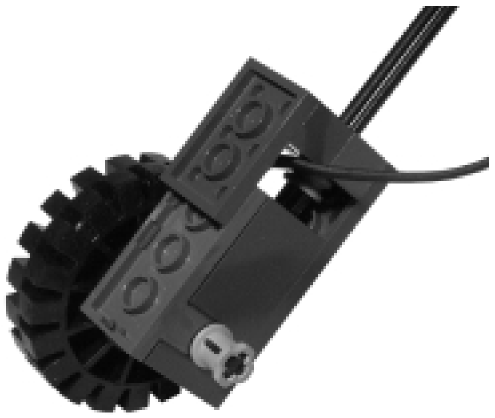
What's peculiar about this robot is that it has beams with all the possible orientations. The skis are studs down, the legs studs front, the body partly studs front, partly studs right, and partly studs up (see Figure 16.19).

Figure 16.19 Skier Top View (RCX Removed)



There's not much to say about the ski poles. The right one is just decorative, placed there for the sake of symmetry. The left one, meanwhile, incorporates a rotation sensor that's directly connected to a wheel (see Figure 16.20).

Figure 16.20 Detail of the Left Ski Pole



Programming this robot is so simple that the topic deserves only a few words. Inside the main program cycle, test the increment in the rotation sensor counts: If this falls in the range that represents the desired speed you chose, switch the motor off; if it's above the range, start the motor in the direction that closes the skis, and vice versa if it's below the range. This will speed up or slow down the skibot as needed.

If you test your skibot in the snow, try to find or create well-packed powder, like those normally found on ski runs. It's not able to ski on black diamond runs or in loose powder!

Inventing…

How Did We Test the Skibot without Snow?

Suppose you want to build this robot and test it, but currently there's no snow outside. This book was written during the hot Italian summer, so we faced that very question ourselves.

Well, we admit we had thought of going to the Alps to visit a glacier where some of the winter snow had survived. Unfortunately, we had to settle for a less expensive and time-consuming solution. We placed four large ice-packs, the kind used in portable camping refrigerators, on an inclined board. Then we covered them with frost taken from the freezer, gently pressing it down. It didn't make for a particularly long run, but it was enough to verify that our robot actually skied. A positive side effect of this experiment was that Mario's wife, coming back home, exclaimed: "You defrosted the freezer—bravo!"

What else could you improvise with to simulate a snowy run? Prepare an inclined plane (a table top would do the trick), and cover it with some fabric like a blanket, a sheet, a tablecloth or anything else you have handy. You have to adjust the slope depending on the kind of fabric you use, and the top will likely need to be steeper that the snowy slopes your robot can actually descend, because real snow produces much less friction.

Creating Other Vehicles

Here we present you with a list of suggestions for possible projects, their common denominator being that all of them are, at least in part, vehicles. They're meant just as starting points.

Elevator

We briefly discussed an elevator project in Chapter 4, in which we explained that a single touch sensor, placed in the elevator car, can control the positioning at an unlimited number of floors. We said also that a second touch sensor could serve the purpose of addressing the car to the proper level, using a simple system where the RCX counts the number of clicks on the sensor.

A variation on this theme is the car park elevator, where you emulate one of those automatic storing systems. It would be nice if your robotic parking could decode a sort of ticket, maybe using colors or shapes, so it can return the corresponding vehicle.

Train

The RCX is almost a natural extension to the LEGO 9v electric train system. They share the same voltage and the same connectors, and in fact many train fans currently use one or more RCXs to introduce automation in their layouts. This topic is so vast it would require a dedicated book, so we will provide only some basic tips.

There are two basic approaches to control a train with the RCX: a) the RCX is on the train; b) it supplies power to the tracks.

In case a), you put the RCX in the locomotive or in one car and connect an out port to the train motor. The train motor is also wired to the wheels, which normally draw current from the tracks, so it will happen that your RCX will supply power to the tracks, too. Nothing bad happens, but DON'T connect the train speed regulator to the tracks as well; you could damage your RCX.

If you are the kind of person who likes customizing things, you can open the train motor and interrupt the connection to the wheels, so your train will be totally independent from external sources. This way you can run many RCX-controlled trains on the same track. You can create some external references to read with sensors, so your train knows when to slow down or stop, or place a proximity sensor on the locomotive to avoid collisions.

In the second approach, b), you substitute the train speed regulator with the RCX and power the tracks from one out port. You can control three independent tracks or segments with a single RCX, and use the input ports, for example, to detect the arrival of the train at the station.

There are many other devices you can automate in your layout: switching points, level crossings, decouplers, semaphores, swing or draw bridges, and so on.

Cable Railway or Gondola

In a real cable railway, there are two pairs of cables: two supporting ones and two pulling ones. The supporting cables are more or less rigidly attached to the lower and upper stations, and work as railways for the cabs that have their pulleys running over them. The pulling cables transfer motion to the cabs: one cable goes from the first cab to the second across the upper station, while the second connects the two across the lower station.

You can place the motor either in the upper or the lower station. If you use the upper station, you can avoid the second pulling cable. Use touch sensors to control when the cab enters the station so you can stop the motor, possibly after a short slow down.

Boat

LEGO inventory includes different kinds of propellers, so one might wonder if it's possible to make a robotic boat. It is indeed possible, but it's not easy to provide the necessary flotation lift using only LEGO parts. The two solutions that come to our mind require uncommon parts, either single mould boats coming from the System product line, or a bunch of TECHNIC air tanks. The idea is to build a sort of catamaran, with both hulls made of two System boats or a row of air tanks.

A simple, cheap, and handy non-LEGO alternative for the hulls is to use common soft drink plastic bottles and attach them to long beams with rubber bands or duct tape.

The RCX will stay on the deck, together with the motor that drives the propeller and the other that controls direction. You can place bumpers on the front to make your robotic boat change direction when hitting an obstacle.

WARNING

The RCX, the motors and most other electronic components don't like water at all. While distilled water is a good insulator, common tap water, or water from the sea, lakes, or pools, conducts electricity extremely well and will damage your devices. Take every precaution not to soak or submerge them.

To minimize the risk of damages in case of an accidental bath, put the RCX into a small transparent plastic bag, with just the wires coming out from the opening, and seal the bag with a rubber band. Run your robot in a controlled environment with calm waters, like a pool with no people in it.

Sailing Tricycle

We are both fans of sailing, and in the wake of the great success of Luna Rossa, the Italian sailboat that won the Luis Vitton Cup 2000, we decided to build a robotic sailing tricycle or land yacht. We named it Duna Rossa (Red Dune) to mimic the original Luna Rossa (Red Moon).

Though building that tricycle has been a lot fun, we must admit that the performances were less than exciting. With a strong wind and a favorable slope... it moved!

See if you're able to do better: keep the structure as lightweight as possible, use a very large sail, and reinforce the mast with shrouds, forestays, and backstays (ropes).

The RCX controls two motors, one to steer the rudder and the other to operate the winch for the mainsail. You can detect the wind direction through a vane on the masthead connected to a rotation sensor. Monitor the position of the boom with a second rotation sensor to adjust it for the proper angle with the electric winch. Finally, you'll need a third sensor (a touch is enough) to control the position of the rudder.

You can program your robotic sailing tricycle for two basic behaviors: adjust the mainsail to keep with the desired course, or adjust the course to maintain a specific sailing point.

Summary

If there's a lesson you can draw from this chapter, it is that any problem in robotics requires a custom-tailored solution. After having long talked of standard mobility configurations, we described some situations where none of them applies.

You're not necessarily required to invent new solutions any time, more often you can just look around you, or on the net, and find something that helps you or points you in the right direction. SHRIMP was designed to move on bumpy terrains, but there are obviously other possible configurations. For example, the robotic vehicles designed for planetary exploration are a good source of inspiration, and there's a large quantity of documentation available in the public domain.

We must confess that our skier was born more for purposes of fun than to demonstrate some general principle. Nevertheless, this small and simple robot has its merits, helping us picture the wide range of applications robotics can be used for.

Chapter 17

Robotic Animals

Solutions in this chapter:

- **Creating a Mouse**
- **Creating a Turtle**
- **Creating Other Animals**

Introduction

Trying to emulate animals in designing and constructing a LEGO robot is a fun and instructive experience. You can approach the problem from different viewpoints—for example, concentrating your attention on the behavior of the animal, or on its shape; you can even create a pure fantasy animal or develop your interpretation of some mythological creature.

In the following pages, we will show you two projects: a mouse and a turtle. The mouse is probably the simplest project of the book, using just one motor and two touch sensors, and it's easy to replicate using only the parts contained in the Robotic Invention System. The turtle, on the contrary, is a bit more sophisticated and requires a few additional parts.

Once again, the robots of this chapter offer us the opportunity to revise and apply some of the concepts stated in the first part of the book. You will discover a new use for caster wheels—through which we will implement a sort of automatic steering feature for our robotic mouse—and a leg geometry for the turtle with a design not shown in Chapter 15. Both the robots employ touch sensors to detect collisions, but we'll also suggest other techniques that result in more intelligent behavior from your robotic animals.

The last section of the chapter contains a list of proposals intended as starting points for new projects inspired by the world of animals: squirrel, mole, ostrich, kangaroo, crab—take your pick!

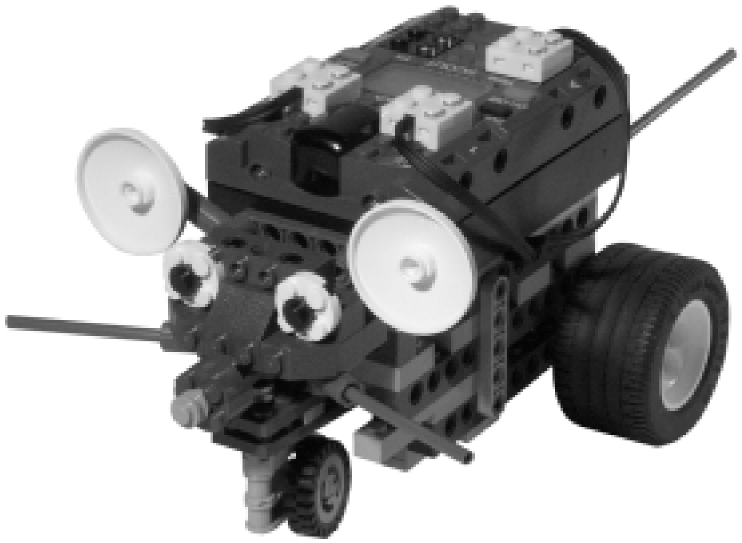
Creating a Mouse

Our LEGO mouse doesn't claim to look very similar to a real one, instead it resembles one of those simple mechanical mice seen in cartoons, which drive cats crazy. The programmed behavior is very simple, too: the mouse goes straight until it hits an obstacle, then goes back for a while, changes direction and goes forward again. If someone pulls its tail, it stops for a few seconds, then restarts. Very young children love to run after it trying to catch its tail, and in case you have a dog or a cat at home they very likely will love it, too—maybe even too much!

In this project, we used the Scout, the younger brother of the RCX found in the Robotics Discovery Set, but you can obviously build yours around the RCX (Figure 17.1).

We thought that size was a very important factor for a robotic mouse: We wanted ours to look like an adorable mouse and not a large rat! For this reason, in designing this robot we put a lot of attention into restraining its dimension. This goal affected our choices regarding the driving system, which employs a technique not yet described in the book: a variation of a steering drive that requires only one motor. In fact, the motor is used to power the drive wheels, while the steering capability comes from the unconventional use of a caster wheel.

Figure 17.1 The LEGO Mouse



Looking at the bottom, you can see the very simple mechanics: a motor drives the differential gear with a 1:1 ratio (Figure 17.2). Combined with the large diameter of the wheels, this makes our mouse very fast. Exactly what we wanted.

The front caster wheel is the key to the turning ability of the mouse. All pivoting wheels tend to follow the direction of the body they are attached to, but this has its range limited by mechanical constraints. In fact, the liftarm attached to the pivoting axles cannot go further than the two left and right plates that block it (Figure 17.3). When the robot goes backward, the caster turns until the liftarm hits the blocking plate, and with the caster in that position, the robot turns. Resuming forward motion, the robot will go straight again. To make the caster even more reactive, we built it asymmetrical.

When describing the steering-drive configuration in Chapter 8, we explained how the distance of a steering wheel from its pivoting axle affects positively its tendency to self-center: the greater the distance, the more effective the self-centering. This applies to caster wheels, too, and it is the reason why we used that particular connector to attach the front wheel: It's the one that keeps the wheel farthest from its pivoting axle.

Figure 17.2 Bottom View of the Mouse

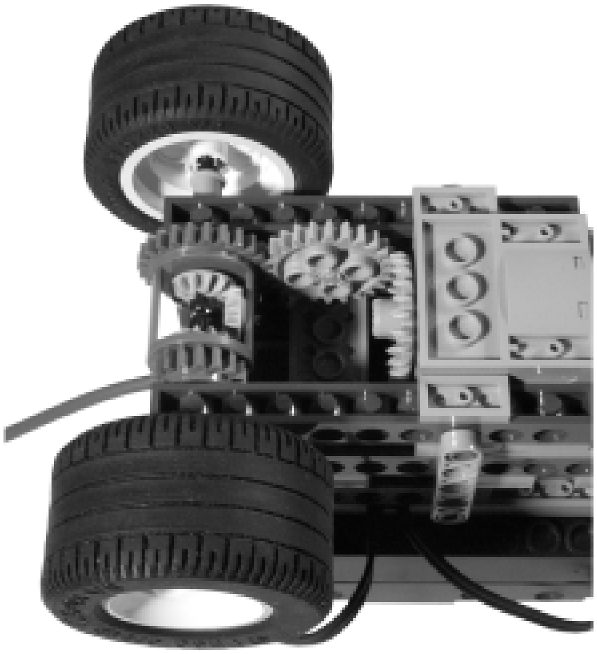
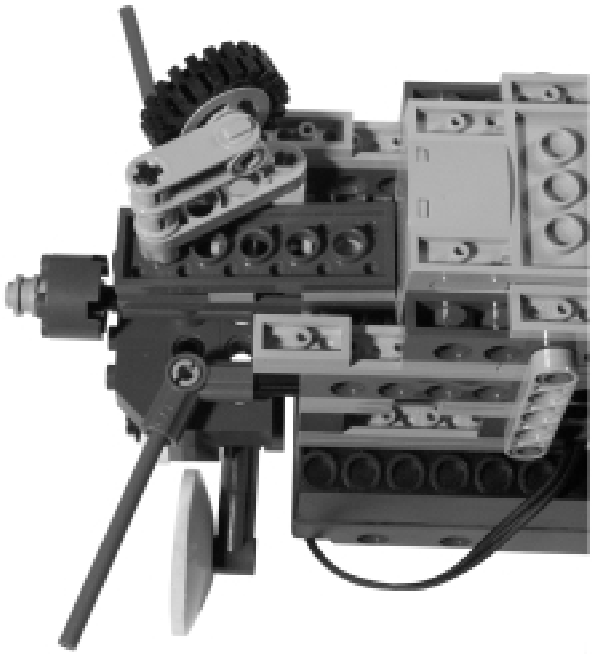
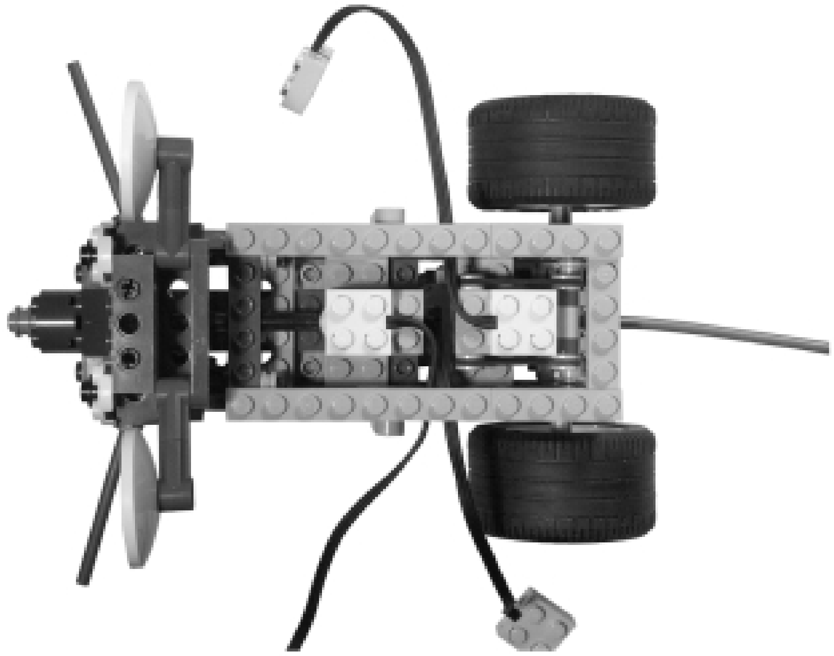


Figure 17.3 The Caster Wheel Turns the Mouse



Removing the RCX, you see the two touch sensors that control the head and tail (Figure 17.4). The rear one is normally kept closed by two rubber bands attached to the tail assembly. Pulling the tail opens the switch and the software stops the mouse.

Figure 17.4 Top View of the Mouse (Scout Removed)



The entire head is a bumper that detects hits through its nose, whiskers, or any other part (Figure 17.5). The short tubes we used as whiskers are the only parts you won't find in the MINDSTORMS kit. We are not suggesting you cut the longer one in two, however! You can use axles in place of tubes.

The head is what actually makes this robot a mouse instead of any other possible creature (Figure 17.6). This demonstrates how a few parts can deeply affect the "personality" of a robot! We leave to you the exercise of converting this mouse into a different animal by changing some of its decorative elements.

Figure 17.5 The Front Bumper of the Mouse (Head Removed)

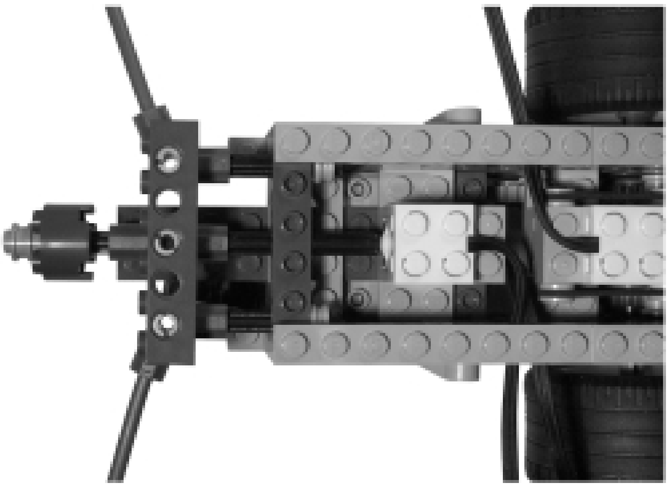
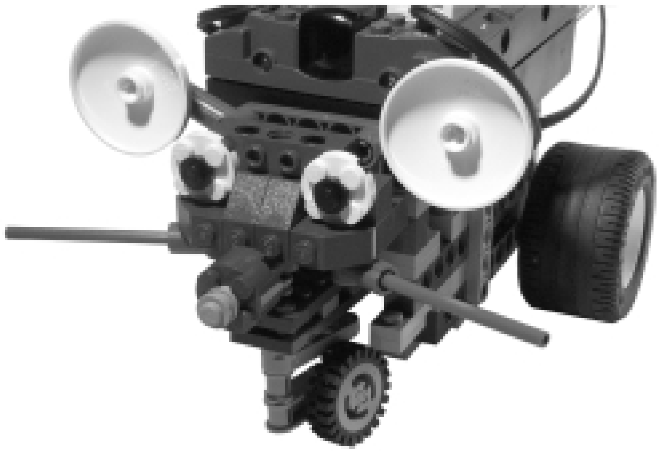


Figure 17.6 The Head of the Mouse

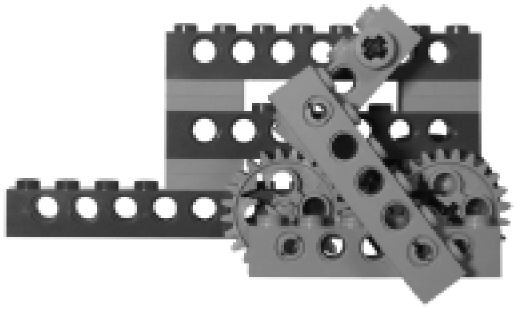


Improvements Upon the Mouse's Construction

We explained that one of our goals was limiting the size of the robot, and we achieved it through a single motor steering drive. The limitation of this setup is that it doesn't allow you to control the direction of the robot, which results in a mouse whose movements are very predictable. Using a second motor to control the front steering wheel you can enrich the behavior of the mouse with some random changes in direction. You should redesign and enlarge the body of the robot to make room for the second motor.

You can also add legs to the mouse to make it more realistic. Some of the legs shown in Chapter 15 will suit this model, but with working legs it will be definitely much slower. We wanted a fast mouse, and this is the reason we chose wheels. An intermediate solution comes from attaching legs to the wheels so that they move and give the illusion they are propelling the robot (Figure 17.7).

Figure 17.7 A Pseudo-Leg



When the programmable brick faces the direction of motion, you can use the IR message emitter of the RCX and a light sensor in the nose to implement the proximity detection technique described in Chapter 4, thus saving your mouse from too many collisions.

Creating a Turtle

The LEGO turtle project is a bit more ambitious than the mouse project, as we really tried to mimic the size, shape, and way of walking (as well as other basic behaviors) of a real turtle. Our robotic version walks around, and when it hits something, retracts its head and waits a few seconds. Then it goes backward a while, trying to avoid the obstacle, eventually resuming straight motion.

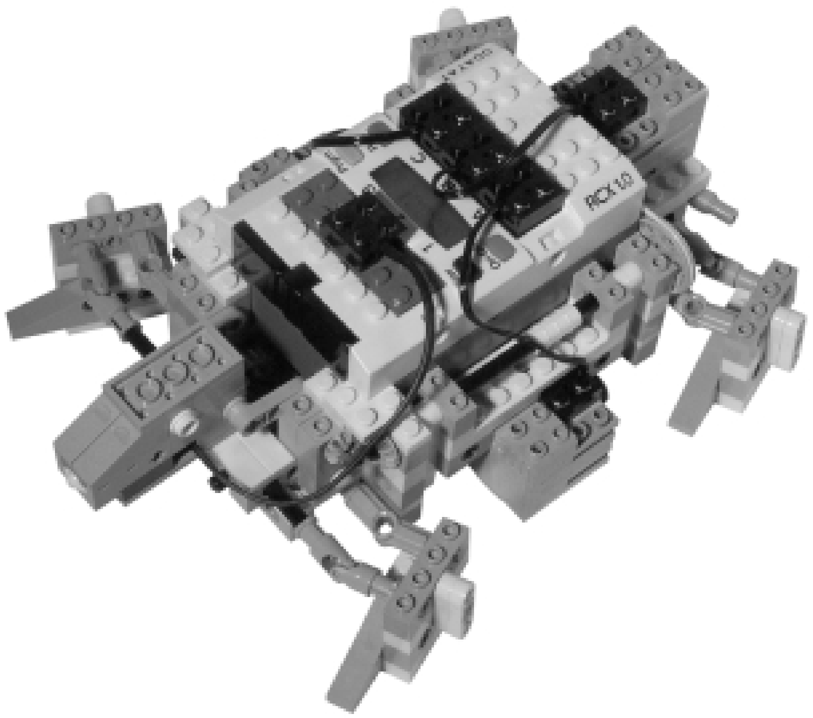
We used a few extra parts, the most important ones being four universal joints and four #3 angle connectors. You can save the third motor and the tiles if you eliminate the retractable head.

Figure 17.8 shows a general view of the turtle. We made a strong effort to fit all the components into a design not too different from the shape of an actual turtle.

The most relevant parts are the legs. In contrast to what we made for the mouse, this turtle walks on feet and uses a leg geometry not shown in Chapter 15. In fact, all the legs shown in Chapter 15 tend to produce tall robots. In this case, however, we needed a compact design suitable for a short creature. You can understand the principle behind our mechanism by way of a simple experiment.

Connect two axles with a bend connector, rotate one of them with your fingers and observe the tip of the other: it rotates in a circle (see Figure 17.9 a and b). Keeping the first axle horizontal, you'll notice the end of the second behaves similarly to the leg assemblies in Chapter 15. We implemented this solution using the angle connector #3, one side of which is attached to the foot and the other to the leg gearing, as you can see in Figure 17.9.

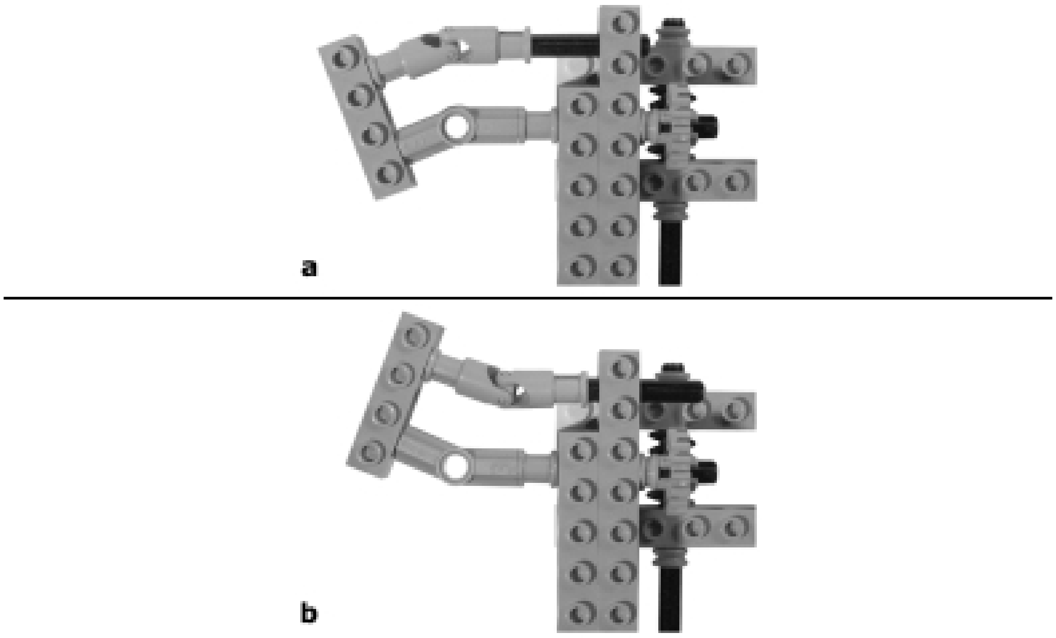
Figure 17.8 The Turtle



We couldn't rigidly attach the leg to the end of the axle, because an animal whose feet turn upside-down would have been ridiculous! Thus, we needed a second connection point for the leg to keep the foot parallel to the ground while the angle connector rotates. The universal joint was the perfect candidate: It allows the leg to cycle ahead, down, back, and up, remaining parallel to the floor all the while. The axle that supports the universal joint is free to slip back and forth inside the hole of the beam to comply with the movements of the leg.

To match the 16t with the worm gear, the distance between the intersecting beams is one plate instead of two as in the standard grid.

Figure 17.9 Detail of the Turtle's Leg Assembly



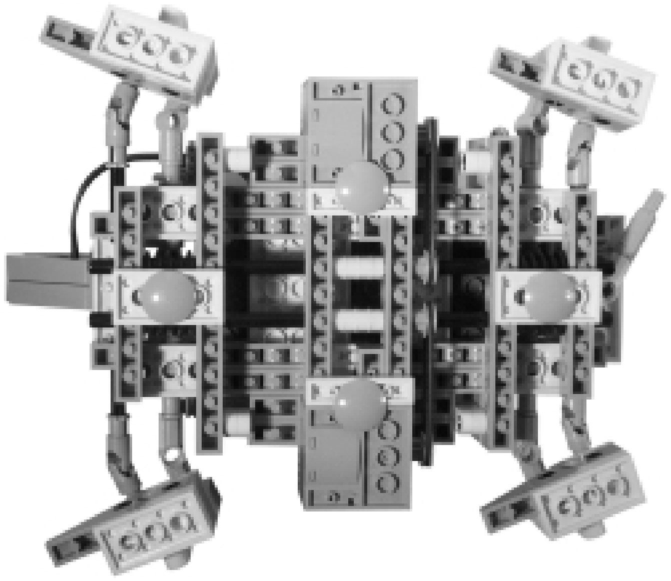
Bricks & Chips…

Using Angle Connectors

There are currently six types of angle connectors in the LEGO line, numbered 1 to 6. In case you're wondering how the numbers relate to angles, here are the correspondences: 1 = 0°, 2 = 180°, 3 = 157.5°, 4 = 135°, 5 = 112.5°, 6 = 90°. They go by increments of 22.5°, a quarter of a right angle.

This turtle is not a real walker, meaning that it doesn't only employ its legs to support its weight; it helps itself along with the rest of the body. Its walk resembles that of very young babies; when they begin to crawl they still need their belly as an additional support, because they are not yet able to co-ordinate arms and legs. Recall the four-legged walker of Chapter 15: It has the legs diagonally paired, the front left with the rear right, and the front right with the rear left. In this turtle, such synchronization doesn't exist; instead, the legs are paired side by side, and go up and down together. The feet of each side are in contact with the ground only during the lower part of their circular motion; during the phase when the feet are raised, the turtle rests on additional supporting points under its body. Figure 17.10 shows clearly the four tiles placed at the bottom for this purpose, as well as the complete transmission system.

Figure 17.10 Bottom View of the Turtle



The fact that there is no coordination between the left and right legs makes the gait of the turtle very irregular. Sometimes the two sides go up and down together and the turtle advances with short but regular steps. Other times the two sides are out of phase and the turtle proceeds by swinging left and right. We really cannot say that this robotic turtle is particularly efficient in its progress, but have turtles ever been the image of efficiency?

The head incorporates a touch sensor, used as a very simple bumper (Figure 17.11). When the software detects a hit, it briefly operates the third motor, placed at the rear side of the robot, to retract the head.

The head motor transmits motion through a basic pulley and belt set that makes the system tolerant about timing. A long (joined) axle runs along the left side of the turtle and through a pair of bevel gears, which transmits motion to the liftarms that retract the head (Figure 17.12).

Figure 17.11 Close-Up of the Head of the Turtle

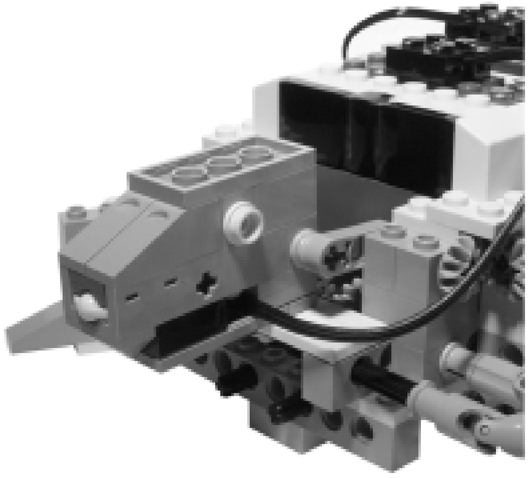
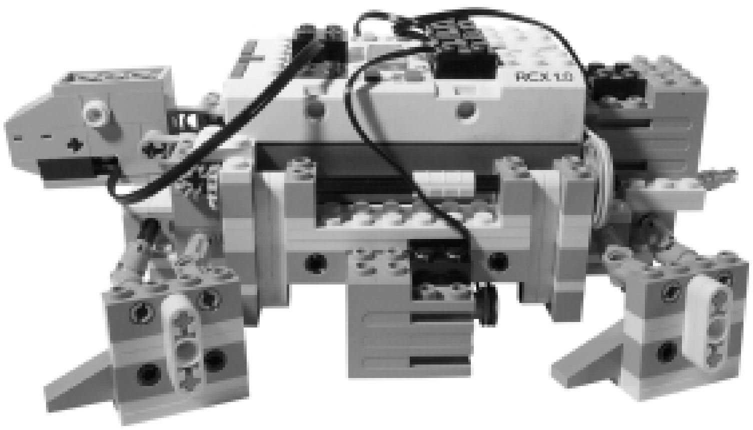
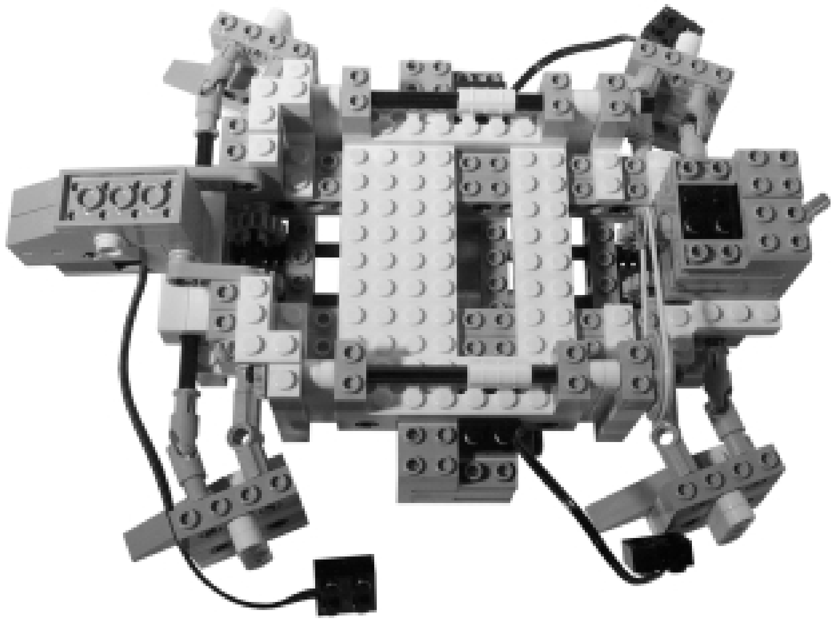


Figure 17.12 Left Side View of the Turtle



The body structure is rather simple and solid, the only notable detail being that it doesn't use the standard grid. As we explained before, the lower layer is separated by only one plate from the mid layer in order to make the 16t gears match the worm gears. To be able to brace the chassis with vertical beams we also had to adjust the distance between the mid and upper layers, so we used three plates (one brick) instead of two (Figure 17.13).

Figure 17.13 Top View of the Turtle (RCX Removed)



Improvements Upon the Turtle's Construction

If you have the necessary parts, you can build a carapace (shell) to cover the RCX, the motors, and other mechanical details thus making your robotic turtle a more refined model.

On the mechanical side, you could try and design retractable legs as well as a retractable tail, but we're sure you'd have to increase the size of the model to fit the required mechanisms.

There are many things you can do to improve the behavior. For example, it doesn't necessarily have to always go straight until it hits an obstacle, but could instead stop or change its direction at random intervals thus showing a more naturally random behavior. Turtles are very cautious creatures, and you can add some additional sensors to make yours more sensitive to changes in the environment: a light sensor could detect any sudden variation in light, or a custom sound sensor could perceive noises, all of which might put the robot on alert.

The nose-bumper is quite a primitive system with which to detect obstacles. It works well when the turtle goes straight into a wall, but it is not very effective in collisions against other types of objects. As with the robotic mouse, proximity detection is a nice addition to this turtle, too, either in the form of a custom proximity sensor that replaces the touch sensor in the nose, or in using the IR message-light sensor scheme described in Chapter 4.

Creating Other Animals

Nature is a wonderful source of inspiration, and you can collect tons of great ideas just by browsing through books about animals. Insects and spiders can be used as templates for multilegged walkers, while other creatures higher on the evolutionary scale present an incredible range of shapes and behaviors. Take your pick.

Matching the shape to the function is almost an art. Even our simple turtle required many trials before we felt satisfied with its design. Top of the list in this field is Jin Sato with his MIBO robotic dog: a nicely shaped puppy with a very sophisticated dual RCX robot that's quite surprising to see in action. But don't be intimidated in the slightest by such an advanced design. You can build many interesting robotic animals without having to reach that level of complexity. The following list provides some examples of what you could make. Keep in mind, there are many other creatures at least as interesting and challenging as these just waiting for you:

- **A squirrel** This robot would collect 2 x 2 bricks as if they were nuts, and perhaps bring them back to its hole (difficult).
- **An ostrich** It won't bury its head in the sand, but it can hide its head between its legs.
- **A kangaroo** We like the challenge of designing a jumping robot, but so far we haven't succeeded. The idea was to implement a sort of spring or rubber band mechanism that, when slowly loaded by a motor, could release its energy all at once, thus making the kangaroo jump. We conducted a few experiments, and although our prototype actually skipped and advanced a bit, we didn't consider it fully successful. It's still an open challenge!
- **An armadillo** This robot would roll up like a ball when disturbed.
- **An oyster** Even a simple animal like this offers some ideas. For example you can build one that closes its shell very quickly when someone steals its pearl, and the game could be to try and take it out without being touched by the shell.
- **A dinosaur** This category includes such a broad variety of creatures that you shouldn't have any problem finding one to emulate.

- **A porcupine or hedgehog** This robot could raise its quills (axles) when detecting some stimulus.
- **A crab or lobster** This type of robot would clamp down on everything that touches its claws.
- **A koala** This robot would climb on a tree. (You'd probably need a specially shaped LEGO tree for this one!)
- **A mole** A dark seeking robot that looks for the darkest places in your room, presumably under some piece of furniture, and rests there until some light disturbs it.

With additional RCXs you could model two or more animals, either of the same kind to cooperate in certain tasks, like ants, or of different types, so that one hunts the other, which tries to escape.

Summary

In this chapter, we discussed designing a robot after existing creatures, animals in particular. In addition to technical issues, you have to face the difficulties that come from the necessity of matching the shape to the function. In fact, in the previous chapters we concentrated our attention on solving technical problems without introducing concerns about the size or appearance of our robots. However, when trying to emulate animals, you cannot do so without a careful study of the shape of the robot; actually, it is the most important factor—it's what makes your robot look like an animal instead of a vehicle. Decisions about the appearance of a robotic animal usually come before any other mechanical choice, and they will push you to look for technical solutions that suit the desired structure. Sometimes you will find them and will be able to carry out your original design; other times you will have to introduce some adaptations into the structure to make a mechanical solution possible.

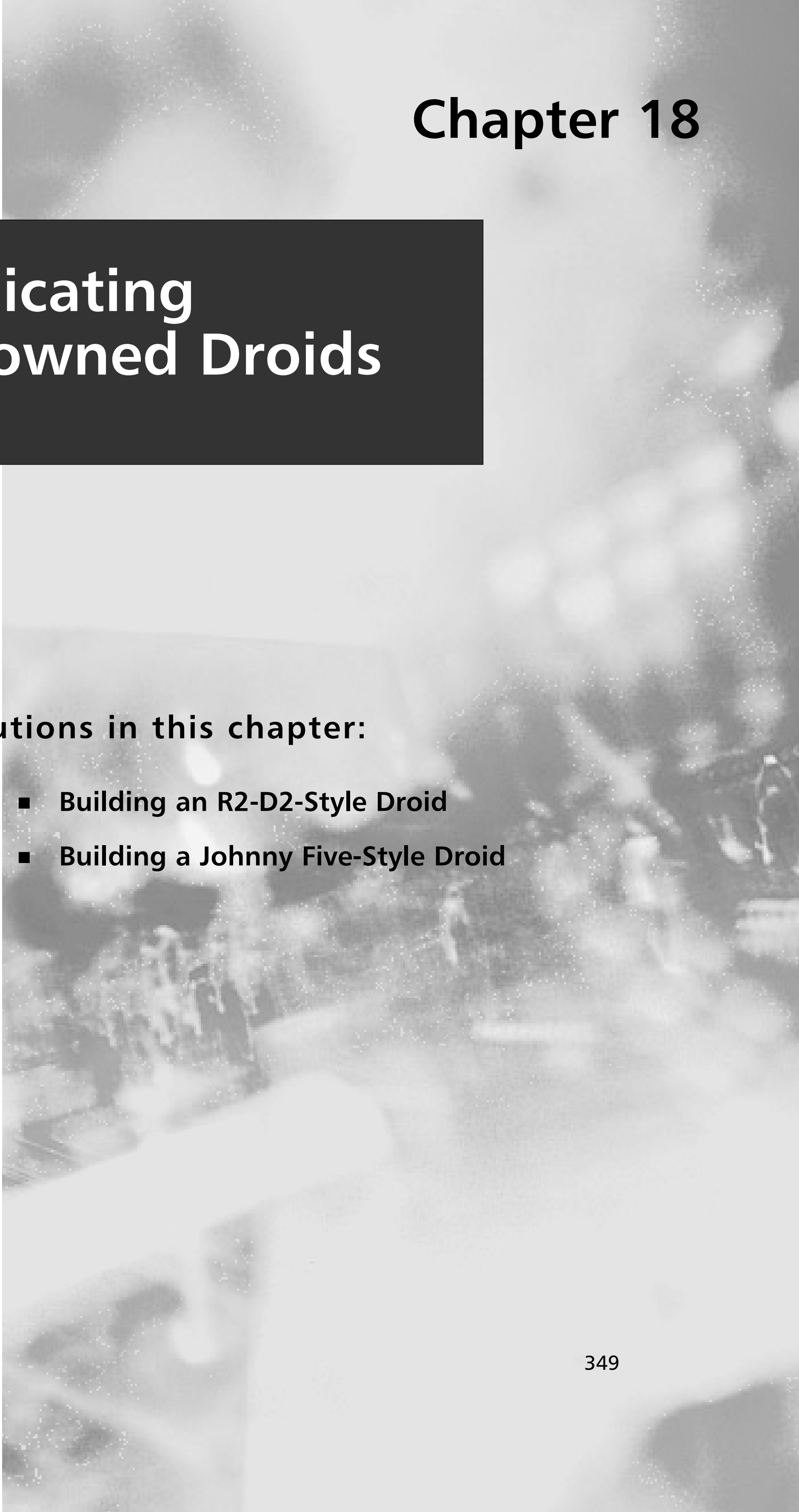
The mouse and turtle described in this chapter are good examples of this approach. In describing the mouse, we explained that one of our goals was to limit its size. This led us to use a variant of the steering drive configuration which, at the price of reduced control regarding direction, works with a single motor. Another goal was to make the robot very fast, which resulted in the use of wheels instead of true legs.

The turtle project started with a different premise: the robot could be slow, or had to be slow, thus a legged configuration was quite appropriate. But the turtle had to be short and flat too, and none of the leg designs described in Chapter 15 were suitable. So we had to figure out a new design that, though not particularly efficient, was appropriate for this robot.

Having stated the importance of shape when emulating animals, we don't want you thinking it's the only thing that counts. Knowledge from the previous chapters should still be taken into account, making you pay attention to the solidity of the structure, to the gear ratio of the mechanisms, to the effectiveness of the bumpers, or to the position of the COG.

As for programming, when the behavior you want to emulate is very simple, like in the case of our mouse and turtle, it is somewhat secondary to the design, and usually does not require a great deal of effort. However, if you aim is to reproduce more complicated behavioral schemes, programming will become an important factor in the success of your project.

The last lesson to remember from this chapter is that by studying and observing animals, you can learn many tricks useful to robotics. Animals provide endless inspiration when it comes to challenging robotic projects!



Chapter 18

Replicating Renowned Droids

Solutions in this chapter:

- **Building an R2-D2-Style Droid**
- **Building a Johnny Five-Style Droid**

Introduction

If you're a fan of science fiction novels and movies like we are, it's quite natural to try and reproduce some of their leading robotic characters. Obviously, you cannot hope to get even close to the complex behavior they show in films. That would be an impossible task even with resources well beyond those of the MINDSTORMS system, but even with a much more modest goal in mind, you will discover this is not an easy task. The difficulties come from trying to model a small scale robot after a large-sized one with a complex shape; something not easy to reproduce using LEGO parts.

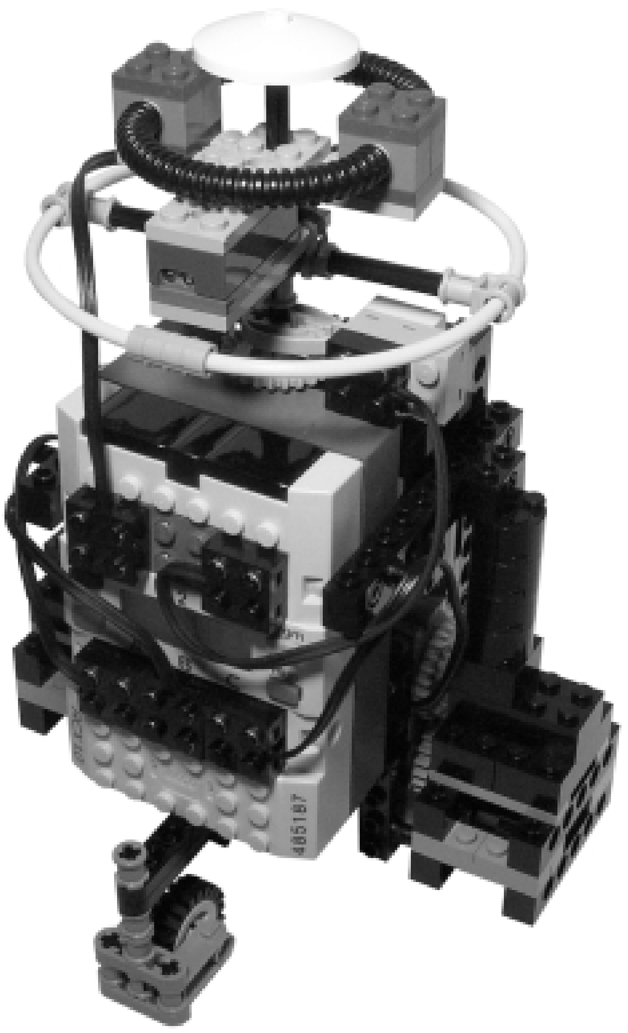
In this chapter, we describe the clones of two very famous robots: R2-D2 from George Lucas' *Star Wars* saga, probably the most beloved android of all time, and Johnny Five/Number Five from John Badham's film *Short Circuit*. We challenged ourselves to build both of them using only MINDSTORMS parts, plus an optional third motor, with both of them designed for light following, a matter not yet explored in the book. As always, you're invited to use our models as starting points for your own variations either in shape or functionality.

Building an R2-D2-Style Droid

The "real" R2-D2 is essentially made of a cylindrical body culminating in a hemispherical head. Two rigid legs come out from its sides, ending with the wheels that provide motion to the robot. R2-D2 is a differential drive. The original character also features a front retractable wheel used only under certain conditions; in our model this is the third supporting point necessary for balance.

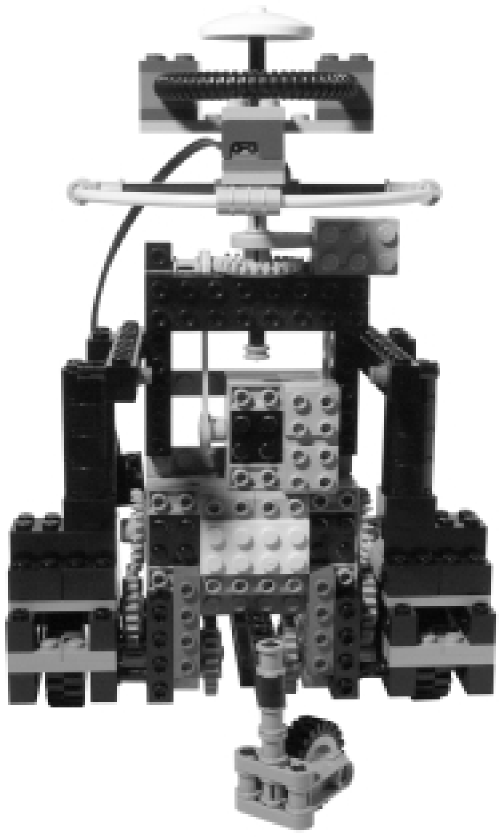
Figure 18.1 shows our R2-D2. You'll notice it's more of a symbolic representation than a realistic model! The RCX, mounted vertically, constitutes the main part of the body, while the head is mimicked by a compound structure of tubes.

Figure 18.1 Our R2-D2-Style Droid



The three motors are behind the RCX (Figures 18.2 and 18.3). Two of them, at the bottom, connect to the wheels with a 1:3 ratio, while the third rotates the head. As we explained at the beginning of the chapter, if you don't have the third motor, you can build a fixed-head version of the robot. It, too, will be able to follow light.

Figure 18.2 Front View (with RCX Removed)



Each of the drive motors mounts a 24t crown gear, which engages another (plain) 24t whose axle ends with an 8t gear. The latter engages a third 24t gear connected to the wheel (Figure 18.4).

The legs are built mainly with plain bricks and 2 x 2 round bricks. They end in a 1 x 2 TECHNIC brick attached to the horizontal beam that locks the upper part of the chassis and carries the RCX (see Figure 18.5). The front wheel is a simple caster, which is very important for the proper balance of the model. When a robot has a vertical shape, like R2-D2 has, the position of its mass is critical for its stability during changes of direction and speed. It's not just a matter of keeping the COG inside the supporting base, but mainly of opposing the effect of inertia, which could make your robot flip over (see Chapter 5).

Figure 18.3 Rear View (with RCX Removed)

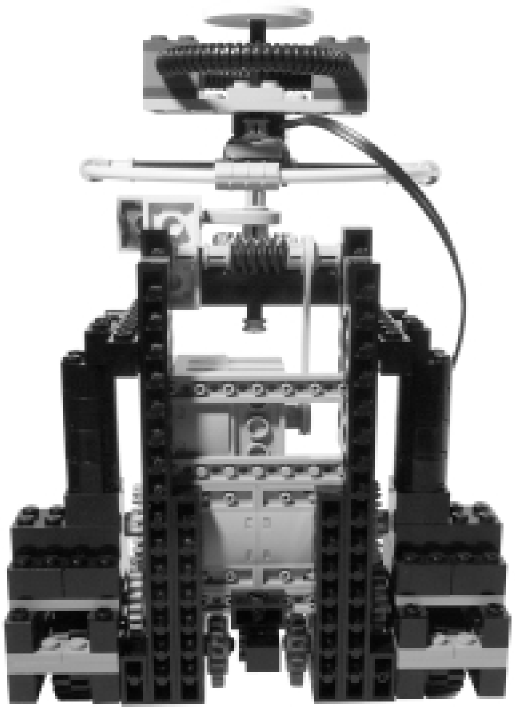


Figure 18.4 Detail of the Gearing

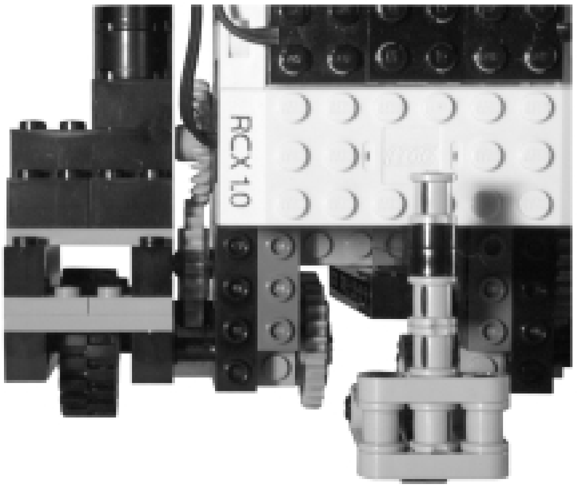
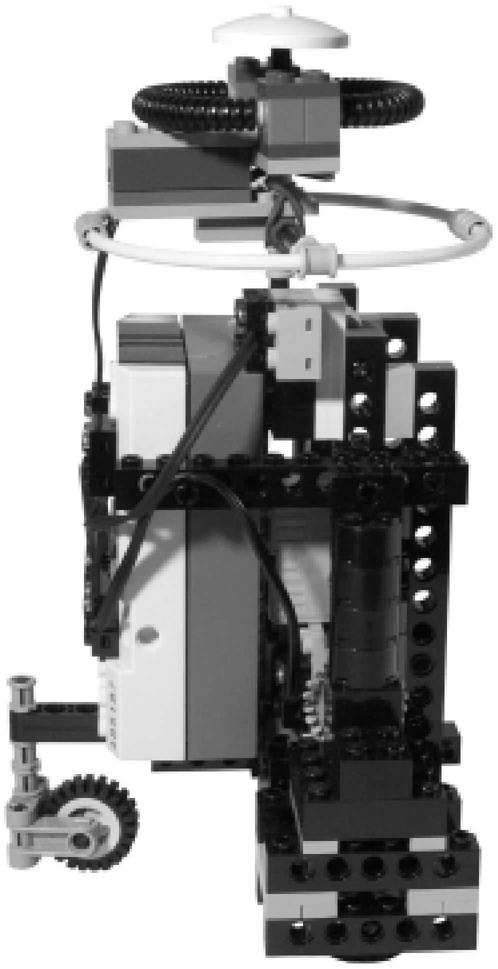


Figure 18.5 Side View



Looking at the bottom of the robot, you can see the beam that supports the pivoting wheel (Figure 18.6).

The head mounts a light sensor (Figure 18.7) that you'll use to locate the light source, and gets rotated through a pulley-belt worm-24t gearing system (Figure 18.8). This is one of those cases where pulleys and belts help in transmitting motion to a distant subsystem. A touch sensor detects the central position of the head through the tip of a cam, with a system similar to the one described in Chapter 14 about the steering assembly of a car.

Figure 18.6 Bottom View

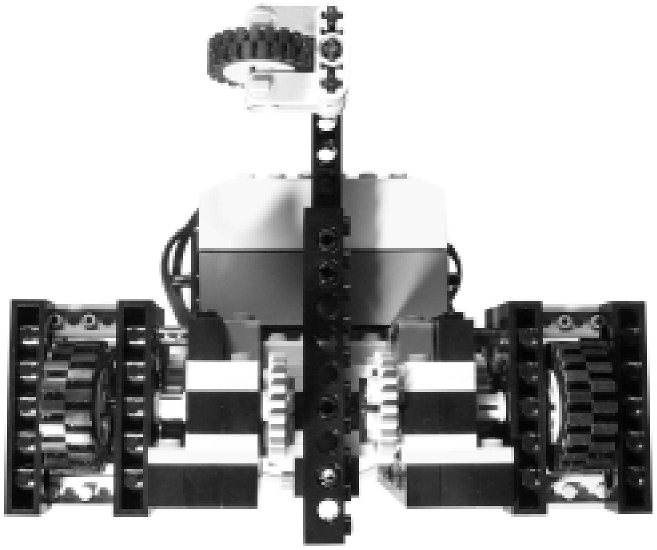


Figure 18.7 Detail of the Head

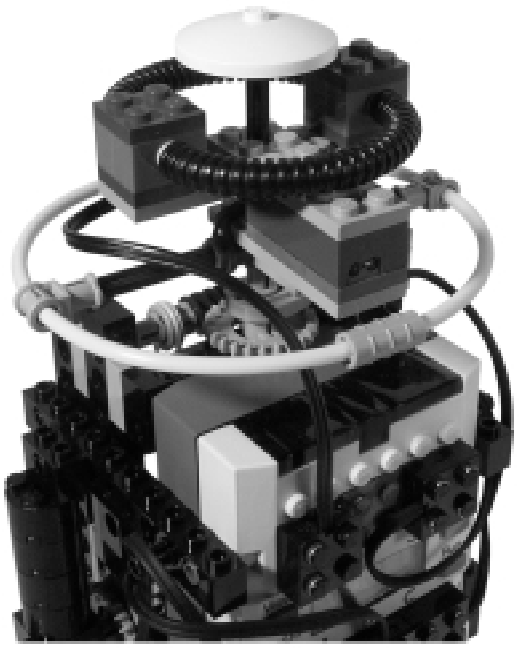
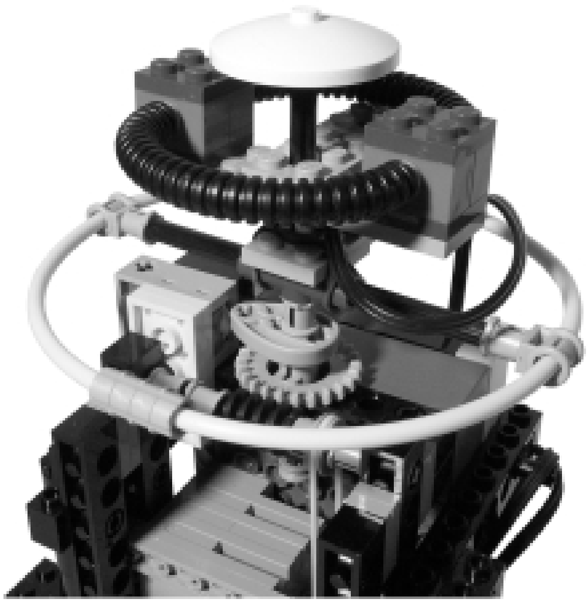


Figure 18.8 The Head Mechanism



Programming the Droid

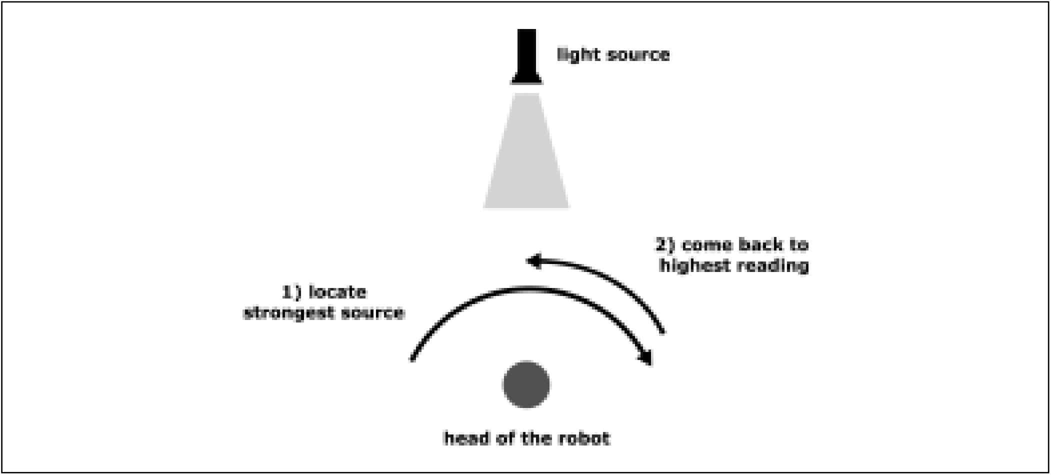
This model of R2-D2 has been conceived as a light follower. The idea is that you drive it using a source of light, like a flashlight. Though similar to line following in some aspects, this task requires a different strategy. The main difference is that unlike line following, in which your robot gets consistent readings (that is, the light sensor always reads the same values for black or white), when the robot follows a flashlight beam, the intensity of the driving light will change as the distance from the source to the robot's sensor varies.

Thus, you have to differentiate the driving light source from all the other sources of direct or indirect light in the room. To achieve this, you must scan the environment, rotating the head of the robot (or the robot itself) to find the strongest value. Then you will rotate the robot until it finds that intensity again (Figure 18.9).

In our first variation in which the robot has a rotating head, the robot is stationary; it rotates the head some degrees left, then centers it again at the same time reading the light sensors and storing the maximum value into a variable. The following NQC code fragment shows an implementation of the algorithm. It assumes that the light sensor is attached to input port 1 and the touch sensor to input port 3, and that the motor that operates the head is connected to output port B. You have to adjust the value of the *ROTATION_TIME* constant to reflect the time your robot takes to rotate its head left at the desired angle:

```
#define ROTATION_TIME 50
maxlight_left=0;
OnFwd(OUT_B);
Wait(ROTATION_TIME);
OnRev(OUT_B);
while (SENSOR_3==0)
{
  light=SENSOR_1;
  if (light>maxlight_left)
  {
    maxlight_left=light;
  }
}
Off(OUT_B);
```

Figure 18.9 The Flashlight Finding Procedure



Then you repeat exactly the same procedure for the right side, storing the maximum reading into the *maxlight_right* variable. Now you know which side is the strongest light source, left if *maxlight_left* > *maxlight_right* or vice versa, and you make the robot turn in place, with its head still and centered, until it finds a similar value. It's important that your test has some tolerance, as the light intensity might have changed a bit from the first moment you read it to the present time when you try and aim the robot at it.

At this point, your robot goes straight for a while, then stops to look for the maximum intensity again and correct its route.

The variation in which the robot has a fixed head is not too different, just change the search procedure to turn the robot in place instead of its head. You first turn the robot left for a few seconds, then you turn it right for a few seconds more while monitoring the readings as in the previous case. The NQC code that follows is very similar to the previous example, but uses the drive motors—connected to output ports A and C—to make the robot turn. In this case the ROTATION_TIME constant should correspond to the time the robot requires to turn in place at the desired angle:

```
#define ROTATION_TIME 200
maxlight=0;


OnFwd(OUT_A);
OnRev(OUT_C);
Wait(ROTATION_TIME / 2);
OnRev(OUT_A);
OnFwd(OUT_C);
ClearTimer(1);
while (Timer(1)<ROTATION_TIME)
{
  light=SENSOR_1;
  if (light>maxlight)
  {
    maxlight=light;
  }
}
Off(OUT_A+OUT_C);
```

When you have determined the maximum reading, rotate the robot until it reads approximately the same value again.

Acting on the *ROTATION_TIME* constant, you can make your robot explore all directions 360° around it, or limit its search to a narrower sector.

Larger angles make your robot more tolerant and flexible, but they will slow down the search process.

In the mobile head version, you can even program your robot to scan the environment *while* it moves. In this case, we suggest you adopt a different strategy: read three values, one at the right (e.g., about 30°), one at the corresponding angle to the left, and one for the center. If the central value is the highest one, continue straight on; otherwise, slightly correct the angle of the route so the robot continues in the direction of the strongest reading. With this technique, the robot makes frequent small corrections instead of stopping to find the new route.

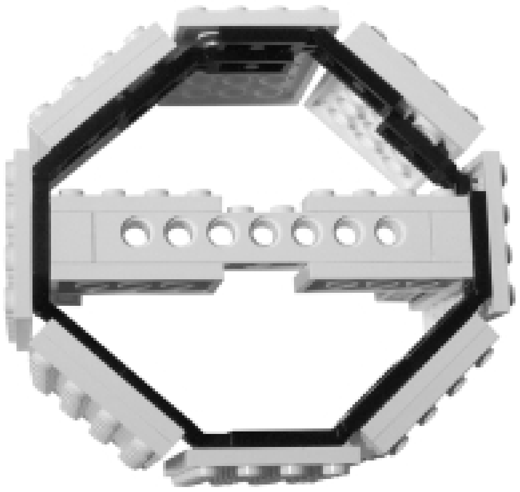
Variations on the Construction

Our R2-D2-style robot can be programmed for different tasks. For example, if you want to equip it for line following you simply have to put a light sensor (that faces down) just behind the pivoting wheel.

Adding bumpers, on the contrary, is not an easy job if you don't want to alter the esthetics of the model: R2-D2 is not very suitable for cumbersome bumpers. By using rotation sensors, you can perform indirect obstacle detection (see Chapter 4), an approach more proper for this droid than conventional bumpers. Proximity detection is a good alternative, if you own a custom IRPD sensor.

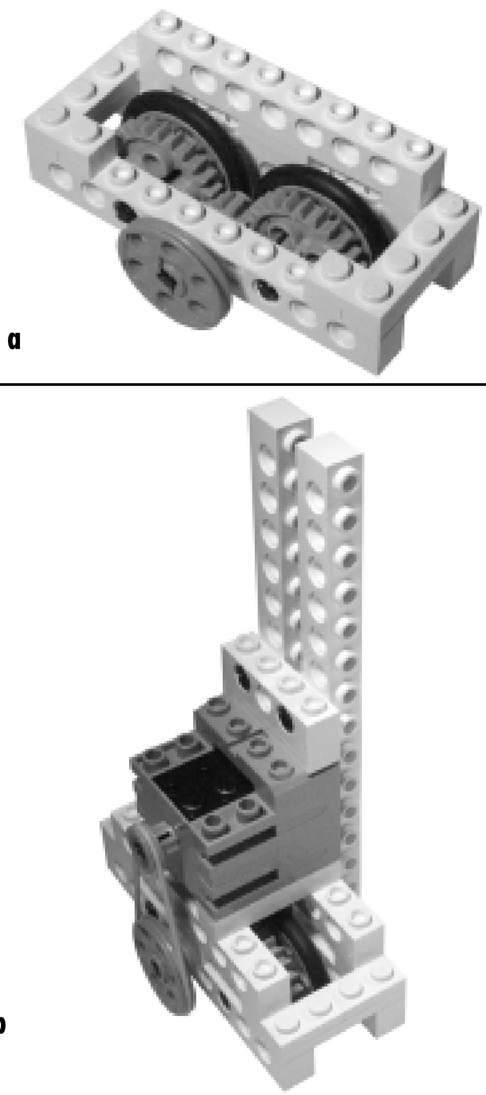
Having extra pieces can make your R2-D2 more alike to the original. For example, with some plates and hinges you can shape an octagonal body (Figure 18.10), a better approximation to its cylindrical body.

Figure 18.10 A Section of a Possible Octagonal Body



You can even remove the pivoting wheel and make R2-D2 capable of standing on two legs by simply placing two aligned wheels into each leg (Figure 18.11). This way, the robot is no longer a differential drive and becomes a skid-steer drive. To use this architecture, it's very important you keep the COG (center of gravity) as close as possible to the ground. Its vertical right should be in the middle of the support base, delimited by the four touching points of the wheels in order to reduce the tendency of the robot to overturn when starting or stopping. A high reduction ratio between the motors and the wheels helps, too.

Figure 18.11 A Double-Wheeled Leg

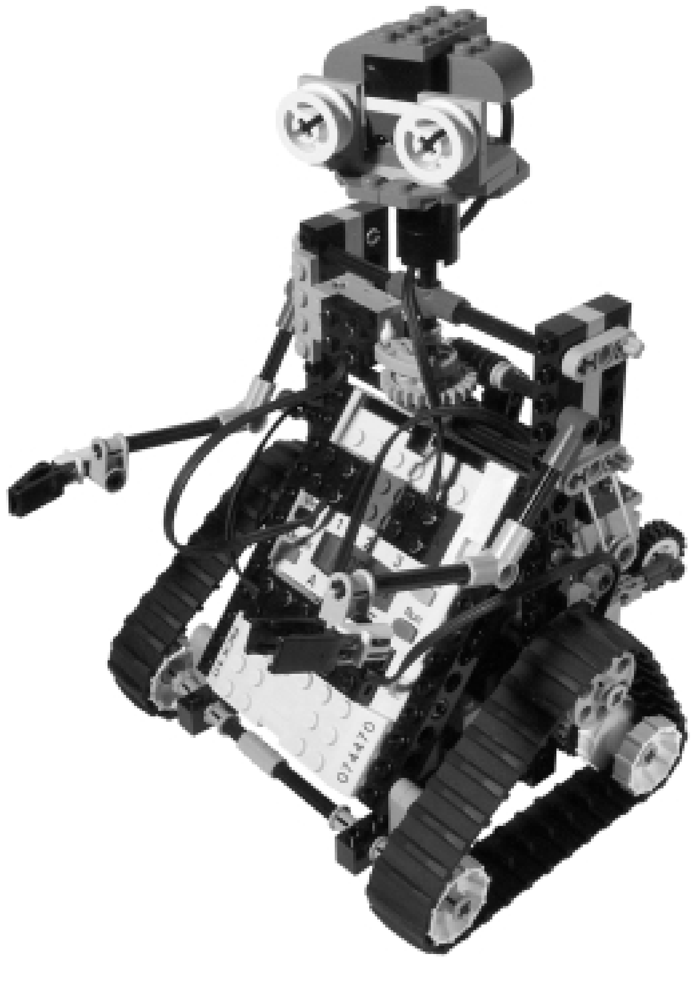


Building a Johnny Five-Style Droid

Johnny Five (or Number Five) has a much less compact structure than R2-D2. Its body is slim and articulated at many points, and the whole is supported by two large tracks. Replicating this in LEGO is quite a challenging task, especially because the RCX and the motors are rather large compared to the size of the tracks available in the MINDSTORMS kit. Things get better if you scale the model up, but you would need many extra parts and, above all, some larger tracks.

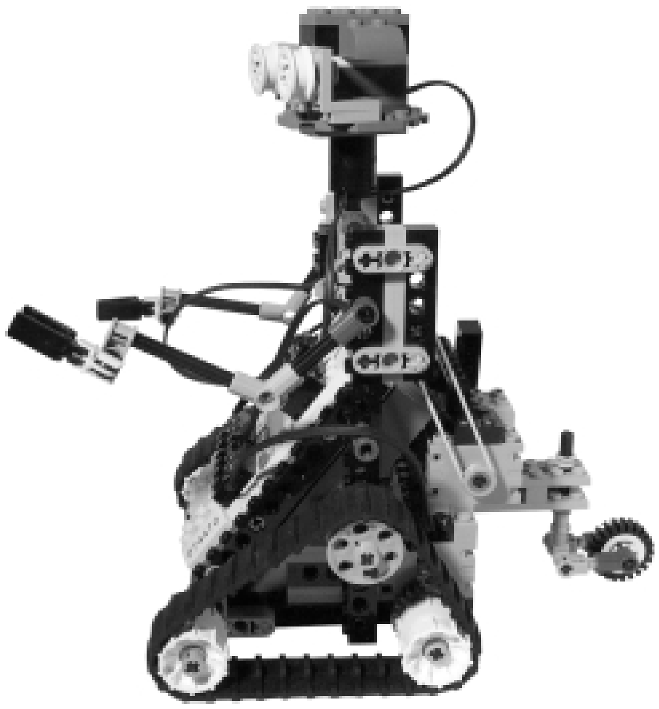
Since we can't have everything, we decided to be satisfied with just reproducing some of the main features of Johnny Five: the triangular tracks, the rear pivoting wheel, a rotating head and two (decorative only) hands (Figure 18.12).

Figure 18.12 Our Johnny Five-Style Droid



The body of Johnny Five has been built around a chassis with a triangular section. Looking at the robot from its side, you'll notice that three beams form a perfect right triangle with sides of length 6, 8, and 10 (Figure 18.13). The vertical 1 x 16 beam also serves as a support for the upper wheel of the tracks and the head mechanism. Since the MINDSTORMS kit includes only four track wheels, we made two more from a pair of pulleys with a bushing in the middle. The pivoting wheel is not actually necessary to support the robot, but it enhances its look.

Figure 18.13 Johnny Five Side View



The gearing of the drive motors is rather simple: an 8t gear on the motor shaft engages a 24t gear connected to the drive axles. (Remember that you also need a 16t gear inside the track wheel to joint it to the axle.)

The third motor lies on a second layer above the first two, and it's braced by a diagonal beam with a quite unconventional slope: this triangle has a base of 2 studs, a height of 7 1/3 bricks that corresponds to 8.8 studs, and a diagonal of 9 studs. The match is not perfect, but the error is less than three parts in a thousand and gives a solid bracing to the motors without disturbing the pivoting wheel (Figure 18.14).

Figure 18.14 Johnny Five Rear View

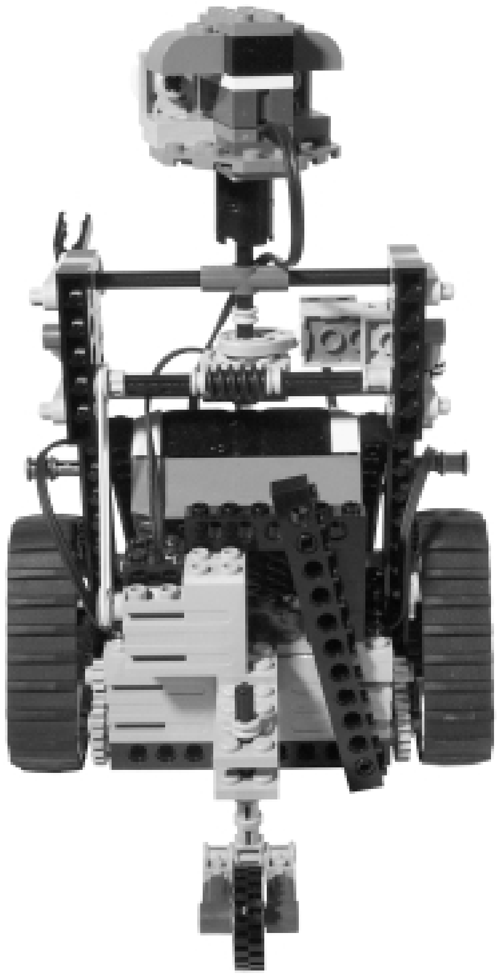
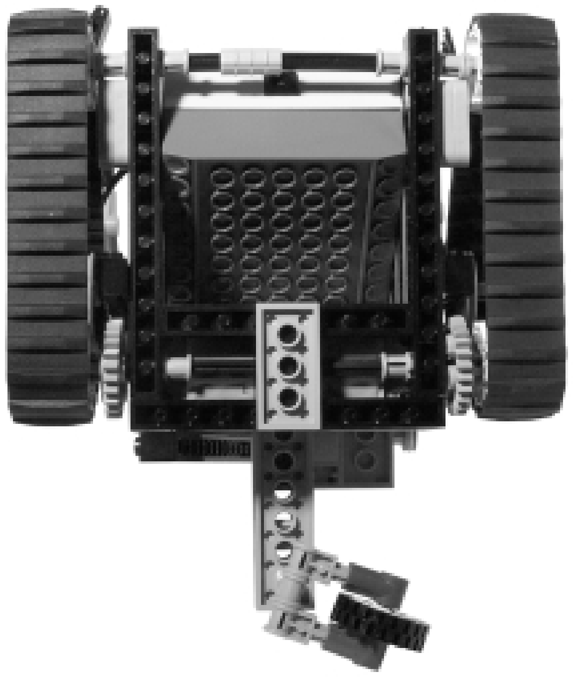


Figure 18.15 shows the bottom of the robot. You'll notice that we joined the front axles together to make them more solid, relying on the fact that the track wheels are free to rotate on them. On the other side, the rear track wheels have 16t gears inside. As explained in the MINDSTORMS Constructopedia, this is the way to securely join them to their axle.

Figure 18.15 Johnny Five Bottom View



The head mechanism is nearly identical to the one we designed for R2-D2: A pulley-belt system rotates a worm gear, which engages a 24t. A cam closes a touch sensor when the head is centered (Figure 18.16).

We got sentimental and rebuilt for Johnny Five the same head we designed together in 1998 for one of our first MINDSTORMS projects, called S3 (see Figure 18.17).

NOTE

Refer to the earlier section on programming the R2-D2-style droid when programming the Johnny Five robot—the two models can be driven by the same software.

Figure 18.16 The Johnny Five Head Mechanism

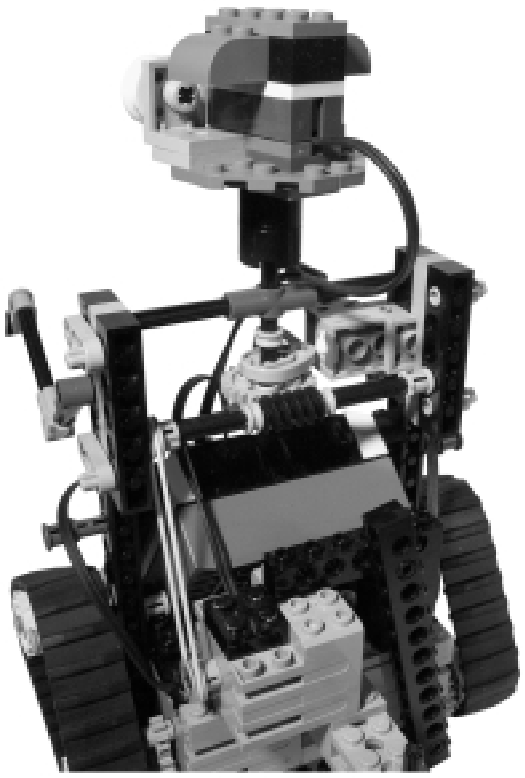
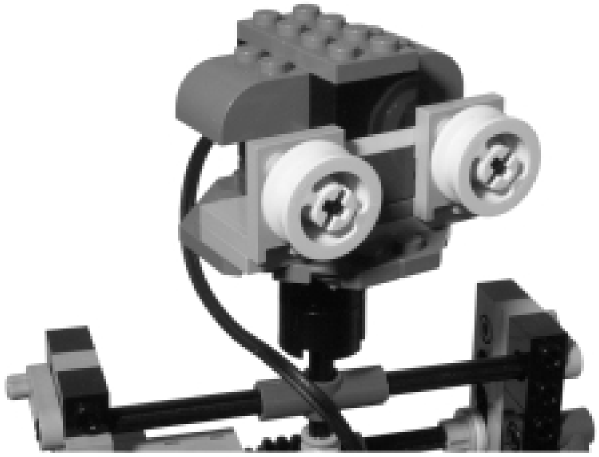


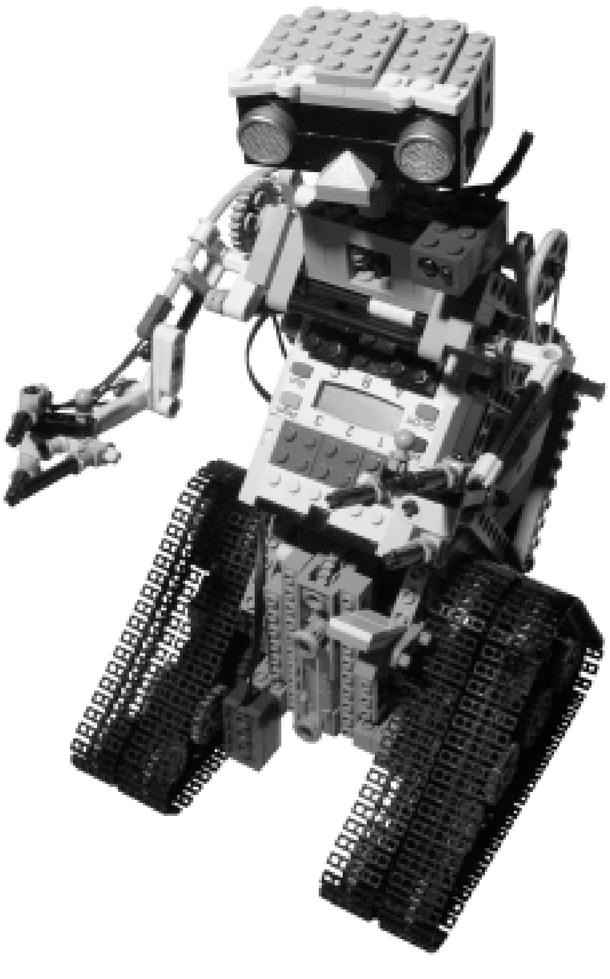
Figure 18.17 Close-Up of the Head



Variations on the Construction

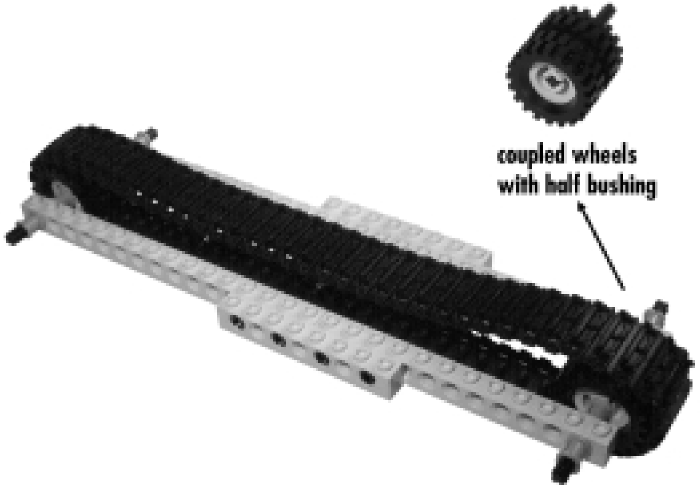
In introducing this robot, we explained that if you want to make your version more similar to the one from the movie, you have to increase its scale. You will need some extra parts, but those are easy to find. The greatest problem comes from the tracks: You can't use the ones from the MINDSTORMS kit for a larger Johnny Five, because they're too small and will make it look ridiculous. Staying with LEGO components you have two alternatives: the Cybermaster tracks and the modular chain link tracks (see Chapter 9), both of which are a bit hard to find. The latter represents a very flexible solution that allows you to adjust the length of the track precisely to your needs, and it's what we used for Cinque, the larger Johnny Five-styled robot described on our site (Figure 18.18).

Figure 18.18 Cinque, Our Replica of Johnny Five



If you're open to nonoriginal components, you can search toy shops for cheap toy tanks: some of them feature tracks that may be adapted to LEGO and might fit your needs very well. Usually, you cannot use the standard LEGO track wheels. Instead, you have to build suitable ones combining wheels in pairs with a half or whole bushing in the middle (Figure 18.19).

Figure 18.19 Nonoriginal Tracks



Bricks & Chips…

Guiding Infrared Light

Cinque was not our first dual-RCX robot—we had already succeeded in co-coordinating two RCX units through IR messages. However, after finishing Cinque, we realized that the two RCXs couldn't communicate because their IR devices didn't "see" each other.

Facing the horrible scenario of starting everything over from scratch, we began looking for a solution to guide the IR light between the RCXs. IR light, though not visible to the human eye, behaves just like visible light, so what worked with visible light would have worked with the IR, too. Our first idea involved LEGO optic-fibres, the ones usually employed together with the FOS unit. We tried to position them in front

Continued

of the RCX bricks, but that didn't work. Then we experimented with a mirror, placing the robot in front of it—and found the IR messages could indeed successfully reach both the units. We were close to the solution; we simply needed a small mirror mounted on the robot. But did it really have to be a mirror, we wondered, or would something easier work?

Breathing sighs of relief, we finally discovered that a simple white reflecting surface was enough to assure a reliable communication. You can see our reflector in Figure 18.18: two white tiles close to the top of the left track.

Summary

If you decide to reproduce one of the famous robots that populate sci-fi movies, you will face difficulties similar to what we described in Chapter 17 about making robotic animals: matching the form to the function.

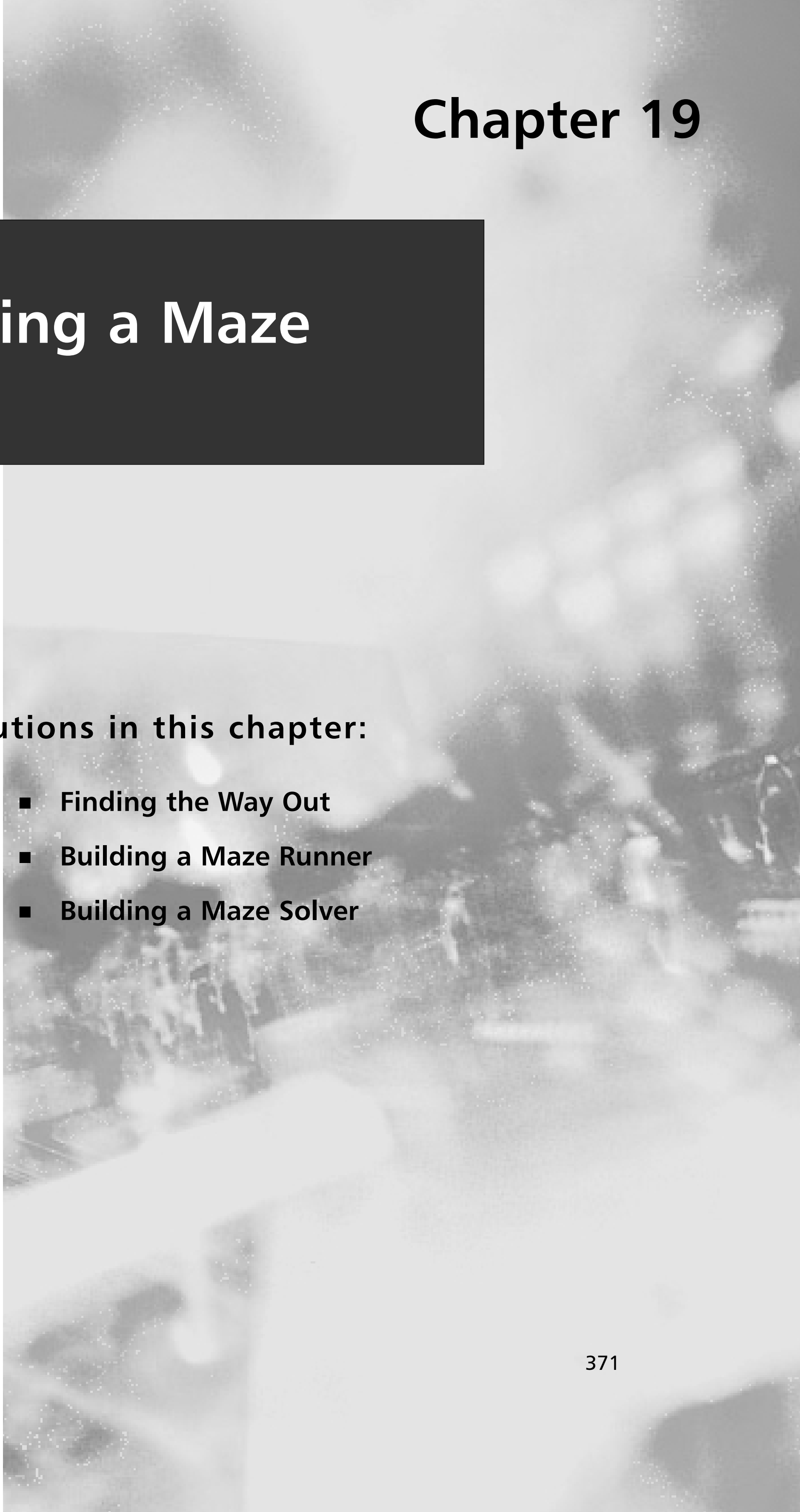
The process can be made a bit easier by choosing the proper scale for the model. Generally speaking, the bigger the size, the better the result, because the size of your elements become less relevant when compared to the size of the model, allowing you to make finer details. Unfortunately, sizing up is not always an option, because you must take into account your own part availability, and the size of some special components, like wheels and tracks, that limit the maximum dimension you can aspire to.

On the technical side, both the droids gave you the chance to see some of the theoretical concepts of Part I put into practice. For example, the vertical shape of R2-D2 requires the thoughtful application of the ideas expressed in Chapter 5 about balancing the robot to oppose the effects of inertia. The Johnny Five model is the first robot of Part II to use the triangular structures described in Chapter 1. It is also the first one that uses tracks instead of wheels, implementing the skid-steer drive scheme described in Chapter 8. To make its tracks outline a triangular shape, we had to build a third pair of track-wheels; this is a good example of the powerful modularity of the LEGO system, which allows you to replicate the functionality of one part by using other basic elements.

This chapter also introduced you to a programming challenge we haven't discussed yet: light following. It has significant differences from line following, because you cannot rely on the constant readings that come from a black and white pad. Instead, you have to scan the environment looking for the strongest light source, and then follow that direction. For line following, we suggested a calibration procedure be executed before running the robot along the line in order to evaluate the maximum and minimum values the robot should expect. In the case of light following, this kind of procedure is performed every time the robot wants to decide in which direction it should go.

We invite you to visit some of the Web sites listed in Appendix A. Most of them will be of great inspiration when it comes to making your own droids.

Solving a Maze



Solutions in this chapter:

- **Finding the Way Out**
- **Building a Maze Runner**
- **Building a Maze Solver**

Introduction

Humankind has always been fascinated by labyrinths, and mythology is crowded with heroes busy finding their way out of mysterious buildings. It was not unusual for large European 18th- and 19th-century villas to have a hedge labyrinth in their garden. Indeed, mazes of different varieties are still common in the amusement parks and games of our era.

The ability to find your way through a maze is considered a good test of intelligence and has been used with mice and other animals to measure their capacities. Now the time has come to test your robots, too!

Before building robots capable of solving a maze, you must understand what "solving a maze" means. In other words, we must understand what knowledge and skills are necessary to find the way out. If you ask anybody to solve a simple maze drawn on a sheet of paper, he or she will probably do it very quickly. But if you ask someone to *describe* the procedure they used, you will likely receive some very generic explanations. This happens because human beings tend to ignore the details of what they do: They employ the knowledge and experience accumulated throughout their life—especially during their childhood—without realizing that such a simple action actually hides a multitude of operations. If somebody were to stop you on the street to ask for directions, would you explain to them what "turn" and "left" means? Surely not. However, in regards to robotics, there's no background knowledge you can take for granted. We explained in Chapter 14 that even an apparently easy task like moving around the inside of a room, or detecting obstacles, requires a thoughtful analysis of the environment and of its interactions with your robot.

This is also the kind of analysis necessary to implement maze solving: you need a strategy, and it has to be detailed enough to be translated into program instructions for your robot. For this reason, we will begin exploring some theories about maze solving, which will lay the foundations for the projects that follow.

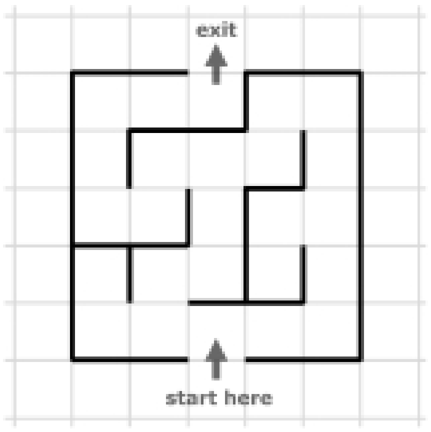
On the hardware side, the robots that you will come across in this chapter don't require many more parts than what you find in your MINDSTORMS box. We built the Maze Runner robot entirely from MINDSTORMS parts, while the Maze Solver robot used some additional elements unnecessary for the success of the first project. As well as teaching some concepts about maze solving, this chapter will also strengthen your skills about working with touch and light sensors, consolidating ideas that appeared in Chapter 4.

Finding the Way Out

Even a simple maze, the kind you can solve in a few seconds with a pencil if you see it printed on a sheet of paper, assumes a completely different perspective when you are inside it. If you don't have any external reference point, and are not allowed to take note of your moves, well, be prepared to spend a few hours!

How can external references or note-taking help you in finding your way out of the maze? Because they help you understand where you are. To introduce this concept, we invite you to perform an experiment: You need a friend who will play the role of the robot inside the maze, while you simulate the sensors that return information about the environment around him. Your friend must find the exit from the maze of Figure 19.1 without actually seeing the picture, only by using your verbal feedback. He can only use four commands inside the maze to direct himself: forward, back, right, and left. You track his position in the maze with a pencil, and if his command is acceptable—that is, if the desired direction doesn't come up against a wall—you move the pencil to the specified adjacent square, answering "OK"; otherwise, you keep the pencil stationary and answer "wall."

Figure 19.1 The Test Maze



Will your friend be able to exit the maze under these conditions? Probably yes, but only after a long time, and with an effort that seems enormous when compared to the simplicity of the maze. In the second phase of the experiment, provide your friend with a squared sheet and a pencil, so he is able to log his movements. When you answer "OK," he will move his pencil to the adjacent square, too, and when you answer "wall" he will remain in the same square, but will mark the specified side of his square with a line which represents the wall. Now things will go much more smoothly for your friend: Looking at his map, he can avoid visiting the same location more than once, sparing himself many collisions and exploring all possible routes until he finds the way out.

Some of you may have noticed that the aids mentioned pertain to the two basic categories described in Chapter 13 regarding knowing your position: absolute and relative positioning. In fact, the use of external reference points represent an application of absolute positioning—you use landmarks to locate yourself—while note-taking has many similarities with relative positioning: You deduce your new location knowing the direction and the distance you covered from the previous location.

Finding one's way in a labyrinth is, in fact, a special case of navigation and requires similar abilities, with the addition of some memory to remember which branches have already been visited. In our previous experiment, the memory was symbolized by the sheet of paper where your friend logged his moves.

Thus, generally speaking, to solve a labyrinth, your robot should be equipped with a navigation system and a map in its memory. There are some notable exceptions, like labyrinths that simply require slavish application of a rule to lead you to the exit, which could be handled by robots with less demanding equipment.

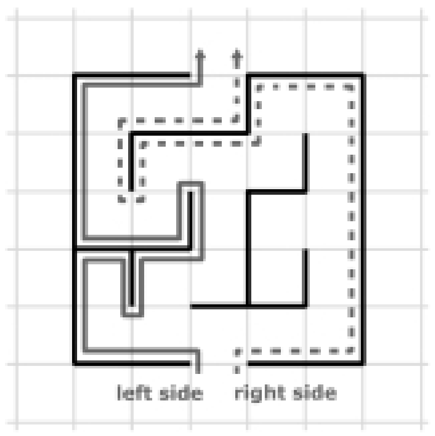
The strategies we are going explain work with flat mazes—not just the ones you can draw on a piece of paper, but any labyrinths that can be *represented* on a piece of paper. For example, hedge and crystal labyrinths usually belong to this category, provided that they don't contain any bridges or tunnels.

Using the Left Side—Right Side Strategy

This technique solves an incredibly large class of mazes, its rule being quite simple to remember and apply. It states that, when applicable, if you follow the left wall and turn left whenever possible, you will find the exit. Easy, isn't it? You're not guaranteed to cover the shortest distance, but you're guaranteed to find the way out. Actually you can just as easily keep to the right side, the two methods being complementary and leading to the exit along different paths. We invite you to test the rule on the simple maze of Figure 19.1. Imagine physically entering the maze and then trying to follow the left wall—eventually, you arrive at the exit. Now try again, this time following the right wall. Again you reach the exit, but from a different route (Figure 19.2).

To be more precise, if you follow the right wall, you use the same route you would if you followed the left wall from the exit to the entrance.

Figure 19.2 Following the Right and Left Walls



This strategy has a great advantage in that you need not know anything about your position and orientation. The only abilities required are that your robot can follow a wall and that it can recognize the exit when it's there.

At this point, the crucial question is: When can you apply this rule? There are essentially two cases in which you can do this:

1. When the maze is flat, and has both the entrance and exit placed along its perimeter (as in Figure 19.2).
2. When the maze is flat, and the entrance and exit are points arbitrarily chosen anywhere in the maze, where the latter doesn't contain any loops. That is, it doesn't contain multiple paths that connect any two points (Figure 19.3).

The rule covers many practical cases. It doesn't work when the entrance and exit are not along the perimeter *and* the maze contains loops, as in Figure 19.4. Notice that the route covered following the left wall brings you back to the entrance without reaching the exit point.

Figure 19.3 The Exit Is Inside a Maze with No Loops

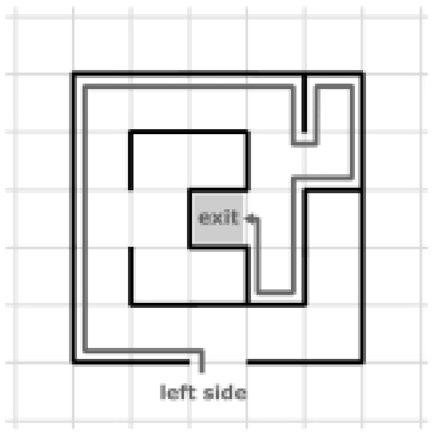
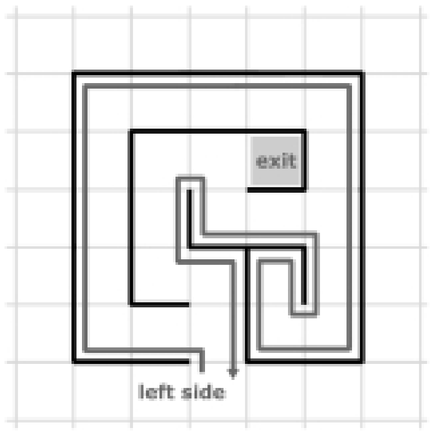


Figure 19.4 The Exit Is Inside a Maze with Loops



Applying Other Strategies

When you cannot apply the rule previously stated, you rely on two strategies:

1. Executing random turns
2. Tracking your route

The first one says that whenever you find yourself at an intersection, you decide which way to go at random. Though this method is guaranteed to find the solution sooner or later, that "later" can be a very, very long time if the maze includes more than a handful of intersections!

The second approach solves the more general case of mazes with more than a few intersections, but requires two valuable ingredients: a position control system and a memory. You must be able to recognize each intersection and mark the branches already explored so as not to explore them again. The right side rule can still be useful as a basic rule, but when you find yourself in a place you've already been, you must be able to backtrack to the first intersection with unvisited branches and take one of those.

We imagine you already see the difficulties in this: You must provide your robot with an affordable navigational aid and with an inner map to represent the maze so you can mark the visited corridors. Don't worry, this time we won't test your patience with trigonometric functions and dead reckoning. If you read on, you'll see that we suggest a Maze Solver that doesn't require anything but the basic MINDSTORMS equipment and some programming skill.

But let's start with something simpler. The first robot of this chapter, the Maze Runner, has been designed to apply the left side rule inside a maze.

Building a Maze Runner

The first robot of this chapter applies the left-side rule and follows the left wall of the maze toward the exit. It has no intelligence, only an ability to follow a wall.

Constructing the Maze Runner

To construct the Maze Runner, we used a differential drive configuration and a couple of touch sensors. The whole robot may be replicated with parts solely contained in the MINDSTORMS set (Figure 19.5).

It works on a very simple principle: one side sensor "feels" the wall, so the robot can always remain in touch with it, turning left when necessary. This covers the case of straight walls and of left turns, but the robot will also have to face situations where it hits a wall in front of it and must turn right. For this reason, we equipped it with a second sensor, that detects front collisions. In Figure 19.6, you see the robot without the RCX, and can distinguish the two touch sensors, both kept closed by the pressure of a rubber band.

The left side bumper, the one with the horizontal wheel at its end, is designed to touch the wall, while the other detects the closed corners that require a right turn.

Figure 19.5 The Maze Runner

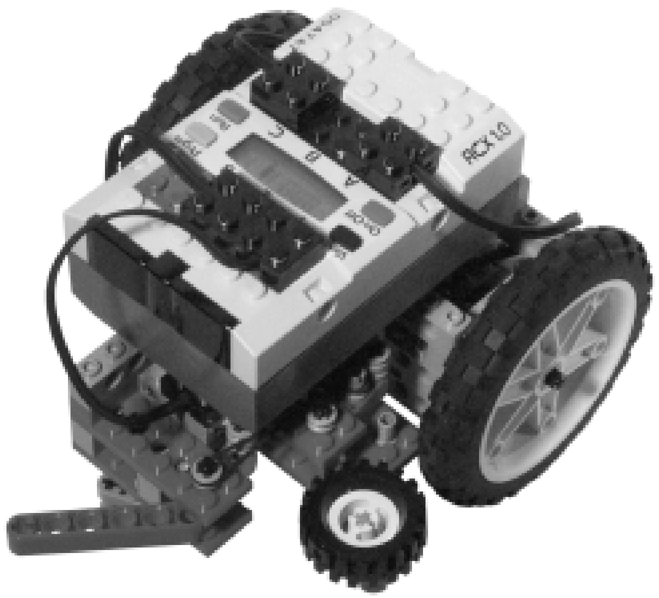
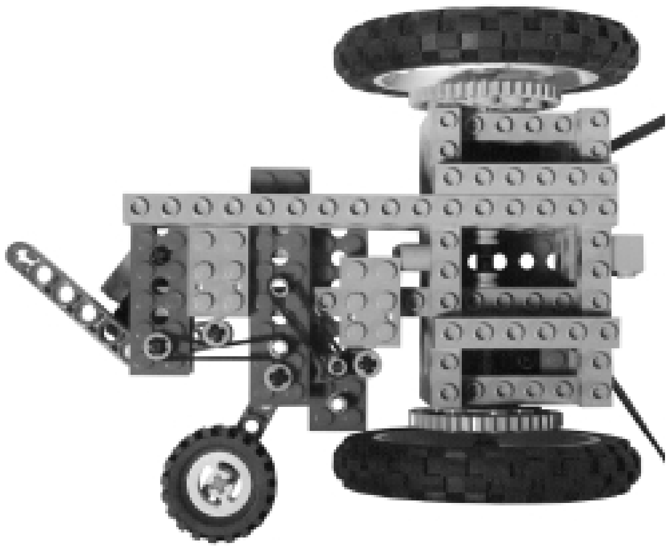
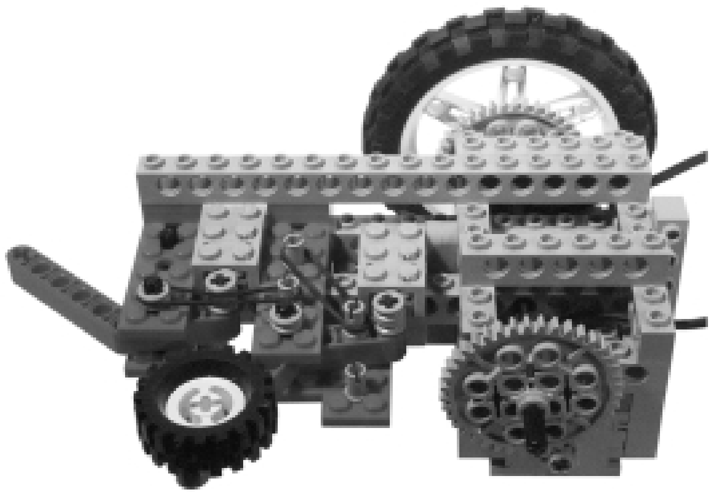


Figure 19.6 Top View (RCX Removed)



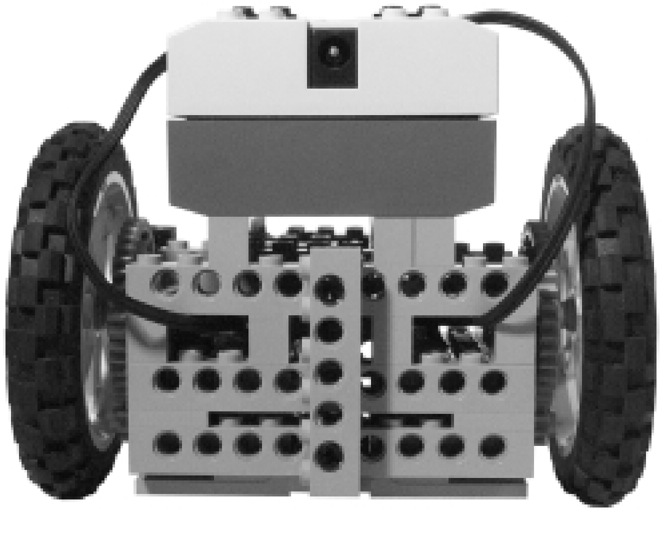
For the differential drive, we used one of the simplest configurations shown in this book: A single stage geartrain made out of a 40t attached to the wheel and an 8t connected to the motor shaft (Figure 19.7).

Figure 19.7 Left Side View (Drive Wheel Removed)



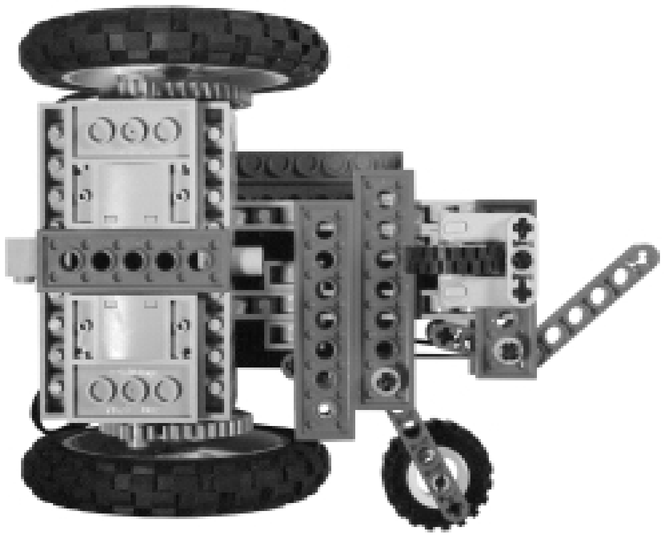
Our robot is very low to the ground because we placed the motors below the beams that support the wheels, but in this kind of task we don't need a lot of ground clearance and this solution keeps the assembly nicely compact (Figure 19.8).

Figure 19.8 Rear View



The front wheel is a simple caster identical to the one we used for the R2-D2 droid in Chapter 18 (Figure 19.9).

Figure 19.9 Bottom View



Programming the Runner

"Playing robot" is always a great exercise for devising or testing the strategy you are going to implement in your program. Even before actually writing any code, imagine running the program in your head, and try to explore the test maze of Figure 19.1 following the instructions step by step.

You will discover that this robot is particularly easy to program. After having initialized the sensors for the proper type, all you have to do is go straight while both sensors are closed. If the left sensor opens, turn left until it closes again. If the front sensor opens, turn right.

In our version of the code, we gave priority to the front sensor, that is, the robot ignores the side sensor until everything is all right with the front one. We also introduced some timing to improve the performances: We added a quarter of a second before turning left, because this gives the rear driving wheels time to reach the optimal point to turn around a corner. A quarter of a second is also the length of the right turn before starting left-side checking again.

Creating the Maze

Now that you have a maze runner, you presumably would like a maze, too! Unless you want to show off your robot at some sort of public exhibit, it's not necessary to build a lovely structure made from wood or other materials. You can test your creature against a makeshift labyrinth made with small pieces of furniture, piles of books, large boxes, cardboard and everything else your imagination will suggest.

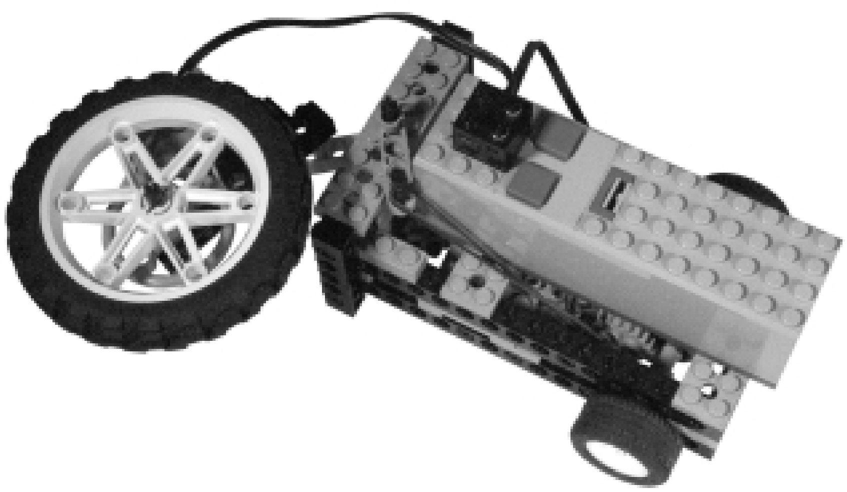
The only thing to make sure of is that all the "walls" have a smooth surface at the height corresponding to the sensors, so they don't block them during the robot's run.

Variations on the Maze Runner

With our maze runner finished and successfully tested, we wondered whether it was possible to build a simpler version of the maze runner, a purely mechanical wall follower, that didn't need any programmable unit to be run. We found one.

Figure 19.10 shows what we came up with. The machine is a steering drive, employing a single motor. We powered it from a battery box, to emphasize the fact that it needs no intelligence, but you can run your version from an output port of the RCX. The trick to this robot is that the large horizontal bumper wheel is used to control the steering wheels. The bumper wheel, itself, is powered by a second motor.

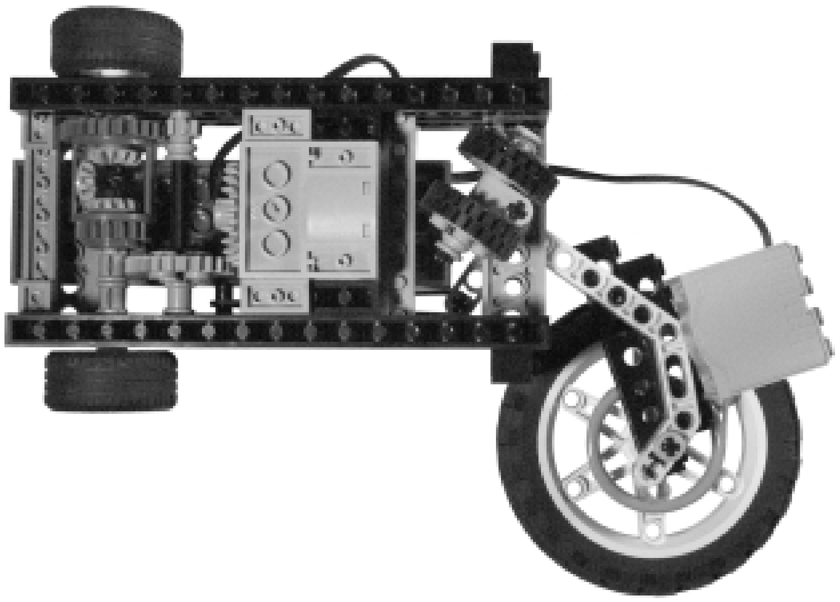
Figure 19.10 A Purely Mechanical Wall Follower



The front wheel assembly is pulled left by a rubber band. When the robot departs from the left wall, the rubber band pulls the steering assembly, and the robot turns left until it is again parallel to the wall. When the robot runs into a front wall, the front wheel rolls onto it pulling the steer right, and in a few seconds the robot is aligned again to the wall.

The bottom of the robot shows its simplicity: a differential drive for the drive wheels, and two free wheels for the steer (Figure 19.11).

Figure 19.11 Bottom View



This robot doesn't work as well as the previous maze runner. In most cases it behaves adequately, but cannot manage all situations. For example, it has problems turning left 180° around a thin wall, or in making two close 90° right turns. Nevertheless, it shows an important principle, that there are many solutions to the same problem, and that sometimes you can work a bit more on the mechanical side to make your software simpler.

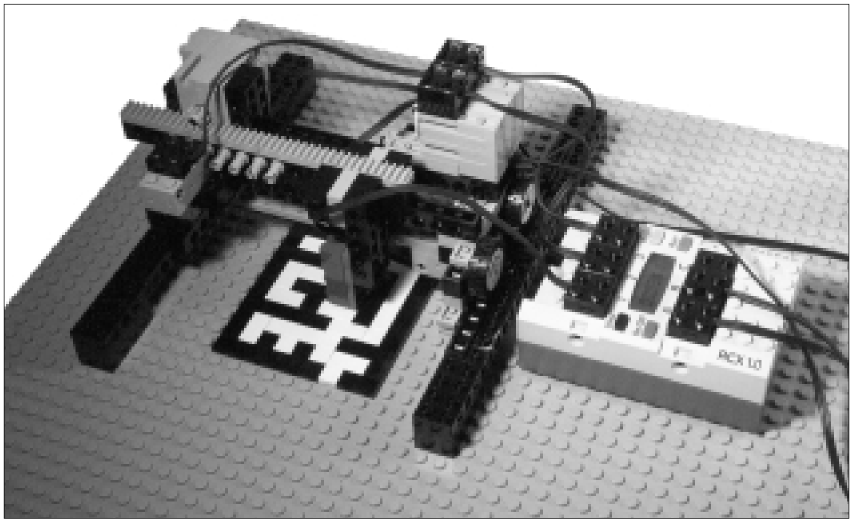
Building a Maze Solver

To overcome the limitations of the Maze Runner and its "left side rule" tactic, and to solve the more general case of a labyrinth with an entrance and exit at two arbitrary points, we designed this Maze Solver.

Constructing the Maze Solver

Even from the first picture of our Maze Solver (Figure 19.12) you can see that we radically changed the approach: The robot is no longer inside the maze. On the contrary, it stays entirely outside and uses a light sensor to analyze and solve it. This makes the robot similar to a human who looks at a maze on a sheet of paper.

Figure 19.12 The Maze Solver



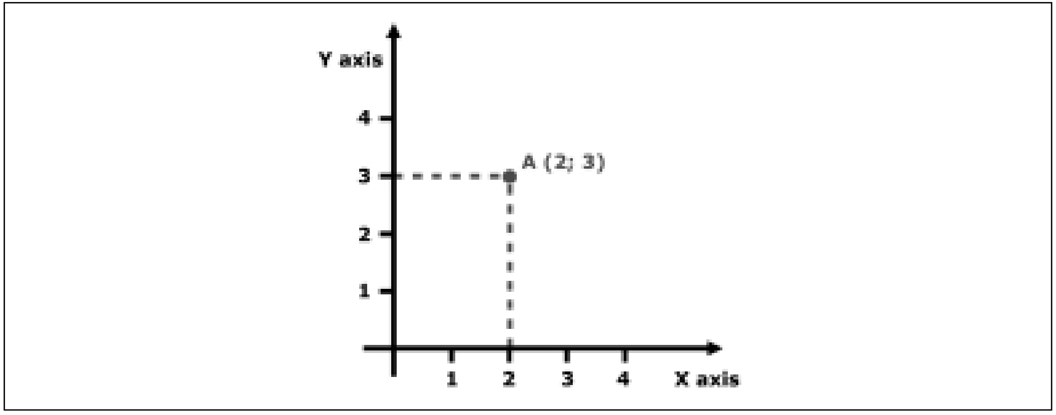
Obviously, the maze in this case is just a flat maze. We used LEGO tiles to draw one on a large baseplate, but you can just as easily draw one on a sheet of paper. Except for the tiles and the baseplate, you will find all the other parts in your MINDSTORMS set.

The idea that forms the basis of this robot is a reading head that moves in two perpendicular directions. We can define this as a *Cartesian* system, because, like in a Cartesian plane, each position is defined by a pair of coordinates, traditionally identified with the letters *x* and *y*. For example, the point A in the Cartesian plane of Figure 19.13 has coordinates x = 2 and y = 3.

In the Maze Solver, the reading head moves over two rails, which correspond to the x and y axis of the graph. This is how the robot tracks its position, a system which has the significant advantage of applying very simple math.

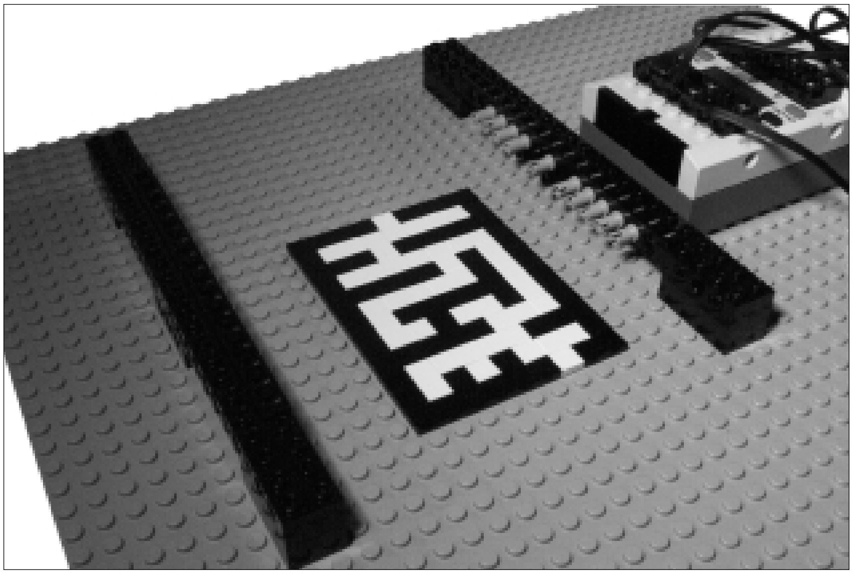
Besides controlling the coordinates of the reading head, the robot monitors a third quantity. The readings that come from the light sensor, which reflect the state of the underlying cell of the labyrinth.

Figure 19.13 The Cartesian Plane



The test maze we used to try out our robot can also be solved with the left side rule (Figure 19.14), but this robot, when properly programmed, is able to solve any kind of flat maze.

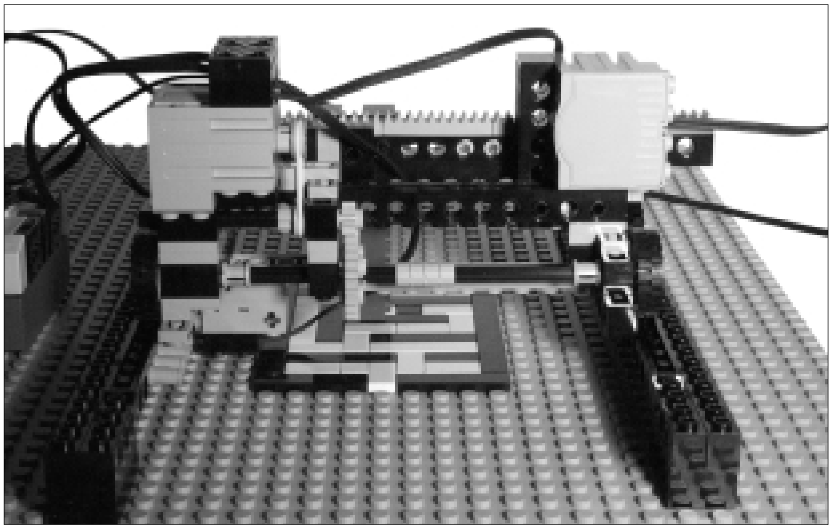
Figure 19.14 The Baseplate, the Rails, and the Maze



In the same picture, note the two rails the robot moves upon. The inner beams are mounted studs-down to provide a smooth surface the robot can run on. On the right side, the beam features a row of pegs that triggers the y-axis touch sensor and allow position control. You can think of the pegs as marks along the axis, placed at a distance of one unit each; the robot detects these marks with a touch sensor and can stop precisely at any of them.

Looking at the rear side of the robot, you notice the y-motor and the corresponding touch sensor at the very bottom (Figure 19.15). The left and right rear wheels are connected with a long joined axle.

Figure 19.15 Rear View



The front side shows the light sensor attached to a 1 x 16 beam covered with racks (Figure 19.16). This is the x-axis, and on the left of the picture note the motor that controls this direction of motion.

Most of the trolley has been built studs-down, mainly to provide a smooth surface which the rack can slide over (Figure 19.17). The rack mounts a second series of pegs to control the x coordinate through a touch sensor with the same mechanism adopted for the y-axis. We used the long cables to wire the motors and the short ones in pairs to connect the touch sensors.

The bottom view reveals the last details (Figure 19.18), the touch sensor and the long plate that gives rigidity to the structure.

Figure 19.16 Front View

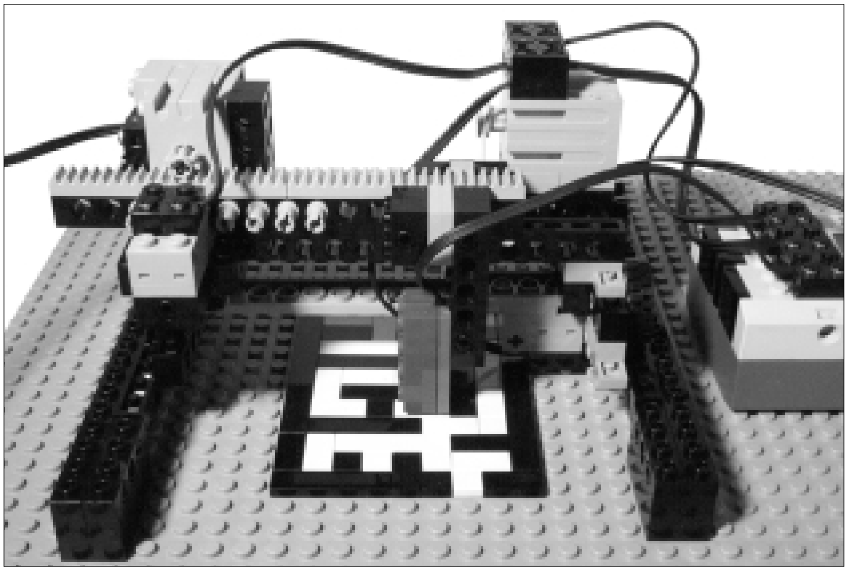


Figure 19.17 Top View

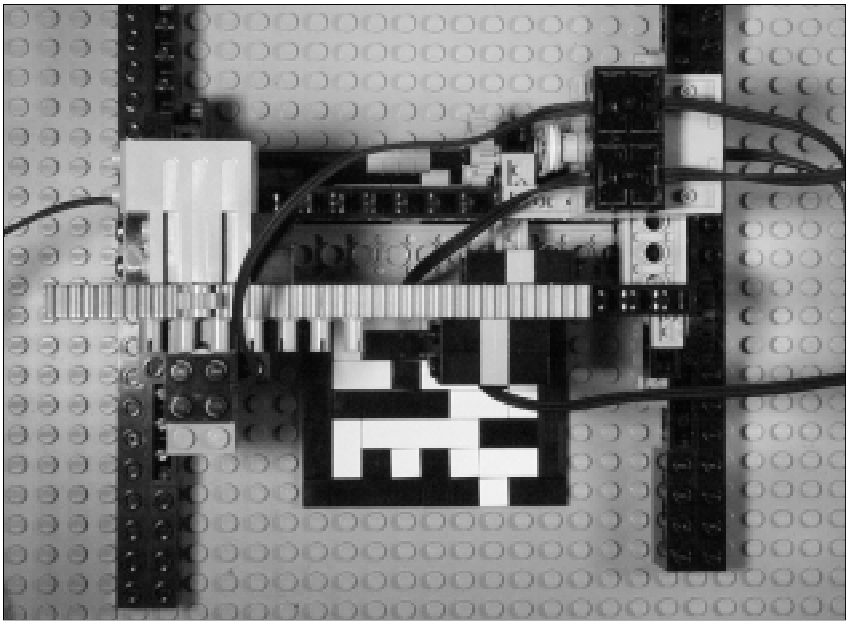
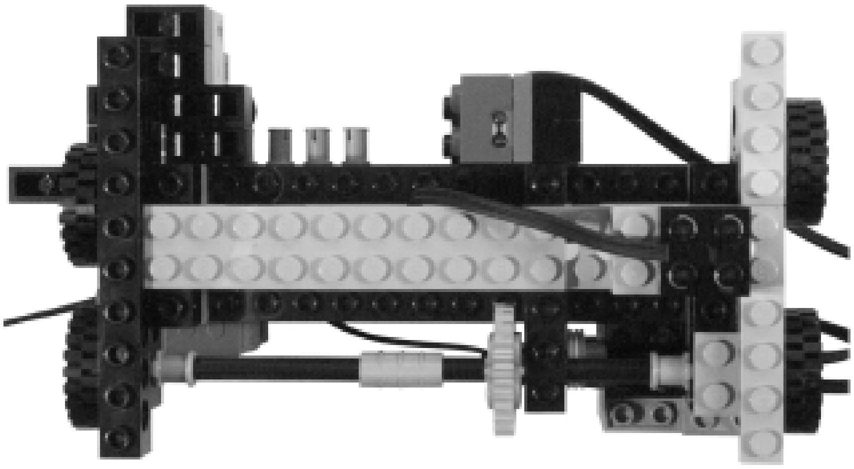


Figure 19.18 Bottom View



Programming the Solver

Our robot can move on a grid of twelve by seven possible positions, thus it's able to explore a labyrinth with 84 cells. We chose to represent the maze with LEGO tiles and, for this reason, we have two kinds of cells: *open* (white locations), which correspond to the path, and *closed* (black ones), which represent the walls.

Different representations are possible: Using a maze drawn on a piece of paper, you can make cells all white with black lines, designating the walls between them. We used a similar depiction for our maze in Figure 19.1.

NOTE

If you draw your maze on a piece of paper, make its cells exactly one stud wide, as this is the standard step of the robot (the pegs that activate the touch sensor are spaced one stud apart).

The way you draw your maze affects its representation in memory. The more obvious choice is an array of variables that correspond to the cells of the maze. In our case, you can use a very simple approach where each cell has only one of three possible states: white, black, and unknown. More complex representations are possible; for example, one where each cell contains information about its neighbors, typically using one bit for any of the four sides of the cell.

Bricks & Chips…

Struggling with Limited Memory

If you're using NQC or other environments based upon the standard firmware, you're probably thinking that you don't have 84 variables to use. Luckily, you don't need that many of them, because there are much more compact ways to store the information you need.

Thinking in terms of bits, two of them are enough for a cell (you need three states and they provide four: 00, 01, 10, 11), thus you can store a whole row of the labyrinth in 14 bits. This means that a row fits in your 16 bit variables with no problem, and the entire labyrinth doesn't require more than 12 variables.

The technique of addressing single bits of a variable require you have a bit of confidence with programming, and with bit masks and bitwise binary operations in particular. Any good programming text will help you understand how this mechanism works.

When you have implemented the storing and retrieval system, you're ready to design your exploration strategy. Here, you have two groups of approaches that are radically different:

- Scan the entire maze, store it in memory, solve it, and show the solution.
- Explore the maze as if the robot were inside it.

In both cases, you must tell the robot which cell is the goal, usually storing its coordinates in the program so the robot knows when it has found it and stops.

If you prefer the first technique, you have to program your Maze Solver to address each cell of the maze, row by row and column by column, reading the state of the cell and storing it in the internal map. Then, you can use a well-known algorithm, like the Bellman Algorithm, that finds the shortest path from the start to the goal; this is a purely computational process that happens inside the RCX. Finally, you can show the result of the process using the robot again, moving its head at the starting point and making it follow the optimal route to the exit. Though interesting from a computational point of view, in our opinion this is a bit like cheating, as if you were allowed to look at the plan of a labyrinth so you can prepare a map before actually entering it.

In the second approach, you proceed cell by cell, building the map *while* you explore the maze, as if you were inside it. Recall the note-taking technique we described in the Finding The Way Out section; this is exactly what you need. While you proceed in the maze, you update the map with the information that comes from the light sensor. You can base your strategy on some basic rule, like going straight whenever possible, or following the left wall, managing as exceptions the cases where you find yourself blocked or on an already visited path. In such cases, you should backtrack to the first cell with unexplored neighbors and restart from there. Use your stored map so you don't visit any cell more than once.

Summary

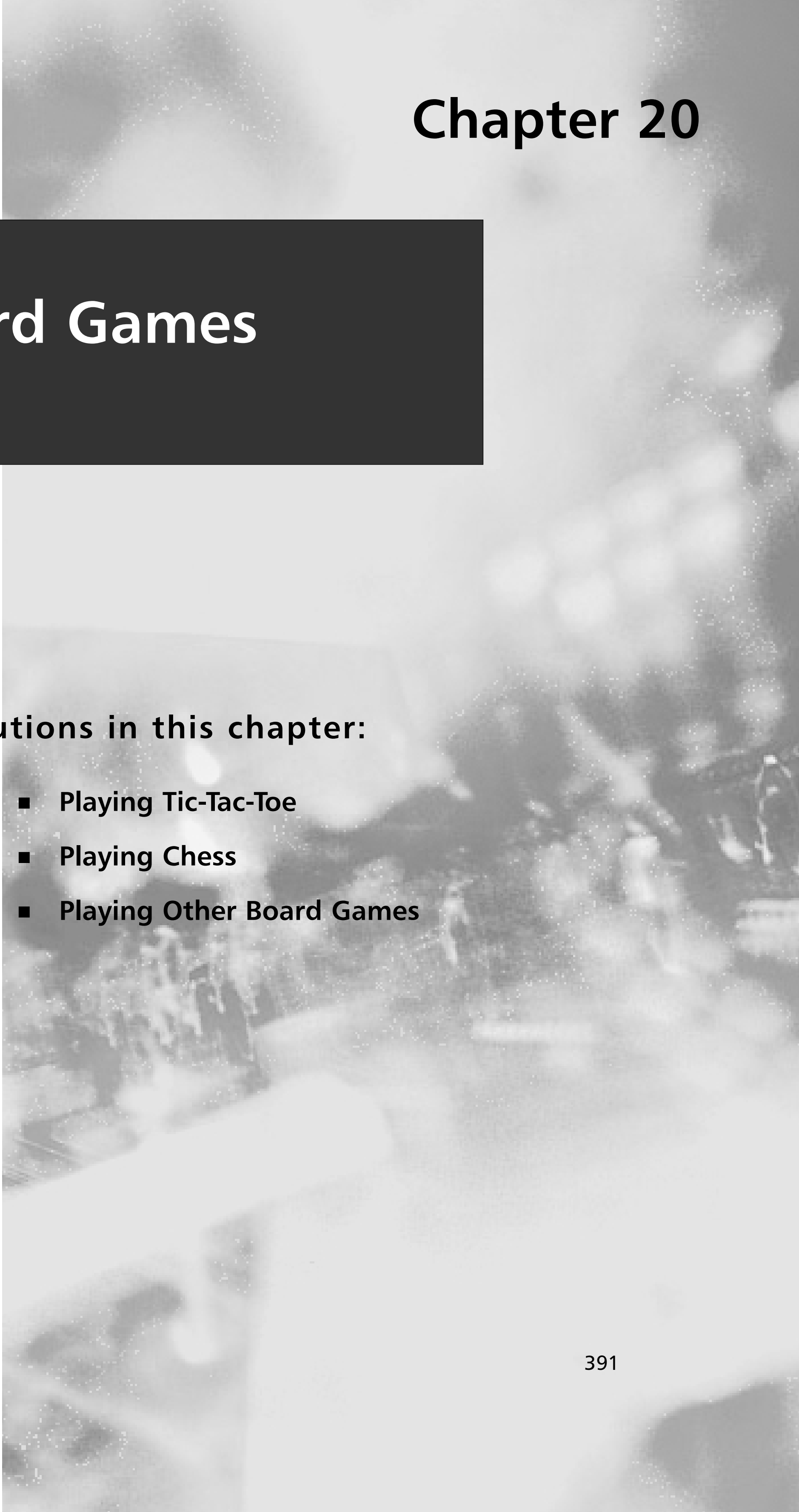
If you want to test your skills in maze solving, the first step you have to take is to understand the details involved in the process of finding the way out. We encourage you to draw a simple maze on a sheet of paper and to "play robot" with it: Take a pencil which represents the position of the robot in the maze and move it according to the "program" you execute in your head. This preliminary study will provide you with the necessary knowledge to successfully build and program your robot.

The robots of this chapter prove that maze solving is in the range of MINDSTORMS robotics, though here we purposefully escaped some major difficulties. In discussing the theory, we explained that maze solving requires a robot with both an accurate navigation system and a memory to store a map of the labyrinth. The navigation system is the more demanding of the two requirements (recall Chapter 13 and the problems involved in finding the robot's location). Both robots described in the chapter avoid this ugly necessity by employing different strategies.

The most important message you should get from the chapter is that sometimes you can look at your problem from a different perspective to find an easier solution. In fact, we discovered in this chapter that maze solving is no more complex than wall following. This means your robot needs only minimal intelligence—a trait reflected in our Maze Runner robot. Going a step further, you can transfer part of the control to mechanics, making the brain of your robot even less important.

The second robot, the Maze Solver, was aimed at solving flat labyrinths of greater complexity, including those which cannot be worked out using the wall-following approach.

This time, our robot did need a navigation system, but looking at the problem from a different angle again, we discovered a trick to avoid some of the mechanics—moving the context from a subjective perspective, that of a robot that runs along the corridors of a labyrinth, to a more controlled space where a reading head moves inside a Cartesian plane. The rails equipped with pegs provided the robot with an easy method of knowing exactly where the light sensor was inside the maze, excluding the complications of odometry and calculations. Thus the "know where you are and where you are going to" problem was reduced to counting pegs with a touch sensor.



Chapter 20

Board Games

Solutions in this chapter:

- **Playing Tic-Tac-Toe**
- **Playing Chess**
- **Playing Other Board Games**

Introduction

Board games are among the most challenging projects you can take on with your MINDSTORMS kit. The RCX does have the power to run software that plays Tic-Tac-Toe, Checkers, and even Chess (at some level), but this doesn't mean that such a program is easy to write and test.

We believe that the hardest but most important part of the job is the creation of the interface between your robot and the physical world. Though running a Chess program on the RCX is quite a challenging task in itself, having a robot physically interpret and interact with the data is another giant step.

The method you choose to represent the board and make your robot actually move the pieces determines in large part the technical difficulties you will have to solve. In this chapter, we are going to describe some of the possible approaches, from the easiest to the trickiest.

Playing Tic-Tac-Toe

As we described in our introduction to the book, in October 1999 we attended the first Mindfest gathering at the Massachusetts Institute of Technology (MIT) at the Media Lab facility.

The Mindfest event featured many activities: lectures, workshops, a construction zone... There was also a large exhibition area were the participants could show off their MINDSTORMS creations.

A couple of months before the event, we had already booked the plane and the hotel, and asked for a table in the exhibition area, but hadn't yet prepared our robot. Showing brilliant intuition, Marco Beri, the third member of our small group, came up with the idea of building a Tic-Tac-Toe machine, a robot able to play a board version of the well-known game. We immediately felt it was the right idea: Board games have been historically considered a good test for machine intelligence, so even if Tic-Tac-Toe can't compare to Chess in complexity we thought it was the right project to present in the "temple" of AI at MIT.

Marco wrote the software, we built the hardware, and just a few days before leaving, we met and refined our prodigy. Our robot, named TTT, worked perfectly and aroused much interest.

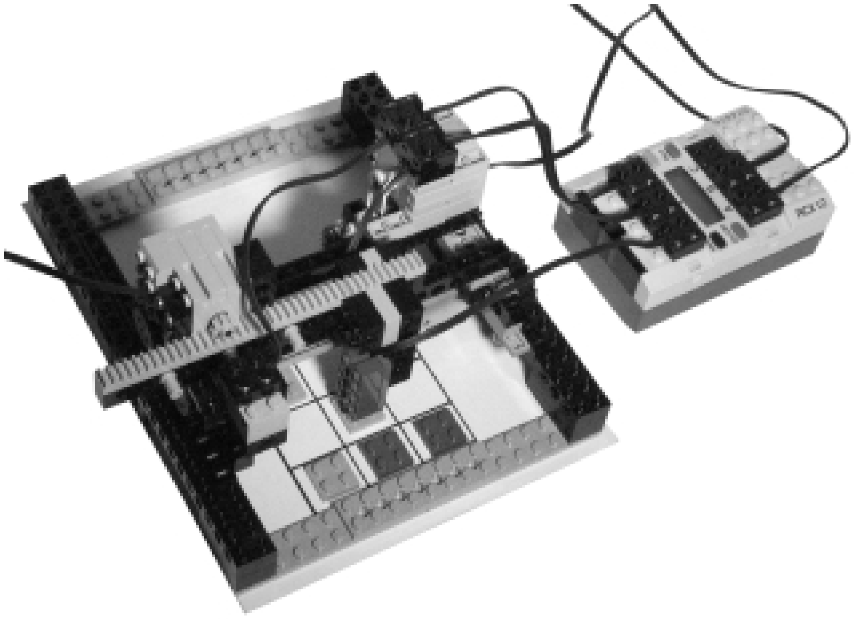
Though we made an effort to keep the requirements minimal, our machine used many extra LEGO parts. For this reason, we decided to build a new, simplified version for use in this book. The description of our original version is still online, however; you can find the link to it in Appendix A.

Building the Hardware

To keep the project replicable using only parts from the MINDSTORMS kit, we gave up the idea of making our robot physically mark its moves with a pen or with pieces. Thus this robot just indicates its move and requires your assistance in recording it on the paper.

If you look at Figure 20.1, you will probably recognize the structure: It's the Maze Solver of Chapter 19, with just some minor modifications. (We won't describe the whole robot again, just the changes we made, referring you to Chapter 19 for the remaining details.)

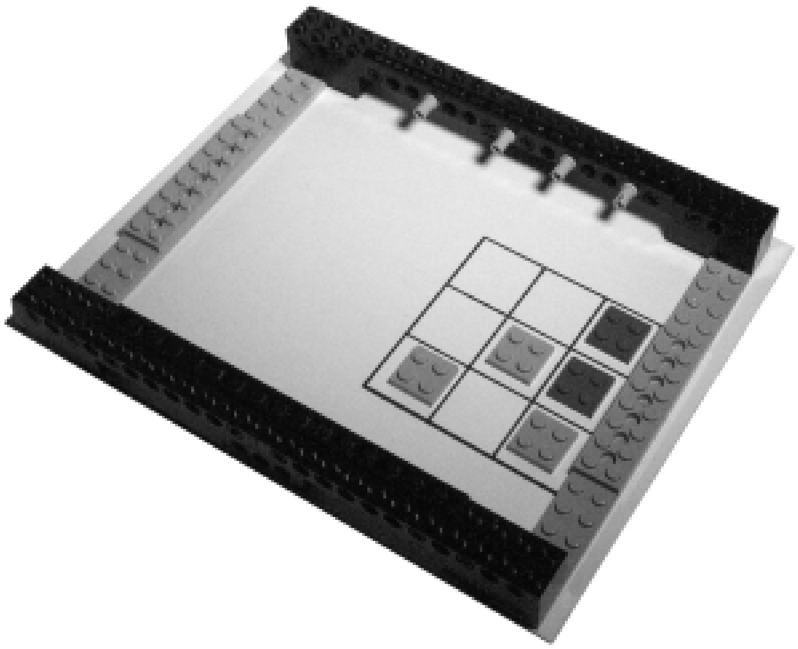
Figure 20.1 The Tic-Tac-Toe Player



The main difference concerns the number of pegs used to mark the position along the X and Y axis. In this project, you need just three stops in each direction, and a fourth for the resting position of the robot. We also removed the baseplate and linked the two longitudinal rails with some plates.

The board is represented by a white sheet of paper, the ideal surface when looking for the highest contrast in light readings. On the sheet, we traced the Tic-Tac-Toe scheme. The centers of its squares align with the stop pegs placed along the rails (Figure 20.2).

Figure 20.2 The Tic-Tac-Toe Board



This robot is designed to read the light value in the center of the squares. You can either mark the moves placing thin LEGO pieces to represent Xs and Os, or drawing large dots with colored markers. In both cases, be sure that the light sensor is very close to the surface, otherwise its readings will be badly affected by the ambient light.

We tested our robot using the 2 x 2 plates included in MINDSTORMS as markers: The gray ones for one team and the blue ones for the other. (You can use green plates with the blue, since they read as almost the same value under the light sensor.)

Writing the Program

Our robot need not know that a game is made of many moves. Every run of the program is a single move, made of the following steps:

1. The robot scans the board, storing a copy of it in a memory array.
2. It evaluates the situation and decides what move to make.
3. It performs the move.

When the user presses the Run button, the robot moves from its resting position and goes to the first row and first column of the board (second peg on the Y axis, first peg on the X axis). It stops there and reads the value of the light sensor. We suggest you use raw values, because, as explained in Chapter 4, they provide finer granularity. In our version, there's a clear-cut division among the readings of the white sheet, the gray plate, and the blue plate, definitely enough to distinguish the three cases with no ambiguity. It's crucial you place the plates directly below the reading positions of the sensor; you can make the placement more exact by marking the squares on the sheet with a thin black line.

Having scanned and stored the first square, the robot will move to the second one, then to the third and so on until the whole board has been scanned. Our TTT software, written in NQC, assigned each square to a variable, but it's possible to use a much more compact representation using individual bits, as demonstrated by Antonio Ianiero's YATTT (Yet Another Tic-Tac-Toe) that employs only two variables for the entire board (see Appendix A).

As for the strategy, when properly played, Tic-Tac-Toe ends in a tied game in which nobody wins. The following list enumerates, in order of priority, the steps to play a perfect defensive game:

1. Check if there's any move that makes you win, a square that completes a row, a column or a diagonal of three.
2. Block the opponent if there's any row, column, or diagonal with two of his pieces and one empty square.
3. In the case where you have a piece in the center and the opponent has two pieces at the ends of a diagonal, choose one of the four central squares of the external rows and columns to force him to block you.
4. If the central square is free, play there.
5. If one corner is free, play there.
6. Play in any free square.

Once the robot has figured out its move, it goes to that particular square and beeps to show it wants to play there, then returns to its resting position.

Practicing with the described strategy is a good idea; take a sheet of paper and play a few games against a friend, or by yourself, following the suggested steps as if you were the robot. This will make you familiar with the possible board situations and the moves required to oppose the attacks.

Don't forget to add a melody to your program that plays when the robot wins; a tune for defeat shouldn't be necessary because it never loses (unless the human player cheats!).

Improving Your Game

The hardware of the robot offers many possibilities for variations and improvements. If you have enough racks, you can cover the side rails with them and use gears in place of wheels, the latter having an unpleasant tendency to slip.

Using a third motor, you can equip your robot with pliers to carry and drop its own pieces, as in our original version. The Mindfest TTT still needed the help of a human assistant to load the piece in a special platform where the robot caught it, but it's possible to devise an automatic dispenser that contains a stack of pieces and drops one on demand.

On the software side, the strategy we described will never let your robot lose, but it's not particularly aggressive either. You can improve it in this area so your robot tries to confuse its opponent whenever possible.

If you have a solid background in programming (and in AI in particular), you can develop a *learning* version of this robot. It would start with no knowledge, playing purely at random in the beginning, but learning from its own mistakes and becoming better and better as the number of played games increases. In our opinion, this is quite a difficult but very impressive and instructive improvement, that makes the robot much more attractive to see in action.

Playing Chess

It's a big step from Tic-Tac-Toe to Chess, but people have succeeded in writing a version for the RCX, proving that the goal is definitely achievable (see Appendix A).

We didn't test the robots in this chapter with the proper Chess-playing software but instead used a reduced set of moves to check the mechanics. We did, however, make an effort to present ideas about interfacing a Chess software with the real world. As always, too, these robots offer tips and suggestions applicable to other areas.

For example, the visual interface employs the Fiber-Optic System (FOS) as an input/output device, using it to emulate a rotation sensor (see Chapter 4) and at the same time give a visual indication of the moves. This demonstrates that the unit, usually considered solely decorative, has greater potential: You can use its eight fibers anytime you want to show the user of your robot a value between 1 and 8.

The robotic arm of Broad Blue, our mechanical Chess interface, is a good example of a system which addresses the points of a plane using an alternative to the Cartesian scheme. In fact, while our Tic-Tac-Toe machine reaches the squares on the board using a combination of two linear movements, Broad Blue uses a combination of two angles. This scheme is very suitable for robotic cranes and grabbing arms designed for various applications, including those aimed at emulating a human arm, whose structure is based on the same principle.

Building a Visual Interface

Comparing Chess to Tic-Tac-Toe, the first thought that comes to mind is that the scan-the-board approach is very complex, if not impossible, to implement. How could a robot tell one piece from another? Even if, with specially coded pieces, you succeeded in the task, the time required for the robot to scan 64 squares would make the game quite boring.

This means you must use a different technique. The most obvious solution is that your robot keeps a copy of the board situation in its memory, updating it with the moves of the players. It knows its own moves, thus all it needs is an input about those of its human opponent. With this approach, the robot no longer needs to scan the board and differentiate the chess pieces, because it uses the copy of the board contained in its memory.

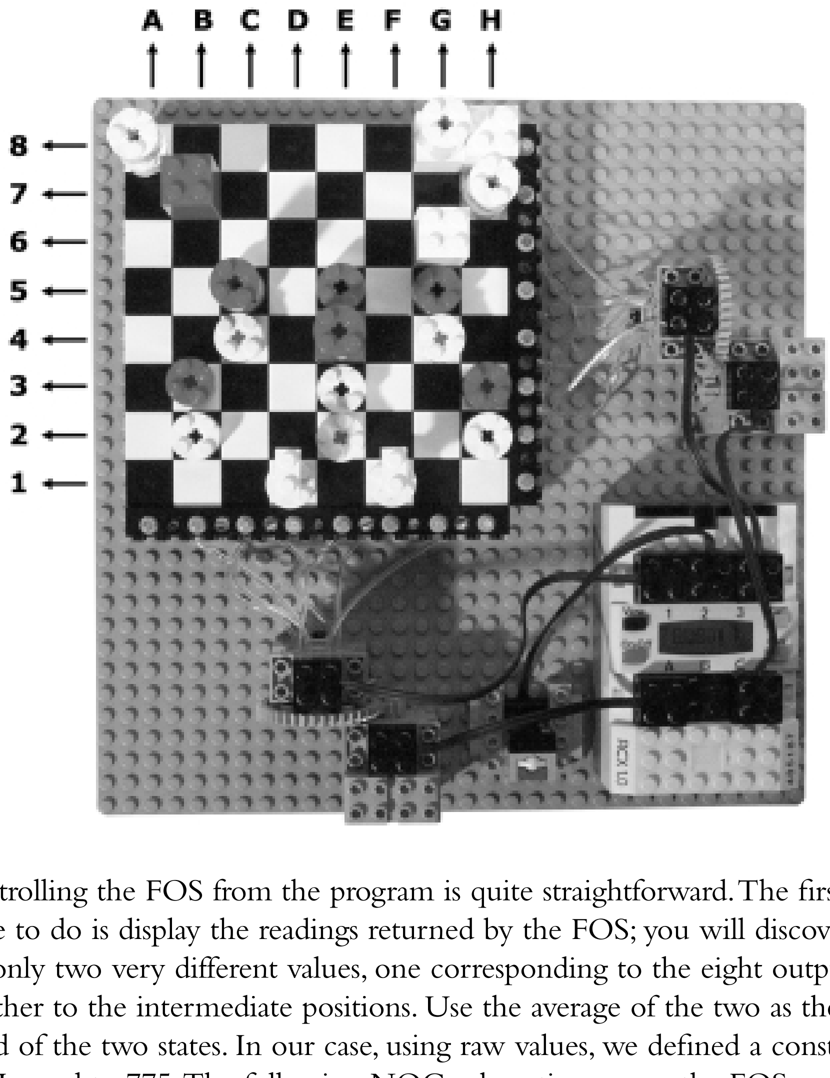
The standard convention to describe board situations in Chess is based on a simple coordinate system where rows are numbered 1 to 8 and columns A to H. Thus all you need is a way to choose a starting column-row pair, that points to the piece the human player wants to move, and a second column-row pair that outlines its destination.

Our first Chess interface works exactly like that. We used the Fiber-Optic System (FOS) to display the selected coordinates beside a small all-LEGO board (Figure 20.3).

An FOS unit contains a rotor which has a tiny red light. Wiring the unit to a power source, like the output port of an RCX, turns the light on. The casing of the FOS device has an axle-hole in its center, through which you can rotate the inner rotor. This rotation aligns the light with one of eight possible holes, from which you can drive the beam of light into the LEGO optic fibers.

In our setup, the FOS units connect to two input ports of the RCX, which are configured as light sensors. The electrical current the units receive from the RCX, aimed at power activated sensors, is enough to make their LED turn on. At the same time, the fact that they are attached to input ports allow you to read their state. In fact, the FOS unit returns two different values, the first when the rotating light is aligned with one of the eight holes, and the second when the light is hidden between two holes. This is the property that makes them good candidates to emulate rotation sensors: You can implement an internal counter which is incremented every time you detect a transition from one value to the other (see Chapter 4). The FOS can't tell you the direction, but if you couple it with a motor, like we did, you know in which direction you are making it turn. In our robot, each FOS unit is powered by its independent motor through a 1:5 reduction stage.

Figure 20.3 The Chess Visual Interface



Controlling the FOS from the program is quite straightforward. The first thing you have to do is display the readings returned by the FOS; you will discover they assume only two very different values, one corresponding to the eight output holes and another to the intermediate positions. Use the average of the two as the threshold of the two states. In our case, using raw values, we defined a constant THRSH equal to 775. The following NQC subroutine moves the FOS one step:

```
#define THRSH 775
int pos;

void FOS_step ()
{
  OnFwd(OUT_A);
  while(SENSOR_1 < THRSH);
  while(SENSOR_1 > THRSH);
  Off(OUT_A);
  pos++;
  if (pos > 8)
    pos=1;
}
```

Every time you call this subroutine, the light will move one step, and the inner variable `pos` will keep track of its position (provided that it started from a known point).

To input the user's moves to the RCX, we used an interface as simple as a single touch sensor. This is how the "dialog" between the players unfolds:

1. The user clicks to choose the column of the piece to move. For every click, the FOS increments one position. When okay, the user double-clicks.
2. Now the touch sensor controls the row position, and in a similar way the user selects the row of the piece to move.
3. Then he inputs the destination column and the destination row of the chosen piece.
4. The program records the required move and checks if it's valid, confirming the result with an appropriate sound. If the move is rejected, the procedure starts again from Step 1.
5. The RCX evaluates its own move, and shows the coordinates of the starting square using the FOS lights. It then waits for a confirmation click from the user.
6. Now it shows the destination square, and waits for another confirmation click.

You can speed up the input process making the program show only the valid rows and columns where the user has pieces for Steps 1 and 2, and the valid destinations for the chosen piece during Steps 3 and 4.

It's possible to improve the communication protocol, too; for example, allowing decrements of position when the sensor gets pressed for a "long" time instead of receiving a fast click.

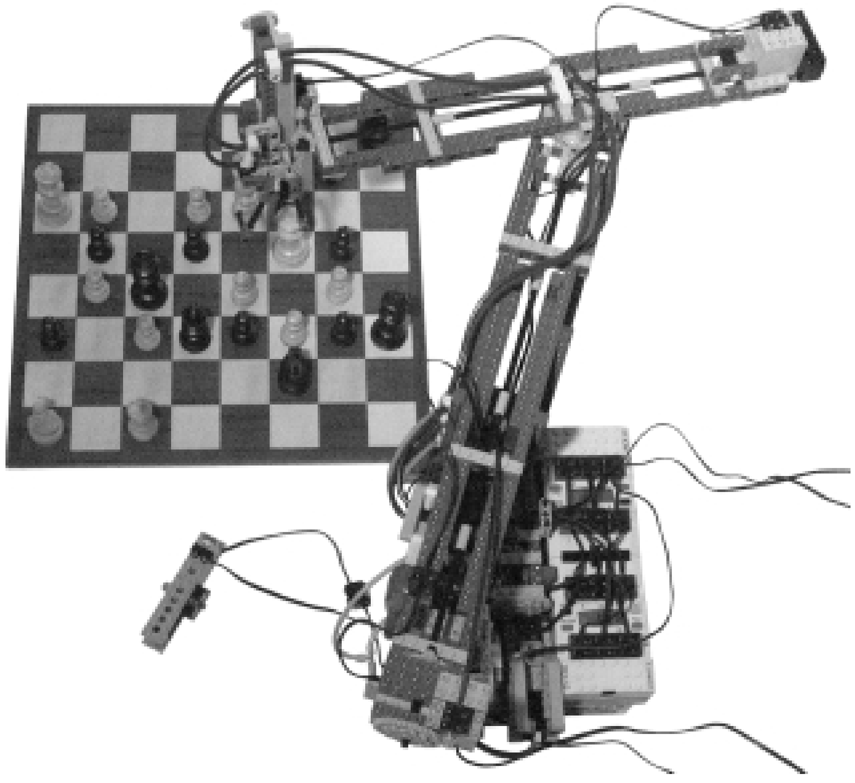
Building a Mechanical Interface

Suppose you're not satisfied by the previous simple visual interface, and want your Chess robot to really move its pieces. What difficulties can you expect?

Your system needs at least three degrees of freedom, two to get to all the squares of the board, and a third for the vertical movement necessary to lift and put down the pieces. Moreover, it needs some grabbing ability to handle the pieces.

The size of the board and the shape of the pieces greatly affect the design of your machine. We challenged ourselves by building a robot able to play on a regular wooden Chess set, and this led to an extra-large robot (Figure 20.4).

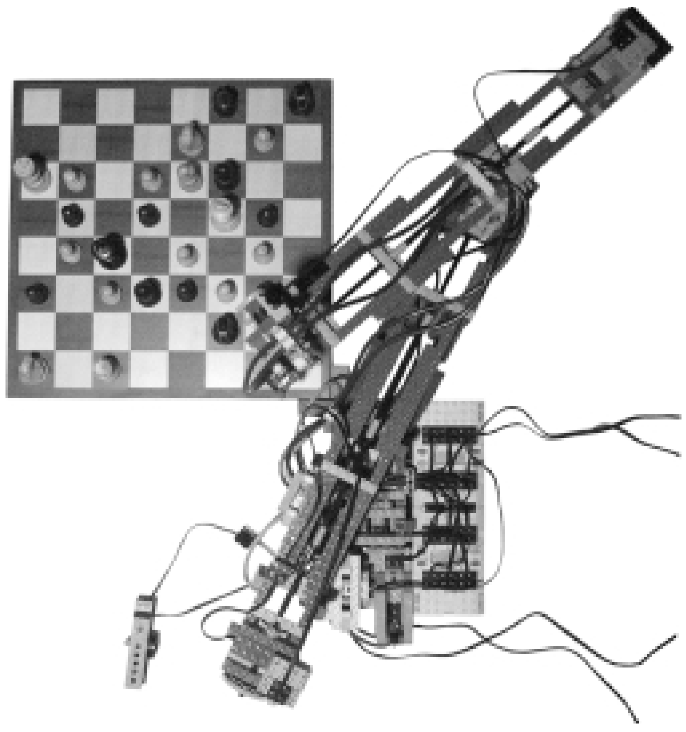
Figure 20.4 The Chess Mechanical Interface



The first thing you notice is that this time we didn't use a Cartesian system to address the square. Our robotic arm emulates the kind of movement a human arm uses, with two articulations that correspond to the shoulder and the elbow. Even the size is not too far from that of a human arm!

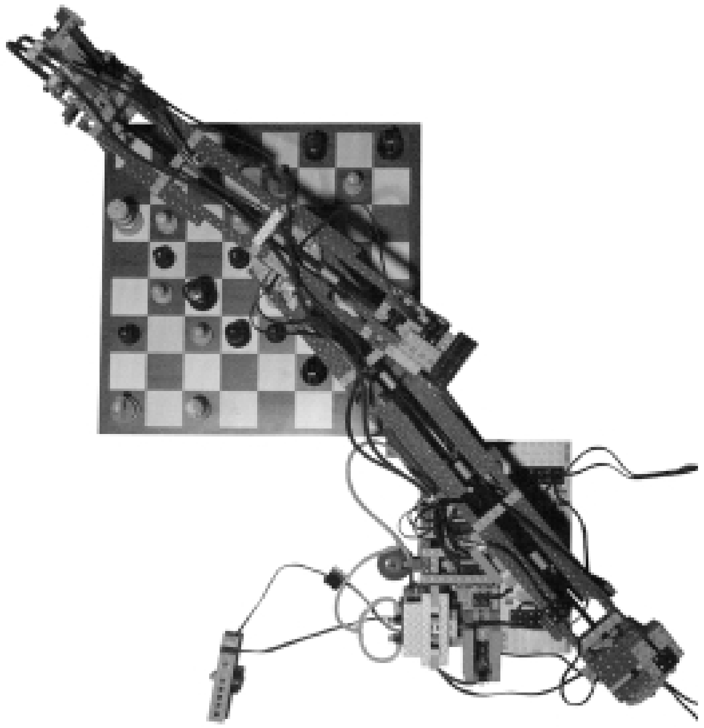
As we made wide use of blue parts, we decided to name this machine Broad Blue. (It's a play on words, considering "Deep Blue" was the name of the famous IBM computer that was the first defeat Chess world champion Garry Kasparov!) This system is able to address any point on the Chessboard, as shown in Figures 20.5 and 20.6.

Figure 20.5 Broad Blue Plays in H1



The pliers at the end of the arm must be capable of a long vertical range, since the pieces must be lifted high enough so they don't touch the other pieces on the board while the arm moves. Their jaws have to open enough to capture the pieces, but not so much that they involuntarily shift the adjacent ones. At the same time, the pliers must be able to grab all the different pieces, from the Pawn to the King.

Figure 20.6 Broad Blue Plays in A8



We used a rack and pinion assembly for the lifting mechanism, and a small pneumatic cylinder for the pliers (Figure 20.7).

On the other side of the lifting mechanism you see a touch sensor that gets closed by two cams at the extreme positions of the rack (Figure 20.8).

The motor that operates the lifting system lies at the opposite end of the secondary arm, behind the turntable (Figure 20.9), and uses a worm gear-to-16t reduction stage.

The advantage of this configuration is that the weight of the motor acts as a counterweight to the mass of the lifting mechanism. To this same purpose, we also added a weighed brick side to the motor. It's very important that the secondary arm is well-balanced, otherwise it will introduce a strong torsion (twisting) on the primary one, and torsion is much more difficult than weight to oppose and compensate for. In the same picture, note also that the secondary turntable gets turned by an 8t gear. This is connected through a pair of bevel gears to the long joined axle that reaches the motor at the opposite side of the primary arm.

Figure 20.7 The Lifting Mechanism

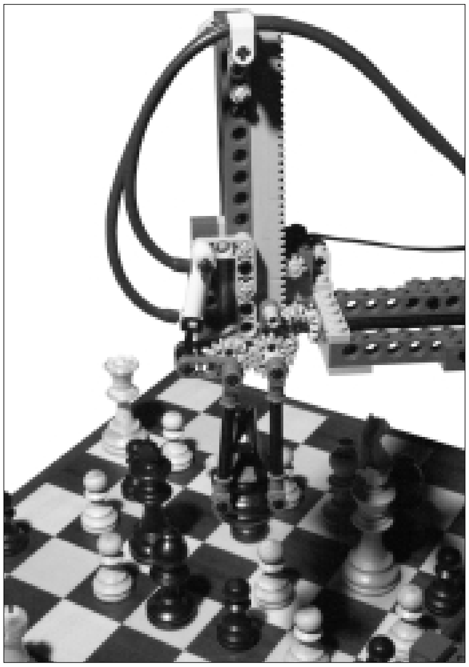


Figure 20.8 Rear View of the Lifting Mechanism

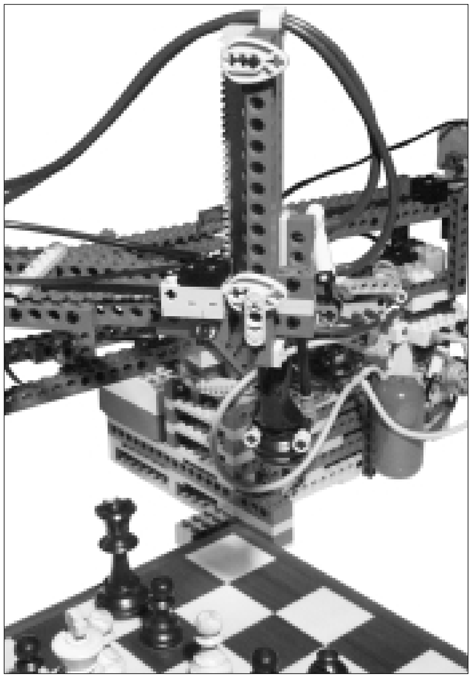
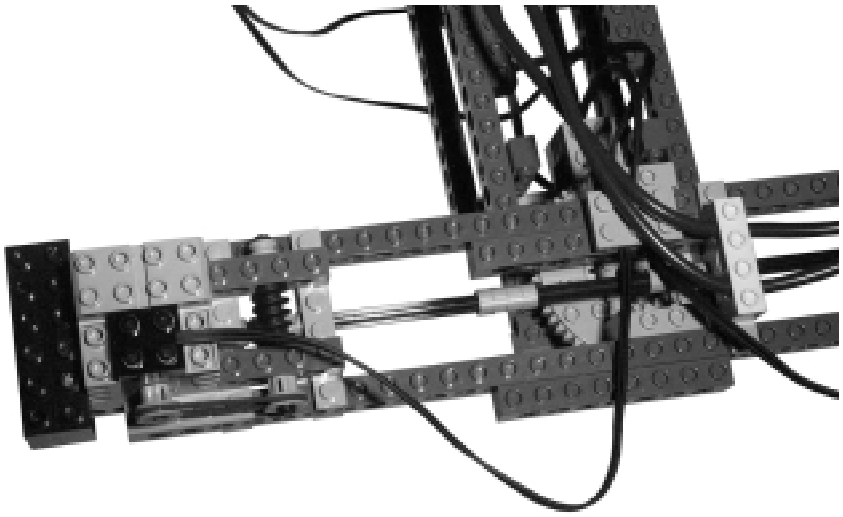
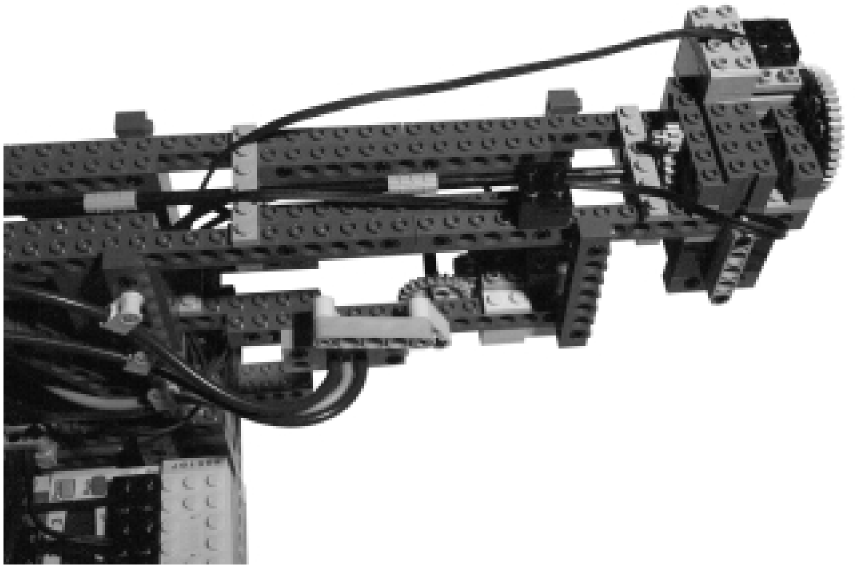


Figure 20.9 The Lifting Motor at the End of the Secondary Arm



The primary arm has a more solid structure, designed to bear its own weight plus the whole secondary arm. At its opposite end you find the motor that rotates the secondary arm, connected also to the rotation sensor that monitors that movement (Figure 20.10). The reduction geartrain is made by an 8:24 stage and a 8:40 one, for a total ratio of 1:15.

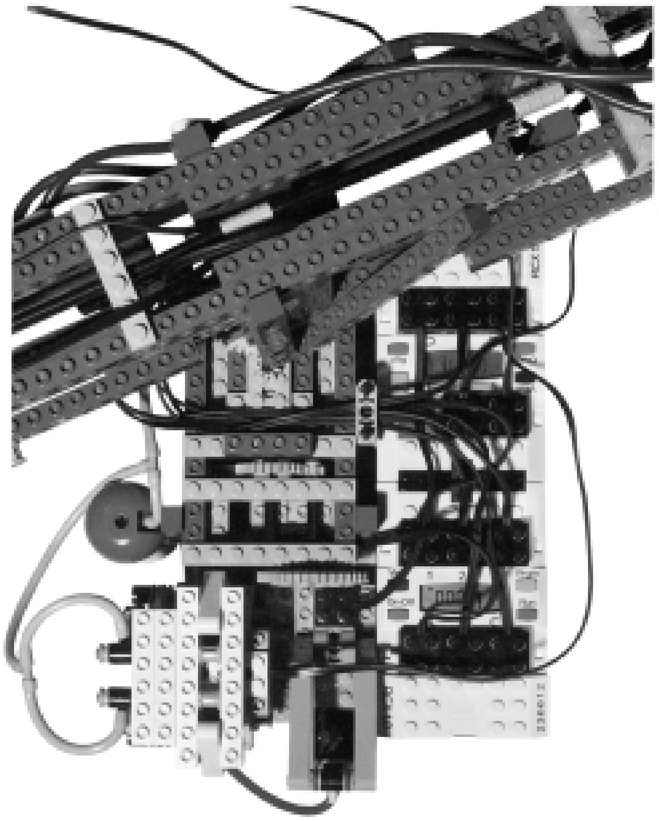
Figure 20.10 The Motors at the End of the Primary Arm



On the lower deck of the arm is the motor that operates the valve switch for the pliers. As for the secondary arm, both the motors and a couple of weight bricks serve to keep the primary arm well-balanced. If you don't do this and consequently induce a strong asymmetric load on the turntable, it may fall apart.

The base of the robot has to be very solid. It contains two RCXs, the motor that moves the primary arm and its rotation sensor, a compressor for the pliers and a pressure switch. All this weight helps in making Broad Blue very stable (Figure 20.11).

Figure 20.11 The Base



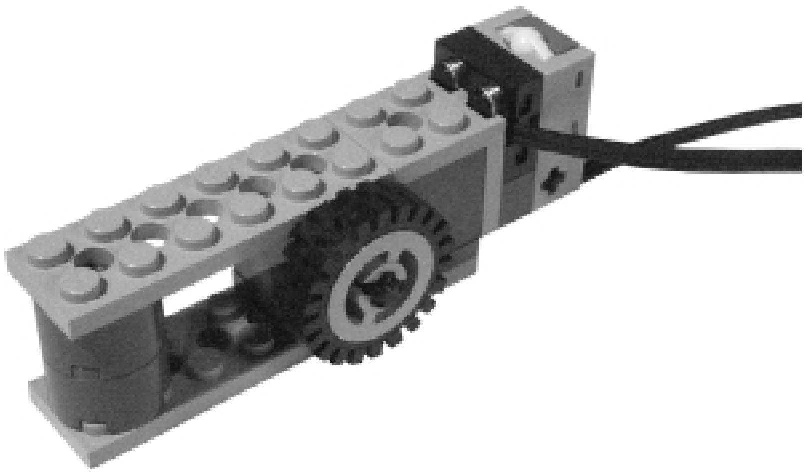
The geartrain of the primary turntable has the same configuration as the secondary one.

To provide the necessary supply of air, we used the double-acting compressor (because we had it ready), but the robot really needs a small quantity of air and any compressor should work.

It's important that you make all the wires emerge from a point close to the center of the primary turntable so they don't affect its rotation. In large robots like this, clean and well-organized wiring helps a lot in keeping things under control.

For the input of the human moves, we replaced the single touch sensor with a slightly more sophisticated unit also made from a rotation sensor. This speeds up the choice of the coordinates, changed according to rotations, displayed on the master RCX, and confirmed by a click on the touch sensor (Figure 20.12).

Figure 20.12 The Input Unit



Connecting and Programming Broad Blue

This is the only project in the book that requires two RCXs. As in most dual-RCX projects, it uses a master-slave configuration: The master decides what to do while the slave merely performs the required actions. Table 20.1 summarizes the wirings and resource allocations of Broad Blue; RCX1 is the master, and RCX2 the slave.

Table 20.1 Broad Blue Resource Allocations

Resource	Function
RCX1 IN 1	Rotation sensor of the input unit; sets user's move.
RCX1 IN 2	Touch sensor of the input unit; confirms user's move.
RCX1 OUT B	Motor that operates the pneumatic valve switch for the pliers.

Continued

Table 20.1 Continued

Resource	Function
RCX1 OUT C	Always ON, only used to supply power to compressor (via the pressure switch).
RCX1 Display	Shows the input square in the form Column.Row.
RCX1 Loudspeaker	Confirms or invalidates user's moves with an appropriate sound.
RCX2 IN 1	Rotation sensor of the primary arm; controls its position.
RCX2 IN 2	Rotation sensor of the secondary arm.
RCX2 IN 3	Touch sensor of the lifting system, closes at all up or down positions.
RCX2 OUT A	Motor of the primary arm.
RCX2 OUT B	Motor of the secondary arm.
RCX2 OUT C	Motor of the lifting system.

RCX1 receives the user's move with a protocol similar to that described for the Chess Visual Interface: starting square column, click, starting square row, click, destination square column, click, destination square row, click. At this point, it evaluates its own move, then performs it with the assistance of the slave RCX.

The communication protocol between the two programmable units is rather simple: You need 64 values corresponding to the squares, plus two more for the pliers-up and pliers-down commands. Include a "done" return value that the slave RCX uses to inform the master when it has finished the required action.

NOTE

If you want to compute the address of each square in terms of rotation sensor positions, you need some trigonometry. The nice thing is that in this case you can compute them just once on your PC, then store them in an 8 by 8 array in the program, so the RCX doesn't have to actually perform any calculation, just read the coordinates from a table.

Each turntable is like a 56t gear, and the 8t pinion acting on them results in a 1:7 ratio. The rotation sensor is placed after the first 1:3 stage, thus each complete turn of the arm corresponds to 21 turns of the sensor. The latter features 16 ticks per turn, meaning that each turn increments a count of 336 units, corresponding to 1.07 degrees per tick.

The actual size of your chessboard, and consequently of the arms of your robot, is very critical. The longer the arms, the higher the angular resolution needed to control their movements. The arms of Broad Blue are both 22cm long from the center of their supporting turntable; this means that the end of the secondary arm covers a circle of about 138cm (22x2xπ). Each degree corresponds to a sector with a circumference of 0.38cm. When the arms are aligned and aiming at the farthest squares of the board, each degree of rotation of the primary turntable acts on a circle 44cm in radius and corresponds to movement of 0.77cm at the pliers. We said that our resolution is 1.07 degrees per tick, and this results in 0.82cm in that area of the board. This is a bit critical: You might prefer to connect the rotation sensor before the first 1:5 reduction stage to increase precision by a factor of five.

The slack between gears drastically decreases the accuracy of the movements. To reduce the problem to a minimum, Broad Blue always approaches the squares from the same side of the board, going a bit farther and coming back when necessary.

Variations on the Construction

Nothing prevents you from building a Chess robot with a large XY system like the one we described for Tic-Tac-Toe. This way you can avoid the rotation sensors entirely and use the good old touch sensors and pegs technique to address all the squares.

A smaller board will result in a smaller robot, meaning less parts, less weight to support, and consequently a simpler structure. Similarly, smaller pieces will need a shorter range for the lifting mechanism.

Is it possible to build a mechanical interface similar to ours, but use only *one* RCX? We believe so. If you are able to use pneumatics for the lifting system outside that used for the pliers, you can coordinate their movement in a predefined sequence (starts with pliers up and opened):

1. Go down.
2. Close pliers (grabs the piece).
3. Go up (now the robot moves to the destination square).
4. Go down.
5. Open pliers (drops the piece).
6. Go up.

What you need is a control system that operates the valve switches in the proper sequence, a rotational or linear mechanism activated by a single motor. Can you devise one?

Applying a completely different technique, you can copy the system used in some commercial chessboards that employ an electromagnet to move magnetic pieces from the bottom. Did you ever see them in action? They drag the pieces, sliding them along the borderlines of the square, so they don't interfere with the other pieces on the board. The advantages of this technique include the fact that you don't need any lifting/grabbing system anymore, thus a single RCX can do all the work; in the way of disadvantages consider that you need many non-LEGO parts, including magnetic chess pieces, a special board, and an electromagnetic coil. LEGO doesn't produce electric coils for the RCX, but they aren't difficult to make. Be aware that they absorb much more current than motors.

Now that we think of it, you can use the sliding technique even from the top. Imagine a robot similar to ours, that instead of grabbing and lifting the piece, simply surrounds it with a sort of cage lowered from above and then pushes it along the proper path. This will save one output port, but will make the software a bit more complex because the arms cannot go straight to the destination square but rather must follow an optimal path to avoid the other pieces.

Playing Other Board Games

The techniques we described in this chapter apply to many other board games, each one having its own peculiarities and requiring some adaptations.

Checkers shares with Chess similar difficulties about moves, with the difference being that you must find a way to crown the pieces when required. Moving crowned pieces without letting them come apart might be not so easy. On the other hand, Checkers needs much simpler software than Chess.

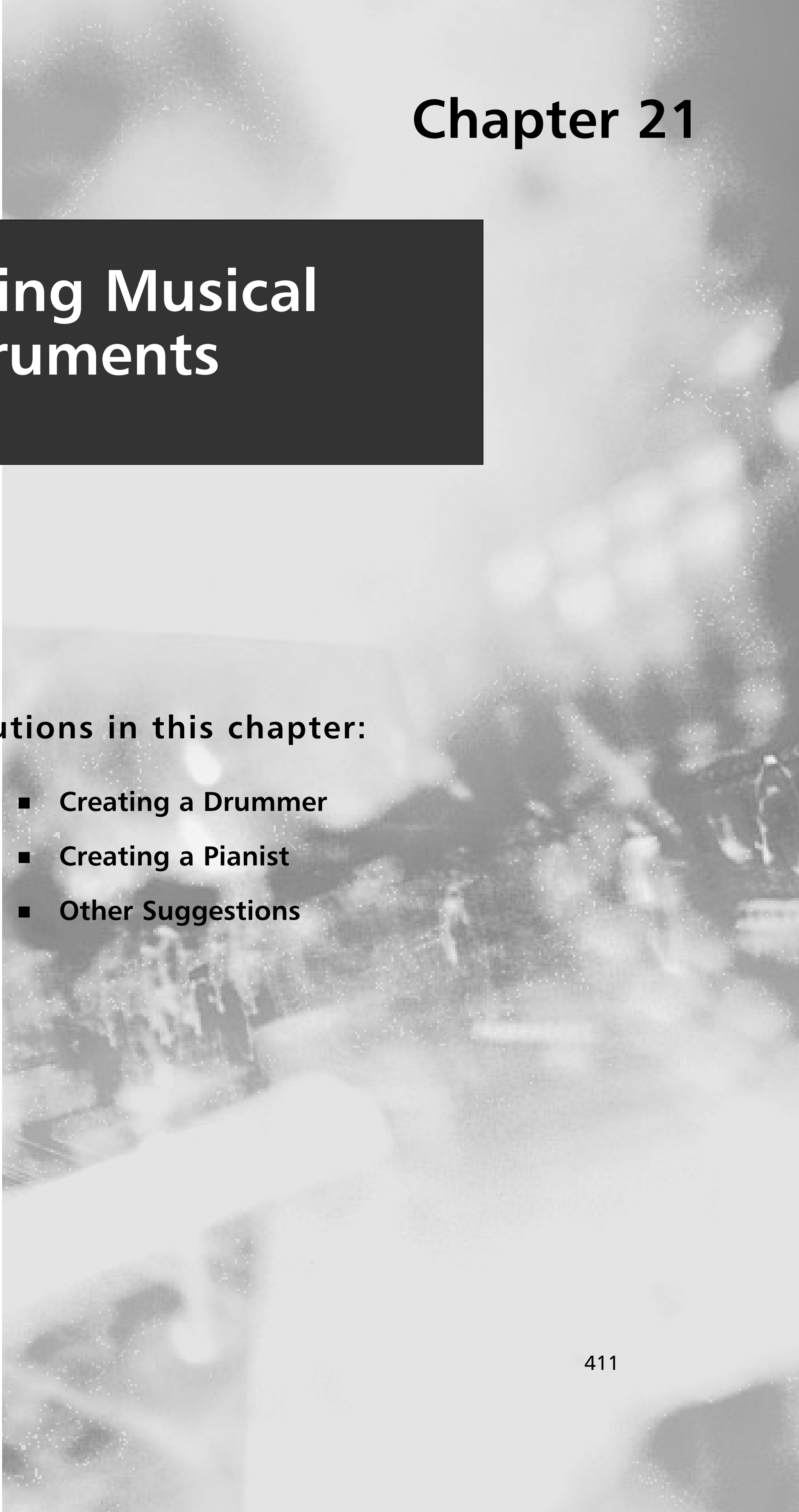
Go, in its slightly diverse variants, is the world's most popular board game. Even though it's not played much in the West, it's widespread throughout all Asian countries. Because the pieces, called *stones*, get dropped on the board but never moved, it seems ideal for a robot equipped with an automatic dispenser. The fact that the stones are black and white also suggests the possibility of using a scanning technique similar to the one we incorporated into the Tic-Tac-Toe robot, though it would likely prove impractical on the official 19 by 19 board and even on the 13 by 13 reduced one. We recommend you limit your robot to the 5 by 5 training board. It will require more than enough effort in the way of programming. Go, in fact, is considered the most difficult game for computers to learn how to play well—a sort of frontier for Artificial Intelligence. While the strongest Chess program defeated the human world champion, the strongest Go program cannot even beat a good amateur. A tough challenge for your RCX!

Summary

Whether or not you have an interest in board games, this chapter describes some interesting tricks that may prove useful in other situations, too. You have seen that you can input data into your system using the combined knowledge of a position and a value read from the light sensor, as we showed in the Tic-Tac-Toe robot.

The simple Chess Visual Interface taught you that even the FOS units, usually designed for decorative purposes only, may work as output devices. It also showed you that you can build a complete input interface around a single touch sensor.

Broad Blue is the most complex robot of the book. It employs some of the concepts illustrated in Chapters 10 and 11 about pneumatics, object grabbing, degrees of freedom, and others. More importantly, it demonstrates that the bounds of what can be done with the MINDSTORMS system are very far reaching.



Chapter 21

Playing Musical Instruments

Solutions in this chapter:

- Creating a Drummer
- Creating a Pianist
- Other Suggestions

Introduction

Chapter 6 describes the sound system of the RCX and the way you can program it to produce music. Here we are going to explore a more indirect way of performing music, one where your robot actually plays an instrument.

The main difference between the two creatures described in this chapter lies in their instruments. The first, a drummer, plays a custom LEGO-made drum-set specifically designed for the task, while the second, a pianist, performs on a real piano. This diversity, as you'll see, reflects heavily on their architecture; the chess machine of Chapter 20 taught you that interfacing LEGO with real-world objects, no matter how common they are, is usually difficult and requires a great deal of effort (and many parts!).

Creating a Drummer

The part you need to base your drummer design on, not surprisingly, is the drum set. From what we know, LEGO doesn't make any part that acts as a drum, so you need to be imaginative and come up with an alternative. When we attempted this, small cans first came to mind as an option, but then we felt a more LEGO-like solution might be appropriate, so we started rummaging through our drawers searching for a part that might provide the right inspiration.

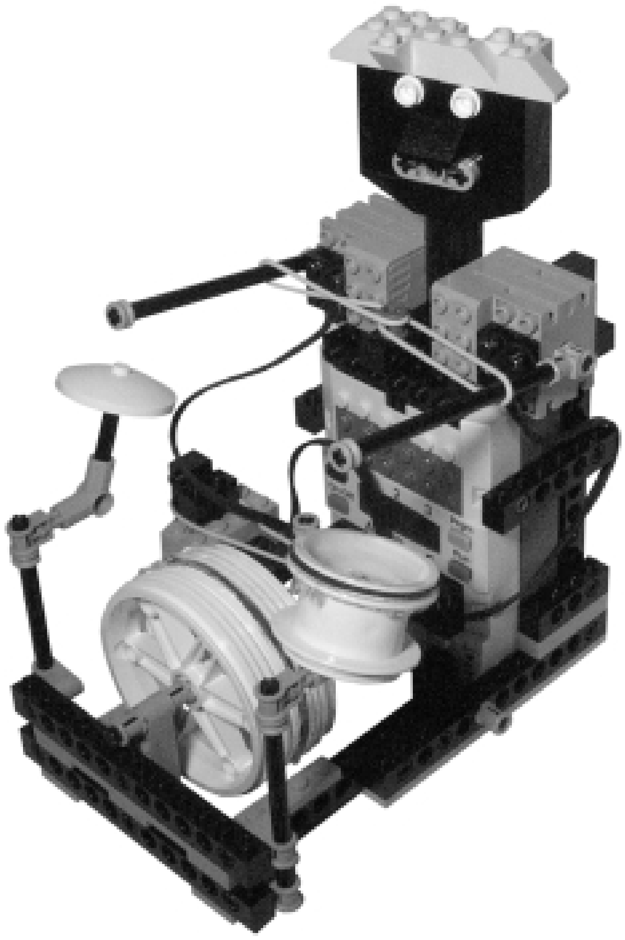
That's when we came upon the wheel hubs. They seemed perfectly suited for our goal, their shape closely resembling that of a real drum. The only missing part was the "skin," the diaphragm that covers drums and that produces the sound when hit. A visit to the kitchen solved this problem, too: ordinary plastic wrap provided us with an answer. We stretched two tight layers of it on one side of the hub, secured it with a rubber band, trimmed the excess wrap, and our drum was ready to use.

The second problem we faced regarded the sticks and, more importantly, the percussion mechanism. Playing a drum is more a matter of speed than strength: the stick must hit the drum very quickly and promptly withdraw, helped in this by the bounce. We tinkered with some different mechanisms in order to emulate a human drummer's swinging movement—with little success. Have you ever banged your head against the wall searching for a complex solution, only to find an answer that's not only incredibly simple? Better late than never, we discovered that an axle perpendicularly attached to the motor does the trick. Keeping the motor powered for a very short time and then switching it into float mode causes it to pass on enough speed to the "stick" that it bounces back with no resistance.

Building the Drummer

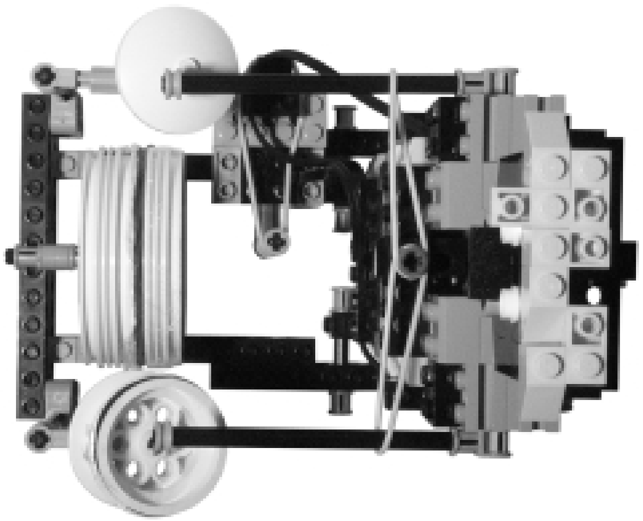
With the basic design and material problems solved, the rest of the robot came together quite easily (Figure 21.1). It is, indeed, one of the simplest robots in the book, made only from MINDSTORMS parts with a third (optional) motor added in.

Figure 21.1 The Drummer



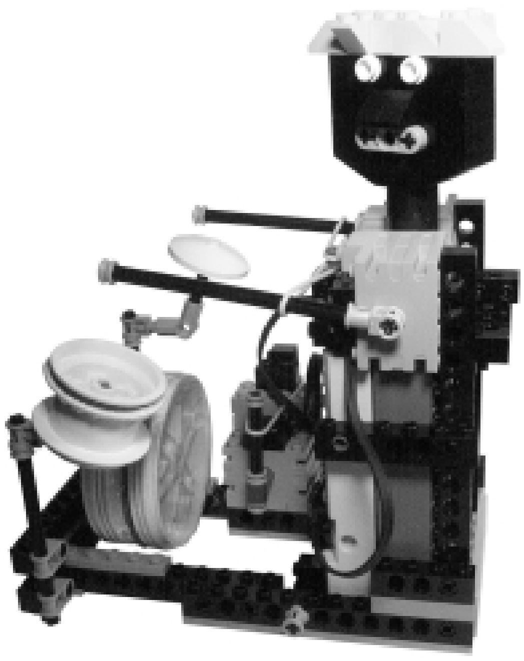
The drum set contains a bass drum, a tom-tom, and a cymbal, though the latter sounds rather dull, since it's a piece of plastic rather than a true cymbal! (Figure 21.2).

Figure 21.2 Drummer Top View



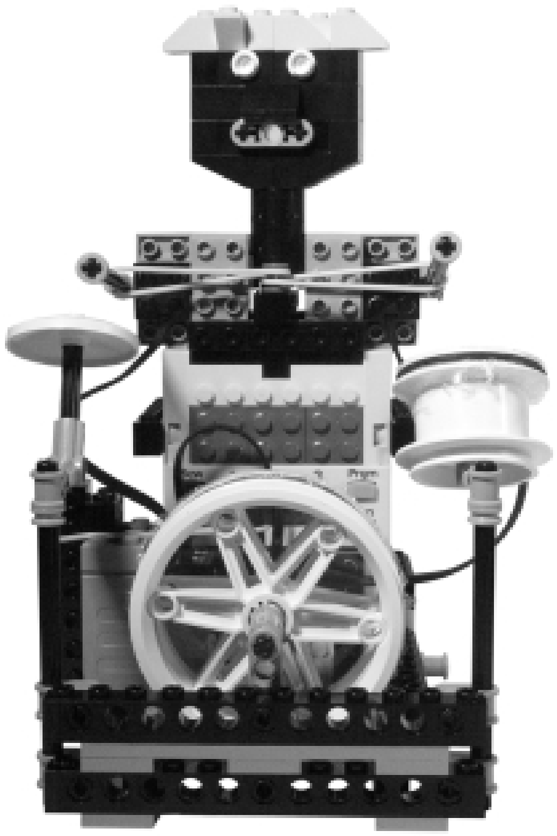
Both the sticks and the pedal feature a slack belt that helps them return to their neutral position (Figure 21.3).

Figure 21.3 Drummer Left Side View



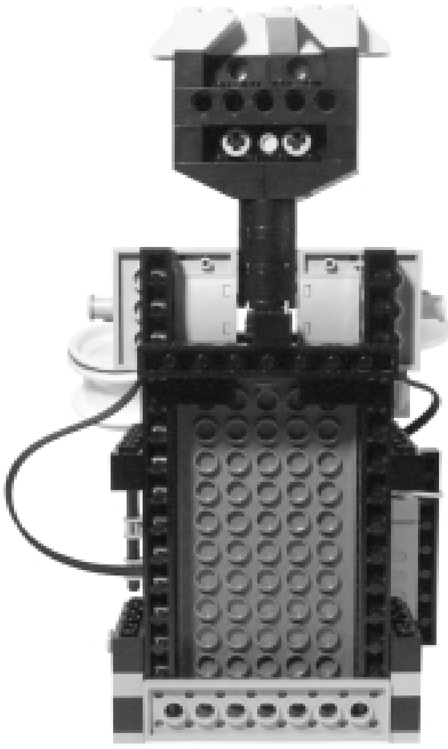
The entire drum set is attached to a base beam, which is connected to the feet of the drummer so as to form a single, solid assembly (Figure 21.4).

Figure 21.4 Drummer Front View



Two 16 long beams run vertically across the back of the robot, supporting the motors and the RCX (Figure 21.5).

Figure 21.5 Drummer Rear View



Programming the Drummer

This is one of those cases where the simplicity of the structure corresponds to the simplicity of the software. As we explained earlier, all you have to do to use the sticks is to run the motor for a very short time before switching it into float mode. In our test version, we kept the motor on for just two hundredths of a second.

Create three routines (or macros) that correspond to the left and right sticks as well as the pedal. Each one will have a structure similar to the following NQC example:

```
void right_stick()
{
  On(OUT_C);
  Wait(2);
  Float(OUT_C);
}
```

We recommend you also write some subroutines for the combined action of two motors (or all three), so that when you want multiple strokes made, they occur together, in synch. When you're done with this, you are ready to code your first drum pattern, which is composed of a series of strokes and pauses.

Variations

You have probably noticed that this robot has no sensors. It doesn't need them to play, but they could add interesting functionalities. Used as input devices, the touch sensors can be programmed to increase or decrease the tempo of the execution, or to trigger a change of pattern or a fill-in.

The addition of a light sensor offers you the opportunity to conduct the performance with a flashlight. For example, the robot could read the time between two passages of the light beam on the sensor, and use that interval as a value for the duration of a beat.

Creating a Pianist

As announced in the introduction, the Pianist has been designed with the ambitious goal of playing on a real piano. It follows the scheme we implemented in the S16 robot that we published on our site. This time we made it even more difficult, aiming at an acoustic piano rather than at an electronic keyboard, requiring a stronger and faster striking of the key to produce the note.

Building the Pianist

This robot requires a lot of extra parts, mainly beams and plates used to make the structure solid enough to withstand the forces involved in the performance. What seems a gentle touch to a human hand, is a strong effort for the small LEGO motors. Consider also that you cannot gear them down too much, because an acoustic piano needs a minimum speed on its keys for the hammers to beat the strings.

Figure 21.6 shows the Pianist in action. You see that its feet go under the keyboard, a simple but effective way to oppose the resistance of the keys and keep the fingers in their place.

It's amazing how well the distance between the keys matches three LEGO units (Figure 21.7). This makes our pianist have its six fingers precisely positioned in the center of each key. As the robot has no mobility (keeping it from reaching the rest of the keyboard), it's only able to play six adjacent white keys.

Figure 21.6 The Pianist

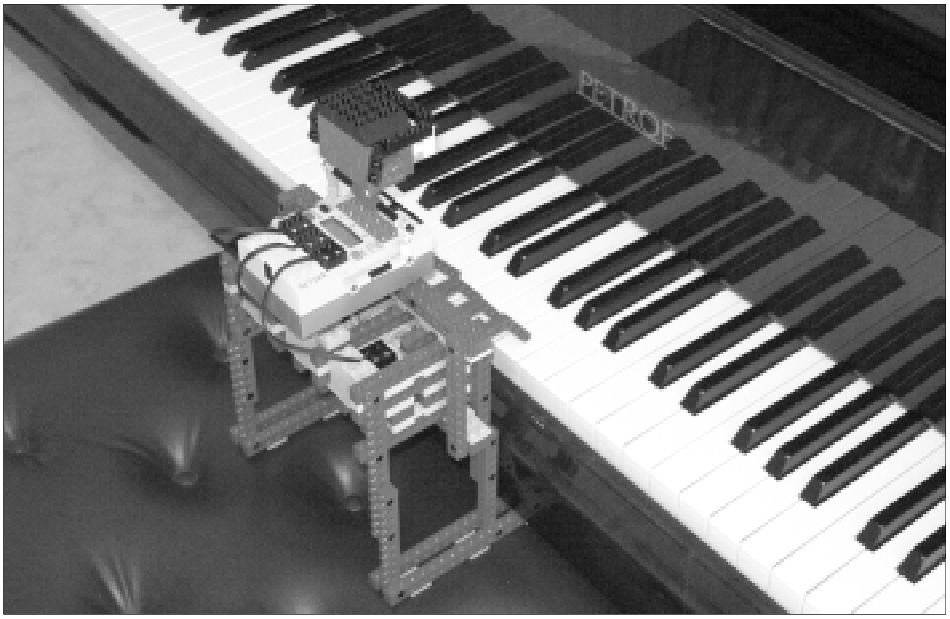
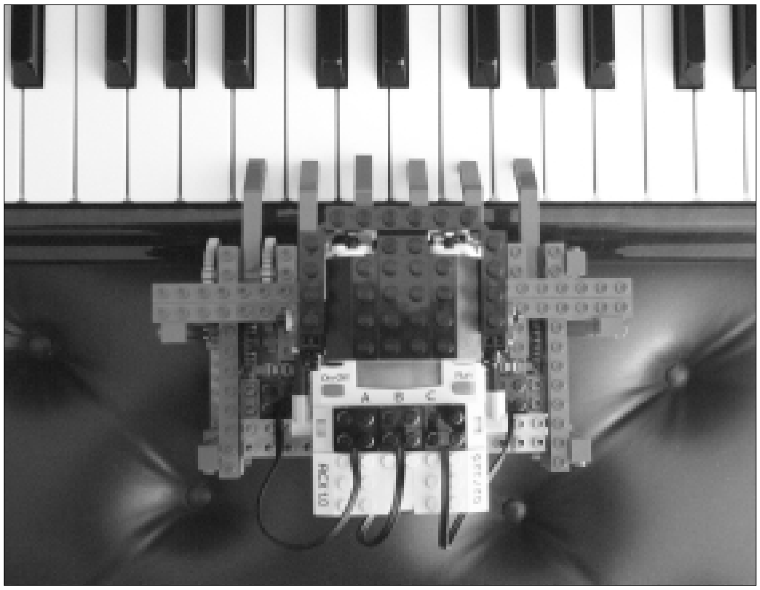
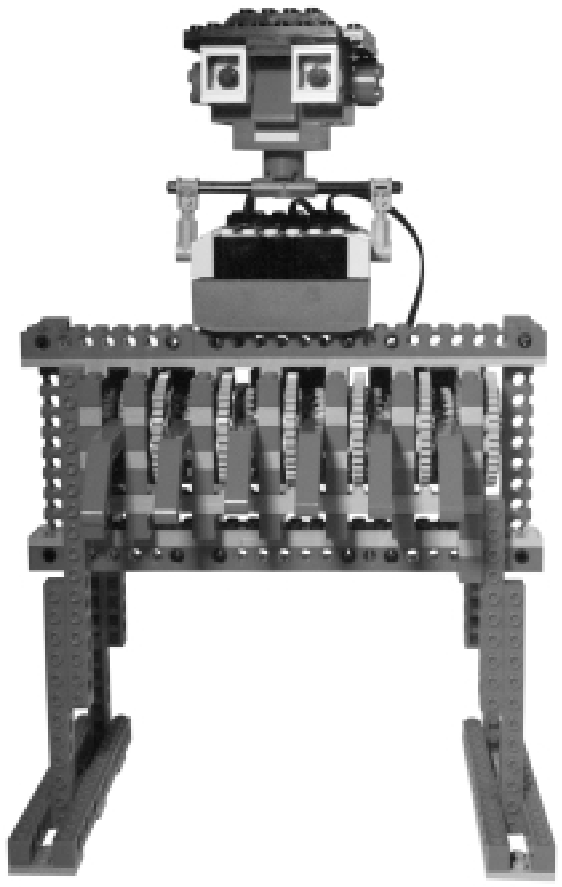


Figure 21.7 Top View of the Pianist at the Keyboard



As we said, though conceptually simple, this robot requires a lot of strengthening. We braced the entire system that operates the fingers between two strong compound beams protruding off the sides of the robot (Figure 21.8).

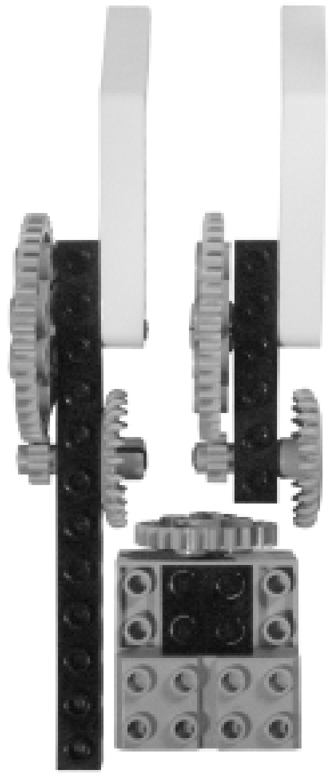
Figure 21.8 Pianist Front View



The head of the robot, with its funky glasses, as you may have guessed, is purely decorative.

The robot uses three motors, each one controlling two fingers. When one finger goes down, its mate goes up. Figure 21.9 shows the mechanism of a pair of fingers with the components slightly taken apart so you can better understand how it works.

Figure 21.9 Pianist Finger Mechanism



Our pianist contains three finger groups like that one, placed side by side. To show them more clearly, we removed the RCX and the strong traversal beam that keeps the fingers together (Figure 21.10). There's no need for rubber bands or other systems to center the fingers since the keys themselves provide the push that lifts them back up.

The sides of the robot are easily adjustable in height, so you can adapt the structure to different keyboards (Figure 21.11).

At the bottom of the structure is a row of parallel longitudinal beams supporting the motors (Figure 21.12).

Figure 21.10 Pianist Top View (with RCX and Beams Removed)

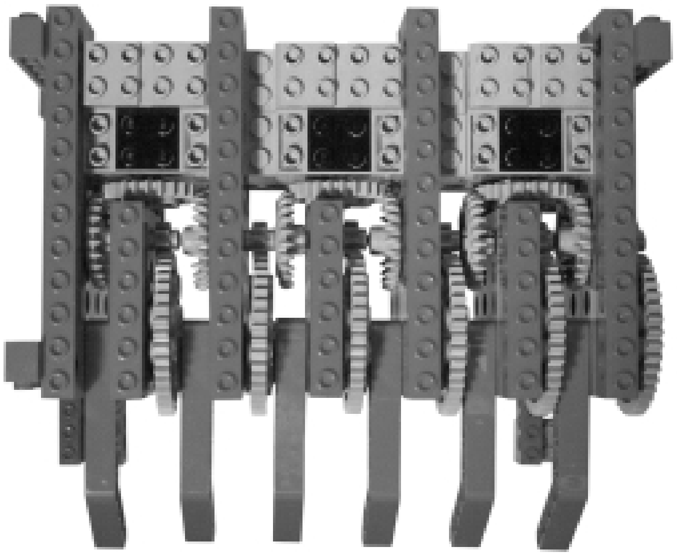


Figure 21.11 Pianist Side View

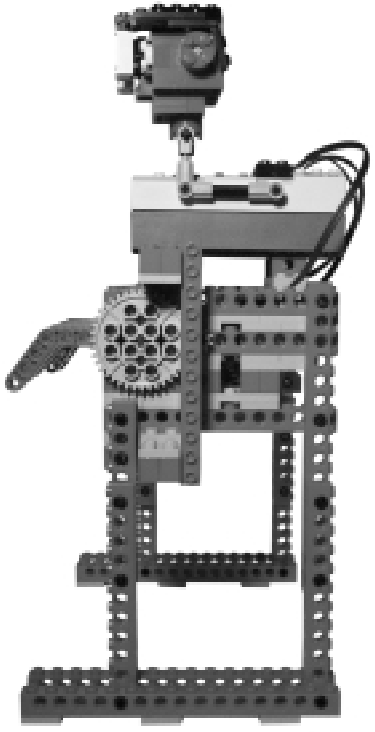
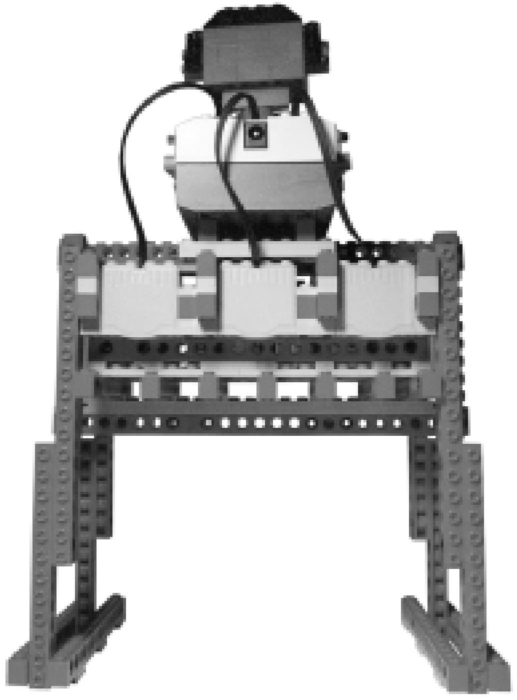


Figure 21.12 Pianist Rear-Bottom View



Programming the Pianist

The technique required to program this robot is indeed very similar to the Drummer. In this case, keep the motors on a bit longer: Our version works with a timing of eight hundredths of a second.

Write a short routine for the activation of each finger, naming them after the corresponding notes. In Figure 21.7, you see the Pianist positioned on six notes that go from B to G, while our code, in fact, contains six routines B(), C(), D()... G().

We wrote the code to play chords, too. The Pianist actually supports a limited polyphony and can perform two or three note chords, provided they don't include two notes controlled by the same motor. Staying again with our example, C+E is a valid combination, while D+E is not.

Similar to what you did for the Drummer, construct your melody using a sequence of calls to the note routines interleaved with pauses. Build your delay values starting from the shortest note your melody contains. Let's say you're going to use eighth, quarter, and half notes, the eighth will be the base of your construction. You must also take into consideration the interval used to operate the motors.

An example: You set the tempo of your melody that results in each eighth note lasting 25 hundredths of a second. The fingers already use 8 hundredths to move, so your delay will be 25 – 8 = 17, or 17 hundredths. Consequently, the quarter—one beat—should last 50 hundredths, resulting in a delay of 42, while a half note (100 hundredths) should have a corresponding delay of 92.

Inventing…

What to Play

Six notes seem so few to play anything. They are, but at the same time you'll be surprised how many known melodies will fit in that range. If you lack inspiration, look at some music scores aimed at beginning pianists: They usually contain many short and simple songs developed around five adjacent notes so the pupil doesn't need to move his or her right hand.

Changes and Improvements

The Pianist, like the Drummer, doesn't feature any input sensor. It is actually fit for the same kind of applications used for touch and light sensors: tempo changes and performance variations. With a sound sensor capable of decoding pitch, like the one described in Chapter 9, you can make your Pianist "learn" and reproduce what you play on a separate section of the same keyboard or on another instrument.

In case you have a second RCX, you can make it control the Pianist via IR messages. For example, you can build a robot that reads a special format score, decodes the notes and converts them into instructions for the Pianist.

If you want your own Pianist to play an organ rather than a piano (or, generally speaking, a keyboard that emulates an instrument that requires long notes), you must figure out a system to keep the keys down without damaging the motors. A simple solution requires that you replace the 24t gears on the motors with clutch gears, so you can keep the motor running without danger of damaging them. You'll probably discover that running them continuously is unnecessary, since you can brake the motors when the key is down, possibly turning them on once in a while if the note has to hold for a very long time.

Other Suggestions

The instruments we assigned to the robots of this chapter are not the result of a random choice: They are the easiest to build!

Percussive instruments in general, from triangle to vibraphone, are a feasible option. Keyboard instruments are, too, if you make minor adaptations to our design whenever necessary.

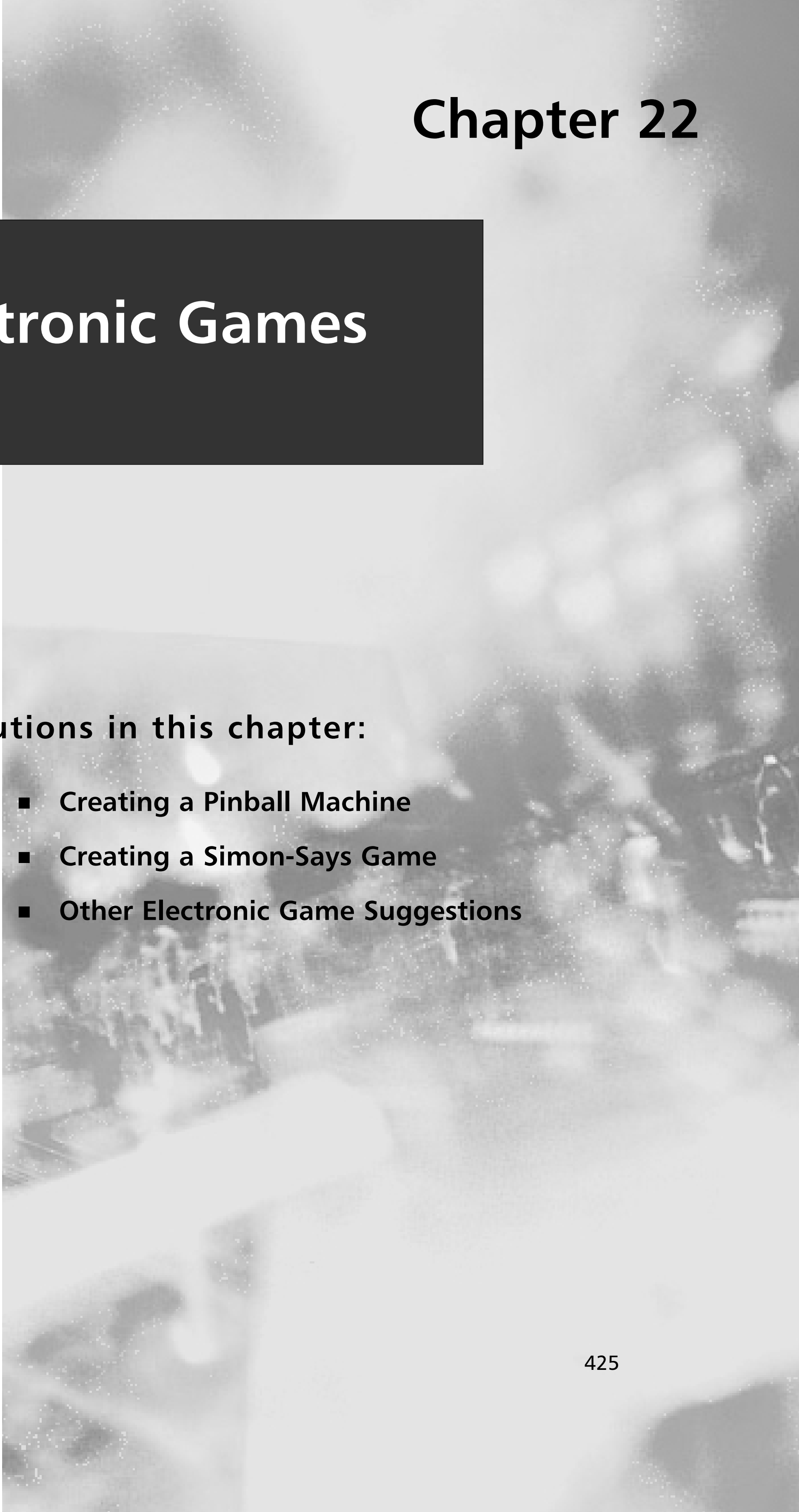
Strings instruments are a tough challenge: Playing a real guitar seems difficult enough, never mind playing a violin. You can probably make a very simplified LEGO version of a "guitar" where your robot controls the tension or length of a single string with a motor, while plucking it at the opposite end with a second motor.

Wind instruments appear out of range at the present time considering the quantity of air they require (even a recorder is too demanding for a LEGO compressor). But we'd love to be proven wrong on this topic!

Summary

The robots of this chapter are a bit low on the android evolutionary scale: Having no sensors they cannot interact with the environment. The Drummer is so simple it doesn't even have gears! However, they show some techniques useful for situations where you need a "fast touch," and the Pianist itself describes a practical application of the principle where a motor can control two mechanisms, each one needing only one direction. You have also discovered that sometimes you can resort to non-LEGO materials to complete your projects: The common plastic wrap is what made our robotic Drummer possible.

Well, if you were expecting LEGO robots capable of reproducing your favorite pop song, or to perform some classic Beethoven piano sonata, we probably disappointed you. Just the same, we hope our players surprised and amused you, and that they have provided some inspiration for your own musical creatures.



Chapter 22

Electronic Games

Solutions in this chapter:

- **Creating a Pinball Machine**
- **Creating a Simon-Says Game**
- **Other Electronic Game Suggestions**

Introduction

Today video gaming is so much a part of our lives today that we forget there was a period when they literally didn't exist. In those days, the amusement arcades were instead filled with amazing electromechanical devices like pinball and sports-based games with balls and targets, buttons and levers, and lights and buzzers. Many of the readers of this book are probably too young even to *know* that such a period existed.

Even here we can find uses for robotics. Actually, we hesitate to call the machines of this chapter "robots." In many aspects, they aren't. Nonetheless, we decided to include them in the book because they show yet another creative application of your MINDSTORMS system—and because they are such fun!

Creating a Pinball Machine

The pinball machine is probably the only survivor in the gaming arcade's evolution to electronics. Not that it enjoys good health exactly, in fact it lingers on the edge of extinction, but it has not yet entirely disappeared. Though many software versions of pinball exist today, anybody who played the *real* thing ages ago can tell you they don't come close to what the actual machines were like, or the feeling you got from playing them.

Our LEGO Pinball Machine, though extremely simplified, should be considered a member of full standing in that category of mechanical marvel.

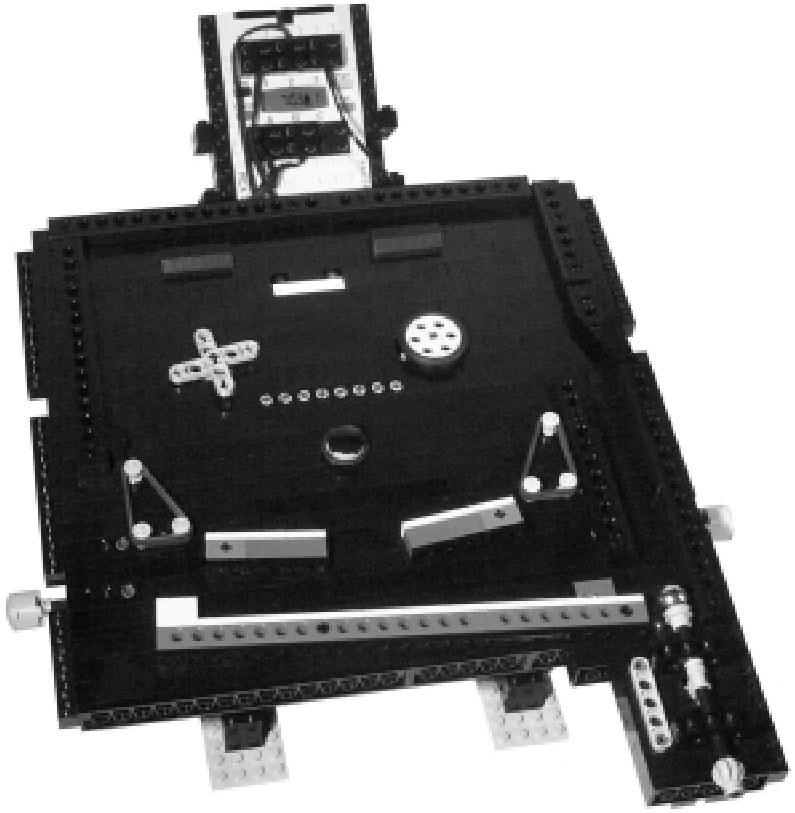
Building the Pinball Machine

Here it is, the Pinball Machine (Figure 22.1). We used an incredible amount of parts to build it, but there is some good news:

- Most of the parts are plain 2 x n bricks.
- Even for the special parts required, what you have in the MINDSTORMS box should be enough.
- You can easily adapt the project to your own inventory.

The surface of the pinball is built with bricks turned on their side (studs toward the RCX). This technique allows us to achieve a smooth surface without using tiles, and makes it easy to insert sensors, axle pegs, and anything else that needs a vertical mount over the surface.

Figure 22.1 The Pinball Machine



Describing the Pinball Machine from top to bottom of the figure, you'll notice the RCX, which displays the score and the count of the remaining balls. Below it there's a dual light brick, connected to an out port. The RCX can light it, for example, to mean that a special feature is active and something will happen when you score a point, like getting a bonus ball, extra points, or something else.

The cross on the left and the wheel on the right are both targets, respectively connected to a rotation and to a touch sensor, that will add points to your score when you hit them.

There's a Fiber-Optic System (FOS) unit under the surface connected to the row of eight gray pegs that shows below the sensors (FOS was described in Chapter 20). We used it mainly to enrich the Pinball Machine with more lights.

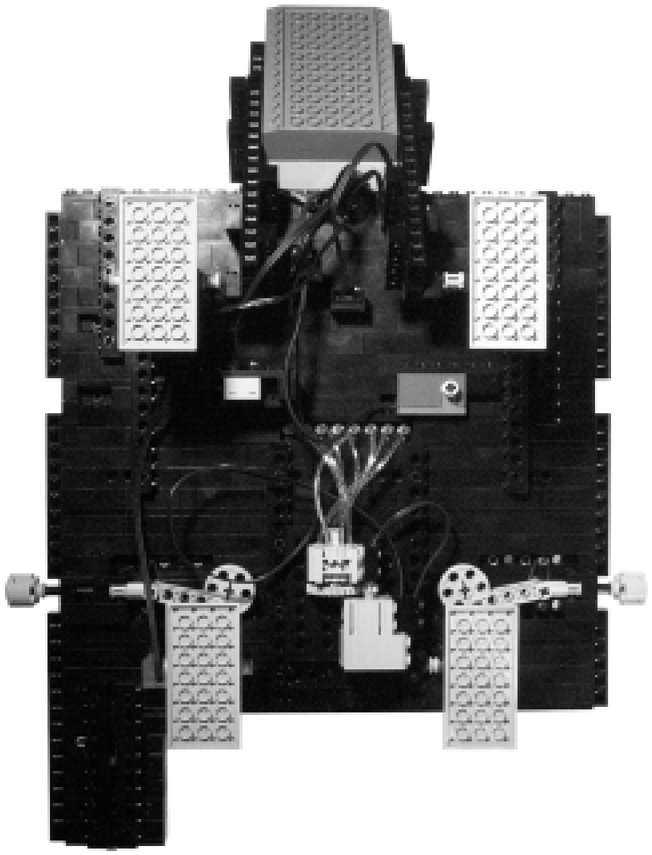
Right in the middle is a static obstacle, a 2 x 2 round brick surrounded by short rubber bands. At the left and right sides are two triangles similarly composed of rubber bands.

Finally, near the bottom of the picture you see the levers to hit the ball and the long slide that leads the missed balls to the throwing system on the right. During this path, the ball passes over a light sensor that detects it and makes the RCX decrease the count of the remaining balls.

We had planned to use the LEGO ball contained in the Soccer sets, but it is so lightweight that it flies off the playing field, so we had to fall back on a small steel marble. Small glass marbles will work as well.

The base of the Pinball Machine reveals its secrets (Figure 22.2). You see the four large and solid supports, the sensors incorporated in the structure, the FOS unit with its motor, and the mechanism that converts pressure on the side buttons to movements of the levers.

Figure 22.2 Bottom View



We connected the fibers to the FOS unit with an alternating scheme, so the lights appear to go back and forth even if the motor always turns in the same direction. This is the sequence, the first number of each pair being the number stamped on the unit and the second the position of the fiber in the beam: 1-1, 2-3, 3-5, 4-7, 5-8, 6-6, 7-4, 8-2. The flipper mechanism is clearly visible in the close-up of Figure 22.3.

Figure 22.3 Detail of the Push-Button Mechanism (Bottom View)

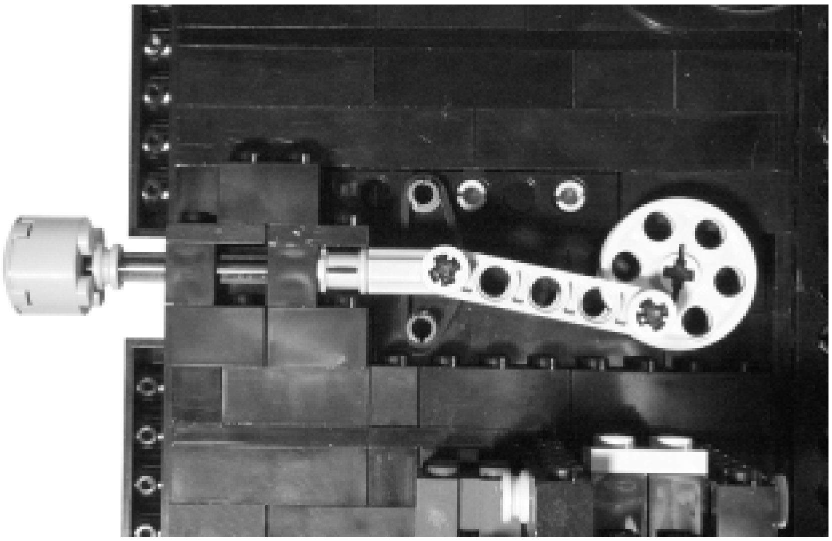


Figure 22.4 shows a detail of the launching system. Notice the light sensor on the left. The steel of the ball reflects the light very well, thus its passage over the sensor triggers high readings easily intercepted by the RCX.

Looking again under the surface, you discover that the launching system, like many other parts of the Pinball Machine, relies on rubber bands (Figure 22.5).

The touch sensor attached to the wheel stays closed with the help of yet another rubber band (Figure 22.6), and opens briefly when the ball hits the wheel from one of two possible directions (going up or down).

Figure 22.4 Detail of the Ball-Launching Mechanism

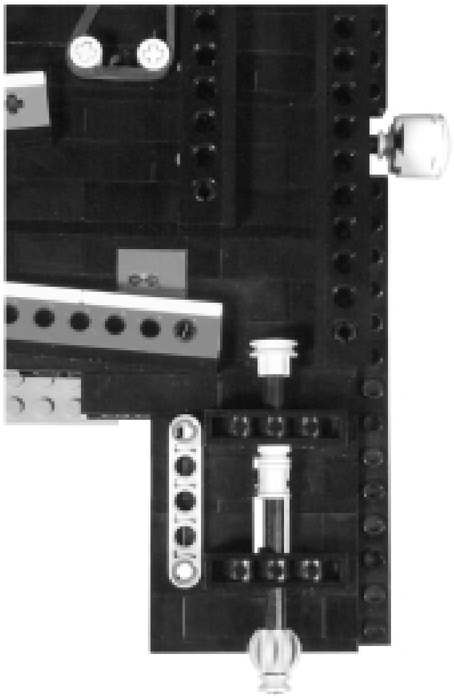


Figure 22.5 Detail of the Launching Mechanism (Bottom View)

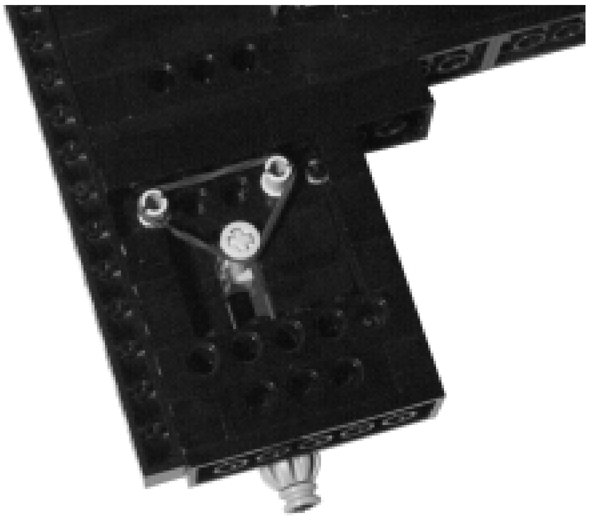
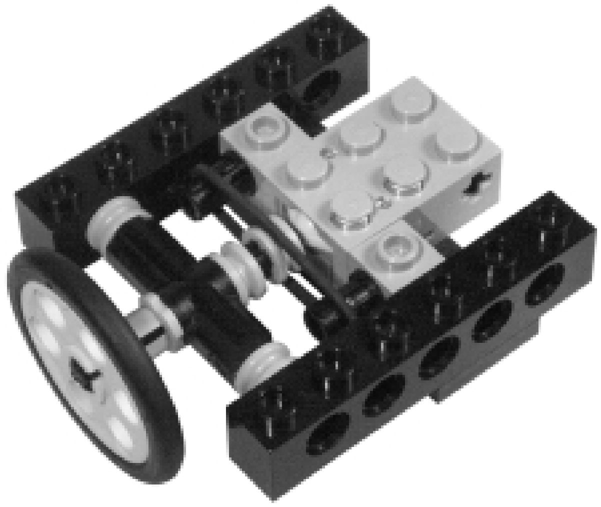


Figure 22.6 The Touch Sensor



Programming the Pinball Machine

The Pinball is not very complicated to program.When the game starts, reset the score to zero and the ball count to the initial number (e.g., five balls).Then start a loop that monitors the following events:

- If the ball passes over the light sensor, it decrements the count by one unit.When it goes down to zero, the game ends.
- If the ball hits the sensors, some points will be added to the score. If some special condition is on, the player may receive a bonus ball or some additional points.
- At random times, the *special* conditions and their lights (light brick, FOS) should turn on and off.

Our version uses three digits of the display to show the score, and the fourth one to indicate the ball count.

Don't forget to add plenty of sound effects; they add a lot to the game. Use the techniques described in Chapter 6 to include background music as well as different sounds to highlight individual events while the game's in play (for example, a ball is lost, a bonus is gained, points are scored, and so on).

Improvements on the Construction

In creating your own Pinball Machine, employ a boundless imagination. Make the machine larger or smaller, add lights, obstacles, sensors, rotating devices, etc.

This is a good place to apply some of the tricks described in Chapter 4 about connecting more sensors at the same port. You can add as many touch sensors as you want to a single input port (all of them will score the same points when touched), or combine a light sensor with one or more touch sensors.

FOS units can work at the same time as rotation sensors and output devices, as described in Chapter 20.

Output ports can drive not only lights but motors, too. We didn't use them, but it's easy to devise spinning mechanisms that throw the ball off course. With the appropriate gearing, a single motor could rotate many of them.

There are so many possible variations and so few constraints that the only limit is your imagination.

Creating a Simon-Says Game

The electronic game we are going to describe is inspired by Milton Bradley's commercial product *Simon*, which has had great success among families. It's not an arcade game, but it's still very challenging and devised so players can compete against themselves or others.

The game features lights of different colors coupled to push-buttons; there were four in the original, in our version there are three. Every time a light turns on or the corresponding button gets pressed, the machine also produces a sound, one that's always the same for each color.

The purpose of the game is to copy the sequence performed by the machine. It starts with just one light, then the sequence of lights becomes longer and longer as the game proceeds. Every time the user successfully reproduces the series, the machine plays it again, adding one more light to the end of the sequence.

Building a Simon-Says Game

Our Simon-Says uses three touch sensors, three light bricks, and no motors. Its hardware is so simple that it barely deserves a description, being just a flat surface with sensors incorporated into it and four legs as supports (Figure 22.7).

Each light brick-touch sensor pair forms a unit that connects to both an output and input port. Figure 22.8 shows such a unit in detail. The lamps face transparent 1 x 2 bricks of different colors. We used Red, Green, and Blue.

Figure 22.7 The Simon-Says Game

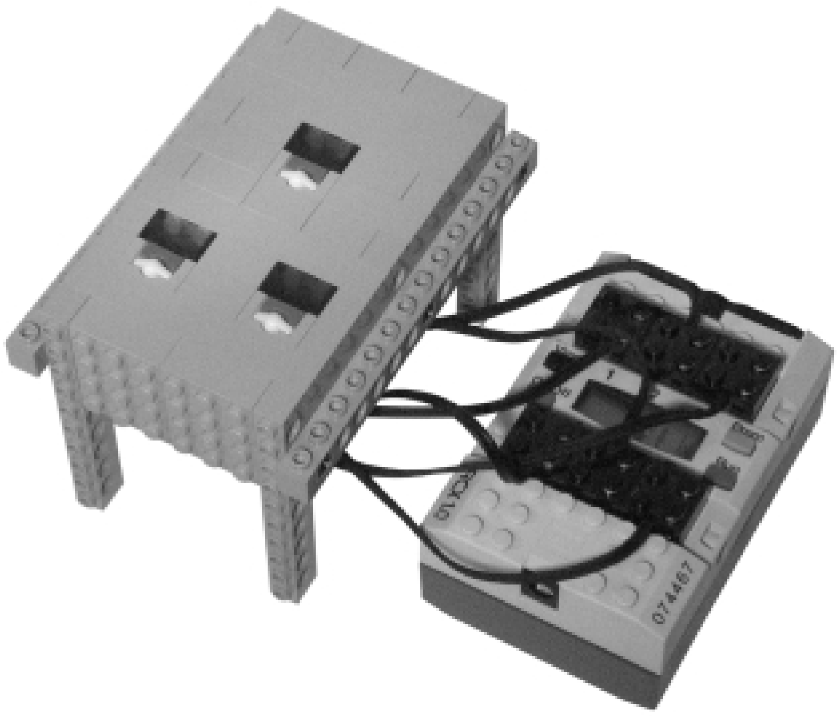
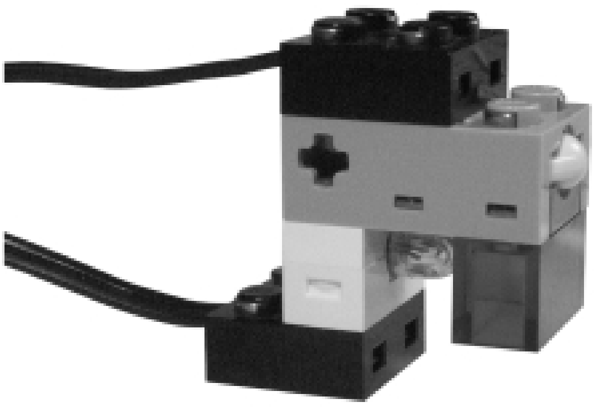
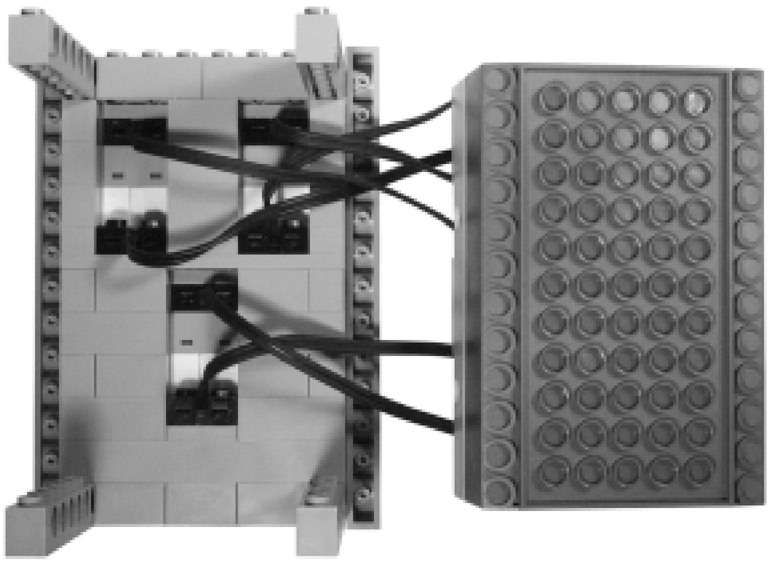


Figure 22.8 The Particulars of a Light-Button Unit



The bottom of the machine has little to show, just six cables that connect to the RCX (Figure 22.9).

Figure 22.9 Bottom View



Programming the Simon-Says Game

The software required by this game is a bit trickier than the one for the Pinball Machine. Our version uses an array to store the sequence, generated only once at the beginning of the game, when the user pushes the Run button. Each cell of the array contains a value that tells it which lamp to turn on.

In the main task of the program, you put a loop that repeats the **play_sequence** and **check_sequence** subroutines for series that each time become longer by one unit. The following NQC code describes the core sections of our implementation:

```
generate_sequence();

n=1;
error=0;

while (n<MAX && error==0)
{
  play_sequence(n);
  check_sequence(n,error);
  Wait(100);
  n++;
}
```

The variable error goes to 1 when **check_sequence** detects a mistake in the user's input, thus forcing the termination of the program, while n represents the length of the sequence. MAX is a constant corresponding to the dimension of the array, representing the maximum length of the sequence.

The generate sequence routine fills the array with random values in the range 0 to 2. In our example, we associated values of 0, 1, or 2 to the colors green, blue, and red, respectively.

```
void generate_sequence()
{
  int i;
  for (i=0;i<MAX;i++)
  {
    seq[i]=Random(2);
  }
}
```

The play_sequence subroutine needs little explanation. It plays the first n values of the array, spaced by a short delay:

```
#define DELAY 30

void play_sequence(int n)
{
  int v,i;
  while (i<n)
  {
    v=seq[i];
    play(v);
    Wait(DELAY);
    i++;
  }
}
```

The play function mentioned in the code turns on the light and sound corresponding to the value of its parameter. Our program keeps both the light and sound on for 20 hundredths of a second and leaves a delay between them of another 30 hundredths. The note we chose to combine with the lights corresponds to the frequencies 262 (C), 392 (G), and 523 (C), which we assigned to constants.

```
#define S_GREEN 262
#define S_BLUE 392
#define S_RED 523
#define ON_TIME 20

void play(int v)
{
  switch (v)
  {
    case 0:
      On(OUT_A);
      PlayTone(S_GREEN,ON_TIME);
      break;
    case 1:
      On(OUT_B);
      PlayTone(S_BLUE,ON_TIME);
      break;
    case 2:
      On(OUT_C);
      PlayTone(S_RED,ON_TIME);
      break;
  }
  Wait(ON_TIME);
  Off(OUT_A+OUT_B+OUT_C);
}
```

Notice that we are defining constants for all the parameters of the program. This is a very good practice which makes the tuning of your program much simpler. If you group all the **#define** statements at the beginning of your code, you have a sort of control panel where you can change the parameters of the application without having to look for constants scattered along the program.

The **check_sequence** routine represents the most difficult part of the job. It must check that the inputs follow the proper sequence, and that not too much time elapses between them. This is a possible outline of it:

```
#define LIMIT 20

void check_sequence(int n, int & error)
{
  int answer,i,err;
  i=1;
  while (i<=n && error==0)
  {
    answer=-1;
    ClearTimer(0);
    while(Timer(0)<LIMIT && answer<0)
    {
      if (SENSOR_1==1) answer=0;
      else if (SENSOR_2==1) answer=1;
      else if (SENSOR_3==1) answer=2;
    }
    if (answer>=0)
      play(answer);
    if (answer!=seq[i])
    {
      error=1;
    }
    i++;
  }
}
```

You notice there's an outer loop that executes until the whole sequence has been replicated or until the user makes a mistake. Inside this, an inner loop waits for the pressure of a sensor or for the expiring of the time limit.

The game ends when the user makes a wrong choice. His score corresponds to the length of the longest sequence he successfully replicated, and you can show it on the display at the end of the game.

Variations

If you don't have the light bricks, you can make your own custom version using common and cheap LEDs. Some of the sites mentioned in Appendix A explain how to do this. LEDs have the advantage that you can buy them in different colors so you don't need the transparent bricks as well. Since they have a polarity, they will turn on only when supplied power in the proper way. You can take advantage of this fact to control two of them from a single port, allowing your Simon-Says to have up to six lights (but this means you'll need a multiplexer for the input sensors).

On the software side, this simple platform supports a few variations to the base game. The easiest version, suitable for very young children, requires that they simply press the proper button, as if the sequence were always just one light long.

A different game uses random sequences of fixed length that change every time. The score corresponds to the time the user takes to replicate a given number of them.

Other Electronic Game Suggestions

During the early 70s, arcades were filled with incredible games, all technologically similar to that of the pinball machine. Each had lights, motors, relays, electromagnets, but few, if any, electronics.

The following are a few ideas we hope might inspire you in creating your own games:

- **Sink the Ship** A ship goes back and forth in front of you, and you must try and sink it with a torpedo. Both the ship and the torpedo are powered by motors (using rack and pinions, for example). If the ship is in the same spot where the torpedo ends its run, you score a point.
- **Ping-Pong** An XY system like the one we built for the Maze Solver can make a ball bounce inside a rectangle. The ball is a small light directed upward and is visible through a panel of translucent material. (For example, large LEGO transparent plates, if you have them.) Add ratchets and shot detection and you're done.

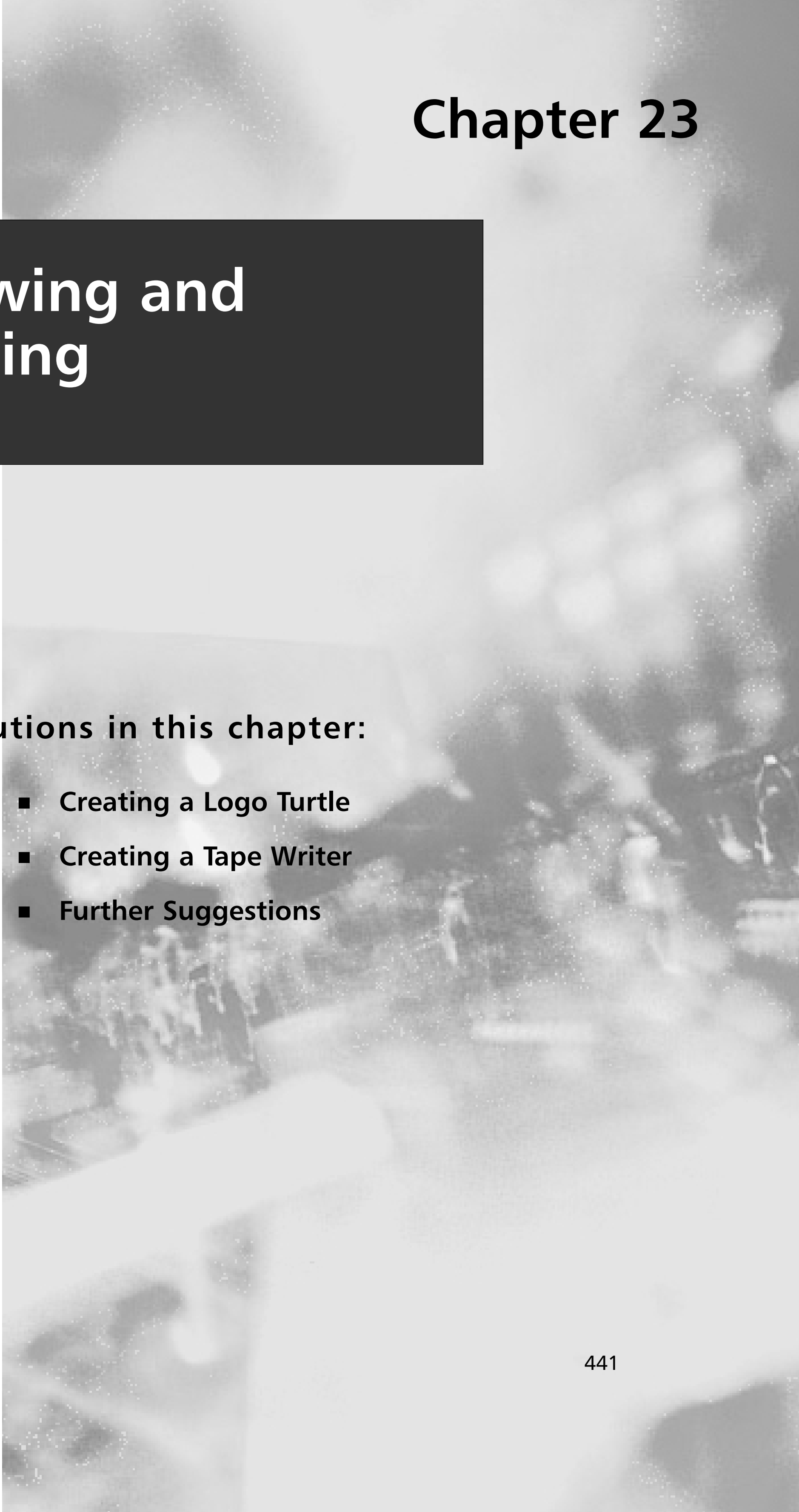
- **Shoot the Tank (or Bear, or whatever else strikes your fancy)** This one's as simple as making a robot that randomly runs around your room featuring a light sensor that you have to hit with a laser pointer. (Don't forget that laser light is dangerous if directed at the eyes.)

Summary

These machines are games rather than robots, but they suggest some creative uses for your MINDSTORMS parts. In the Pinball Machine, we presented some new solutions aimed at detecting a ball. (A process that will prove useful in other applications.) For example, with the help of a rubber band we made a touch sensor sensitive to collisions coming from two directions. You could employ this technique to build bumpers that detect both a "push" and "pull" action. We showed also that a light sensor can detect the passage of a ball; the principle behind this is that the ball affects the amount of light reflected into the sensor, and you monitor this change from your program.

The Simon-Says game is a good example of a MINDSTORMS machine with no mechanical parts: It demonstrates that sensors and lamps can be combined into an effective interface, and that lights can be driven from output ports just as easily as motors. It also applies some of the concepts of Chapter 7 regarding using sounds to communicate with the user.

In our opinion, however, the most attractive feature of these games is that they are so much fun to imagine, build, program, and use. Your inventions will be a big hit with your friends and kids. Our versions have been played a lot—just for testing, of course!



Chapter 23

Drawing and Writing

Solutions in this chapter:

- Creating a Logo Turtle
- Creating a Tape Writer
- Further Suggestions

Introduction

Can a MINDSTORMS robot be made to draw or write? Sure. Believe it or not, that's not even a very difficult thing to implement. In the following pages, we will show you two projects, the first mainly meant for drawing and the second for writing. Both of them require some additional parts, but both have wide margins for modifications and allow for less demanding variants.

Creating a Logo Turtle

Many of you may already know that Logo is a programming language specifically targeted to education. Born in the late 60s at the Massachusetts Institute of Technology (MIT), Logo is derived from Lisp (with a lot fewer parentheses!) and features interactivity, modularity, and extensibility. More than a programming language, Logo is a learning tool which has gone through a number of changes and improvements over the years.

The most known characteristic creation of Logo is the *Turtle*, a symbolic turtle that moves across the computer screen according to the instruction it receives. With simple instructions like forward 10 the turtle moves straight ten units, and with right 90 it turns clockwise 90 degrees. The statements penup and pendown specify whether the turtle leaves a track behind it, thus producing drawings or rather just moving to a different location. Obviously the language includes many other commands, but these are enough to understand the principles of the *Turtle Graphics* that made Logo so famous.

What many people don't know is that in its first version, the Logo program controlled a small robot that actually drew lines on the floor. In subsequent releases, the turtle became just a virtual animal on the screen. Our interest here, however, is in replicating its first robotic version.

NOTE

Dr. Seymour Papert was one of the early promoters of Logo, and designed the original Turtle. Under his guidance, the Epistemology and Learning Group at MIT devised the first *programmable brick*, whose concepts led to the development of the LEGO MINDSTORMS line.

Building the Turtle

The idea is quite simple: Build a small robotic platform that's able to go forward and backward, turn in place, and lower and raise a pen. Despite this apparent simplicity, if you want a turtle that works as expected, the task has many stringent requirements that must be adhered to. For instance:

1. The robot must go absolutely straight.
2. The pen must be exactly in the pivoting point of the robot, because it must stay in the same place on the floor while the robot turns (otherwise it would trace a curve).
3. You need a tracking system to measure both traveled distances and angles.

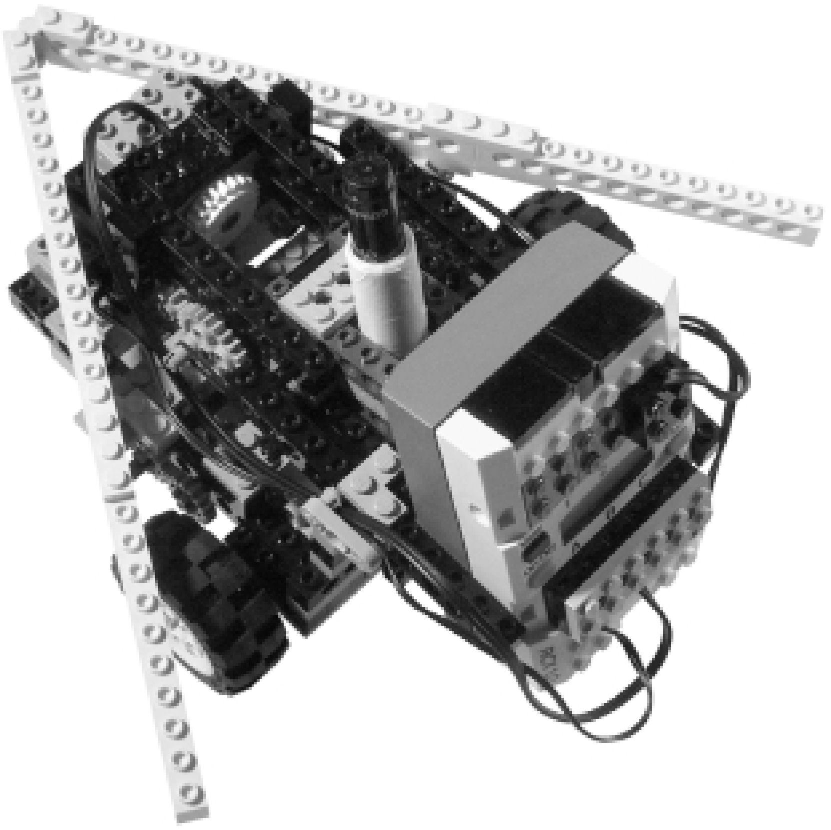
If you remember the driving architectures described in Chapter 8, you already know the solution to the first point: Use a dual differential drive. The simple differential drive is suitable for this project only if you apply an active control to the wheels to be sure they travel exactly the same distance, while the synchro drive would work as well but at the price of greater complexity and not so evident change in orientation during action. Another advantage of the dual differential drive is that it requires a single encoder to comply with point 3: when the robot goes straight it measures the covered distance, when turning it measures the angle.

OK, so we have requirements 1 and 3 covered, but there's still the matter of the pen being the center of rotation, which is at the midpoint of the imaginary line that connects the wheels. Conceptually it sounds easy, but you have to build your robot with this point in mind.

The original turtle—a differential drive—featured a transparent plastic dome to cover the gears. We provided our turtle with a triangular shape (Figure 23.1), because we wanted to mimic the screen turtle of some widespread Logo systems. Anyway, those V-shaped beams are definitely not necessary and you can shape your own turtle according to your wishes.

Our differential drive does not use a caster wheel, because they tend to affect the direction of the robot slightly when resuming straight motion after a turn. With casters, the straight lines would have a short wiggly segment, so we preferred to use a simple tile as the third supporting point. To keep the friction on the floor to a minimum, we placed the RCX suspended behind the drive wheels, like a sort of counterweight, bringing the COG of the robot very close to the drive axles and thus most of the weight upon the drive wheels.

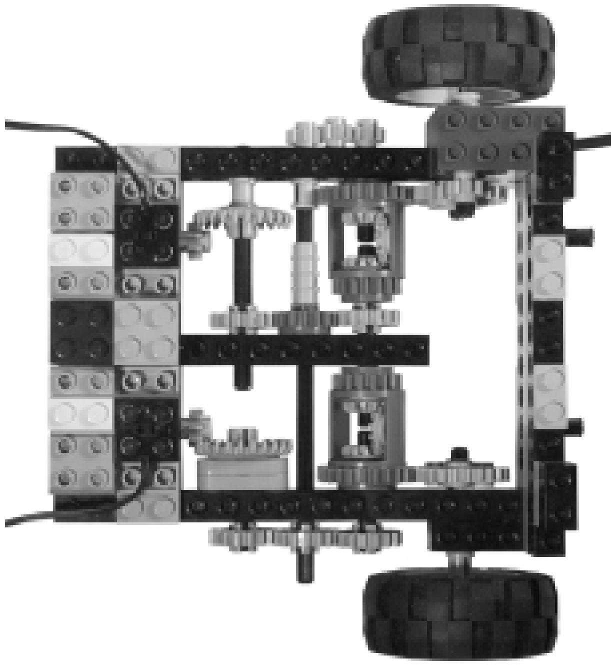
Figure 23.1 The Logo Turtle



There's another advantage to having the RCX pointing upwards: This maximizes the possibilities of communications between the tower and the robot, using the ceiling of the room as a mirror for the infrared (IR) rays (see the sidebar, "What's Infrared Communication?").

Let's start exploring the dual differential drive chassis that drives the robot (Figure 23.2). The gearing is more compact than those shown in Chapter 8, but it works exactly the same way: One motor makes the differential gears and the wheels rotate in sync, while the other rotates them in the opposite direction. You can notice the rotation sensor coupled to the right wheel. The dark gray 16t gear right in the middle of the photo is an idler gear which connects the other two 16t gears; its center hole is not cross-shaped and thus it doesn't couple with the long joined axle that crosses the base of the robot.

Figure 23.2 The Turtle Dual Differential Drive Platform (Top View)



Bricks & Chips…

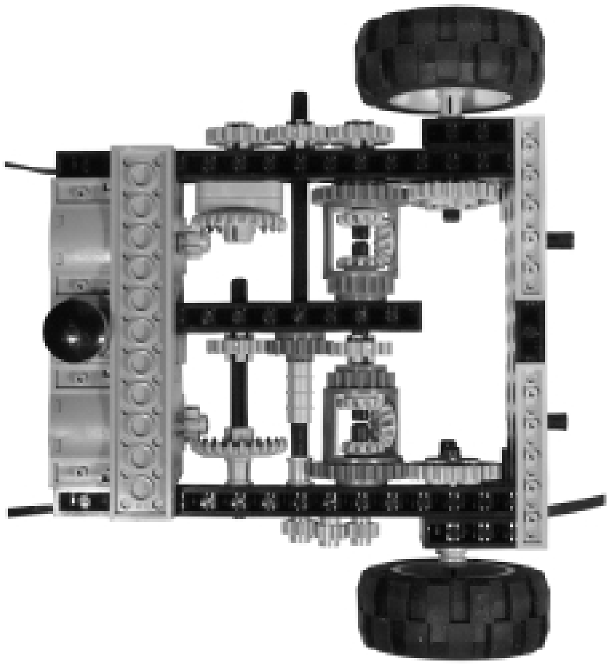
What's Infrared Communication?

Infrared (IR) light is of the same nature as visible light, but its frequency is below that perceivable by the human eye. Provided the intensity is high enough, we usually feel IR radiation as heat.

For most properties, IR light is really identical to visible light: It gets reflected, refracted, diffused, or shielded by different kinds of bodies. When you want your robot to stay in communication with the tower, they must "see" each other all the time. This is not always easy when the robot moves and changes orientation, but for indoor situations, you can take advantage of the ceiling, as described, to reflect the IR beams downward. In most cases, placing the RCX with the tower facing upwards works very well and solves the problem.

Looking at the bottom, you can see the front skid plate. (Figure 23.3).

Figure 23.3 The Turtle Dual Differential Drive Platform (Bottom View)



Using a technique described in Chapter 11, we placed a rubber band to make the mechanism bi-stable, so that when the pen is down it tends to stay down, and vice versa (Figure 23.4).

The pen is a non-LEGO part, a common marker with its body wrapped in adhesive tape so as to make it fit tightly into the 2 x 2 studs squared hole reserved for the purpose. It stays there with nothing but friction.

The pen control mechanism is a swinging assembly operated by a third motor (Figure 23.5).

Now the turtle is ready. Place a large piece of paper on the floor, uncap the pen and adjust its height so it touches the paper gently when in it is in the down position (Figure 23.6). We strongly discourage you from writing *directly* on the floor. We're sure *somebody* won't like it!

Figure 23.4 Side View of the Turtle Pen Mechanism

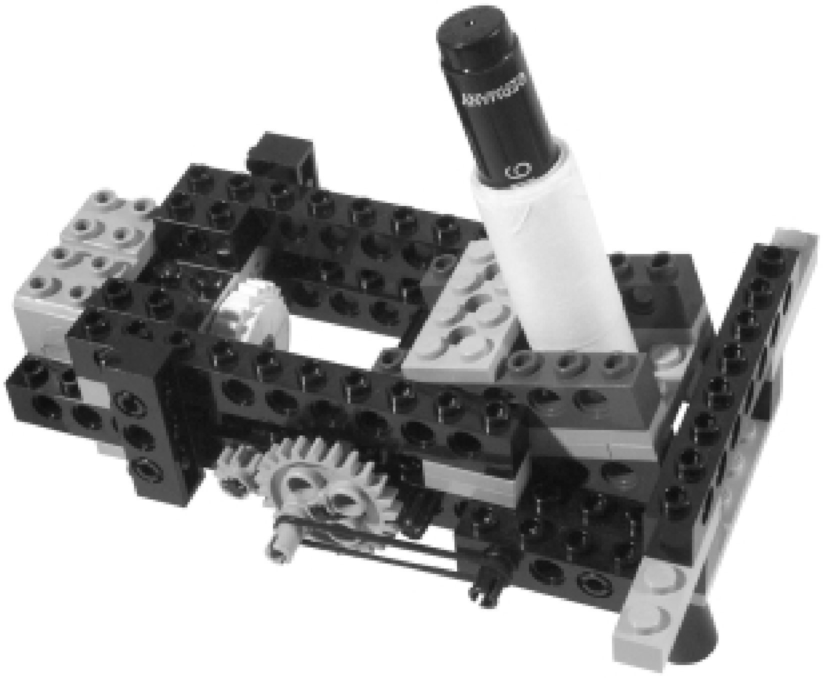


Figure 23.5 Turtle Top View

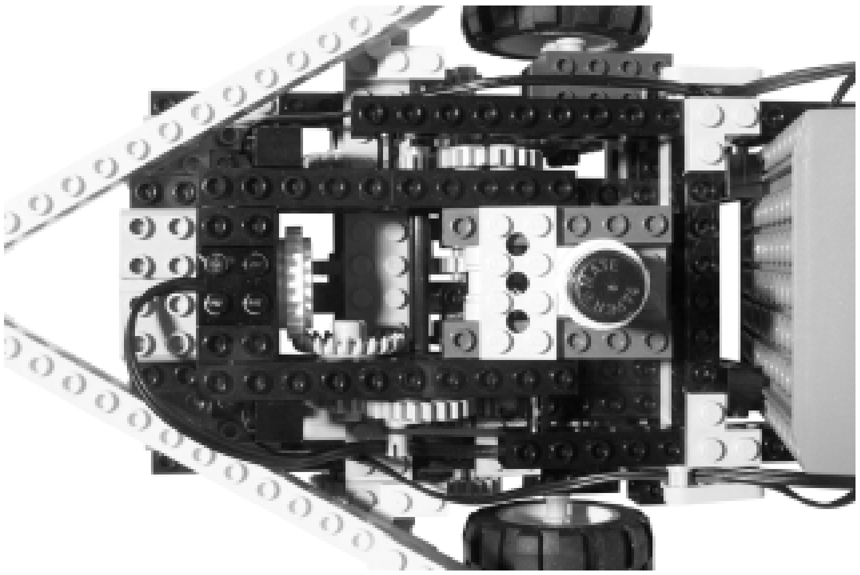
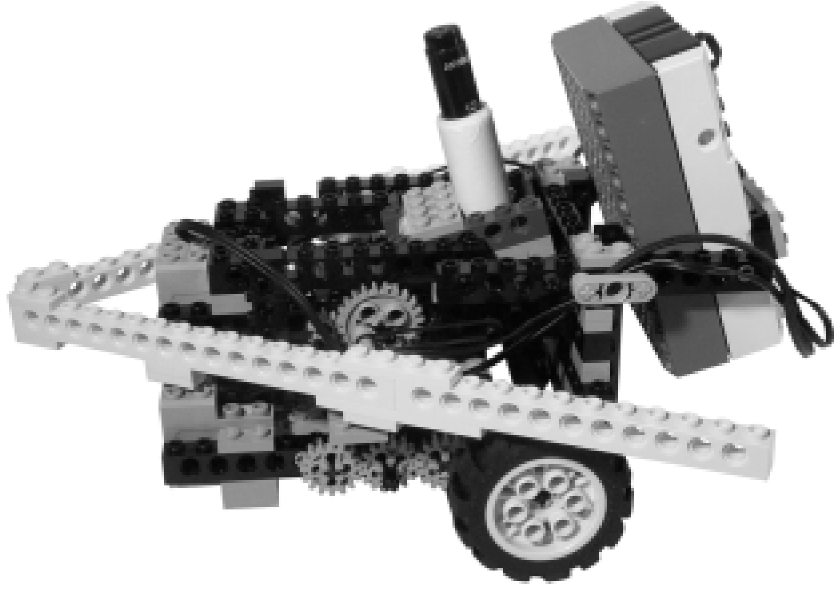


Figure 23.6 Side View of the Turtle Ready for Operation



Programming the Turtle

The first task in programming the Turtle is to create the primitives that control the basic actions. Let's start with the easiest ones: the **penup** and **pendown** commands. A short impulse to the pen motor does the trick—nothing more is required. If you want to avoid lowering the pen again when it's already down, in case of repeated **pendown** commands, you can monitor the status with a variable. In the NQC example that follows, we defined two constants UP and DOWN to make the code clearer; in fact, the instruction pen=DOWN is much more self-explanatory than its equivalent pen=0.

```
#define DOWN 0
#define UP 1
#define PEN_TIME 15
int pen;

sub pendown()
{
  if (pen==UP)
  {
    OnFwd(OUT_B);
    Wait(PEN_TIME);
    Off(OUT_B);
    pen=DOWN;
  }
}
```

The constant PEN_TIME will be typically something like 15 or 20 hundredths of a second. The penup routine is obviously identical except for the direction of the motor and the values the pen variable is tested and assigned.

The **forward** and **back** commands, meanwhile, are not very difficult to implement, but require that you dig into the physical properties of your robot. You must discover what distance it covers for any increment of the rotation sensor. The model is the same as that explained in Chapter 13 when we discussed dead reckoning, but here it is simplified by the fact that the wheels always travel at the same speed. The equation was:

$$F = (D \times \pi) / (G \times R)$$

where D is the diameter of the wheel, R the resolution of the rotation sensor, and G the gear ratio between the sensor and the wheel. We used a wheel with a nominal diameter of 5 cm. The resolution of the rotation sensor is 16 counts per turn, and is geared 1:3 with the wheel—thus our formula becomes:

$$F = 5 \times 3.1416 / (3 \times 16) \approx 0.327 \text{ cm}$$

This means that every time the sensor counts one unit, the wheel covers about 0.327 cm. What actually interests you is how many ticks you should count to cover a required distance, so the formula becomes:

$$\text{Count} = \text{Distance} / F$$

If you have to manage the calculation with whole numbers, be sure to express the formula in your code in order to keep maximum precision. Dividing for 0.327 is like multiplying by 3.06, from which the following code:

```
count = (dist * 306) / 100;
```

This is the theory. The actual robot will probably require some in-the-field tuning, because the distance covered by the wheels is affected by other factors: The weight compresses the tires and reduces their diameter. There might be some slippage, too. We suggest you proceed by experimentation, making your turtle draw a line, measuring it and then correcting the factor until you're happy with the result. All this process is meaningful only if you care about having your turtle use units that correspond to some common length unit. If you don't care, simply use the rotation sensor counts as units.

With the required count determined, your subroutine will simply reset the rotation sensor and start the motion motor until that count is reached. The NQC example that follows assumes that the rotation sensor is attached to input port 3, and that the motor that drives the turtle straight and forward is powered by output port C:

```
void forward(int dist)
{
  int count;
  ClearSensor(SENSOR_3);
  count=(dist*306)/100;
  OnFwd(OUT_C);
  while (SENSOR_3<count);
  Off(OUT_C);
}
```

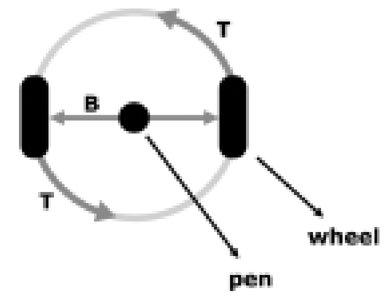
The **back** subroutine will be symmetrical to this one, with the motor reversed and the sensor counting negative numbers until a negative limit is reached.

Now the last part: the turning primitives right and left. If you glance again at Chapter 13, you find that the change in orientation ΔO_R (in radians) depends on the distance covered by the wheels ($T_R - T_L$) and the distance between the wheels (B). When the dual differential drive turns in place, both the wheels travel the same distance (T) in opposed directions, so we can express the equation in simplified terms:

$$\Delta O_R = 2 \times T / B$$

This relationship is shown in graphical form in Figure 23.7. Don't worry if you are not familiar with measuring angles in radians. We are going to convert radians into degrees in a few steps.

Figure 23.7 Computing Changes in Orientation



Actually you know the ΔO_R you want to get, it corresponds to the desired turning angle of the turtle, and it's the input to your subroutine. What you're looking for is the Count of the rotation sensor that produces that ΔO_R. The first step is to obtain T from the previous equation:

$T = \Delta O_R \times B / 2$

As Count corresponds to T / F, you get:

$\text{Count} = (\Delta O_R \times B / 2) / F = \Delta O_R \times B / 2F$

Here is where you convert radians to degrees, using the following formula:

$\Delta O_R = \Delta O_D \times \pi / 180$

Applying the conversion to the previous equation for Count, the new expression becomes:

$\text{Count} = (\Delta O_D \times \pi / 180) \times B / 2F = \Delta O_D \times \pi \times B / (360 \times F)$

In our turtle B is 18 studs, that is 14.4 cm. (It's important you express both F and B in the same unit, whatever you choose.) Computing the expression $\pi \times B / (360 \times F)$ into a single constant, we get the value 0.384. Multiplying by 0.384 is like dividing by 2.6, so you could write your code as:

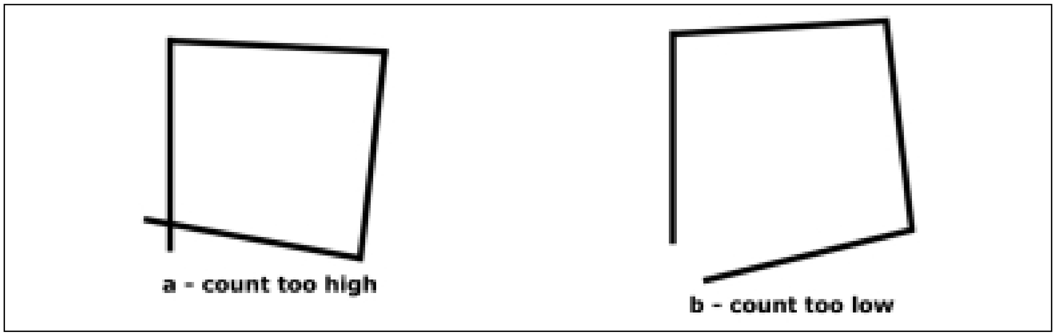
```
count = angle/2.6;
```

Where angle is the desired turning angle, and count the number of ticks of the rotation sensor that correspond to that angle. But there is no floating point math on the RCX, so you cannot divide by 2.6 and you need to scale your numbers up to express this ratio using integers (see Chapter 12). Here is the complete NQC implementation of the **right** routine (the turning motor connected to output port A):

```
void right(int angle)
{
  int count;
  ClearSensor(SENSOR_3);
  count=(angle*10)/26;
  OnFwd(OUT_A);
  while (SENSOR_3<count);
  Off(OUT_A);
}
```

As for the motion routines, this one will need some adjustment too. Your turtle is not likely to draw proper angles on the first try. We suggest you make it draw a simple polygon, like a square or an equilateral triangle, and check if it closes the path properly. For example, the sequence of four forward(20) right(90) should draw a square; if the last segment intersects the first one but not at its starting point, the count is too high, and you have to increase the divider (e.g., 27 instead of 26), and vice versa: If the square doesn't close at all, you should decrease the divider (Figure 23.8).

Figure 23.8 Tune Calculations by Testing Your Turtle in Drawing a Square



Instead of working on the software, you can often change the geometry of the robot. Altering the distance between the wheels by moving them in or out along the axles is a very effective way to tune the robot. Make small adjustments until your square comes out perfect.

Looking at the problem another way, you can force a specific result from the expression π x B / (360 x F), for example 1/3:

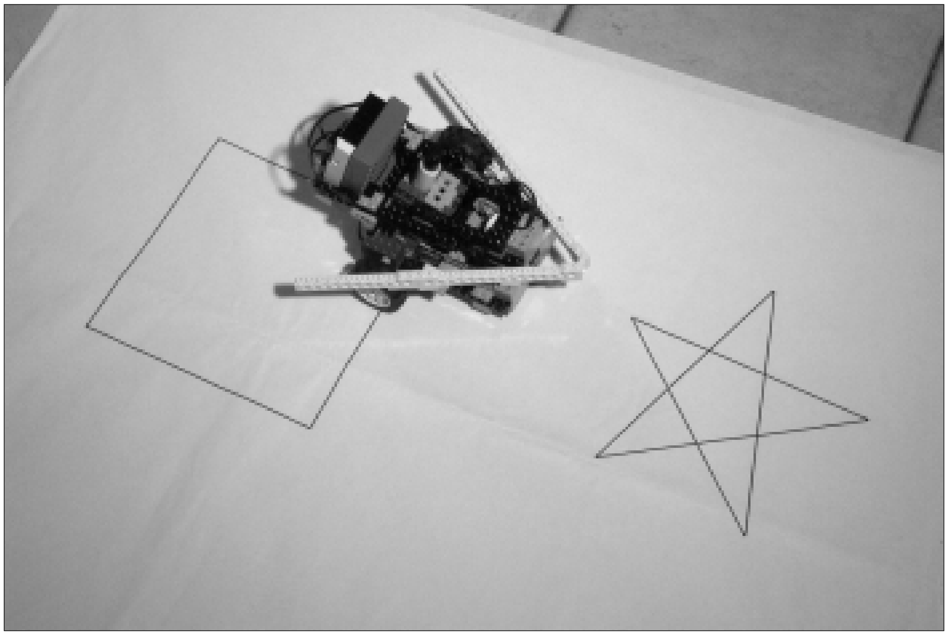
π x B / (360 x F) = 1/3

B = (360 x F) / (3π) = 12.49 cm

We suggested 1/3 because this would lead to the code count=angle/3. That's very simple, but more importantly, it doesn't suffer from rounding errors on many common multiple of 3 angles like 30°, 60°, 90°, 120°, 180°, etc.

Work patiently on your turtle and its code. The result will astound you! Figure 23.9 shows our turtle drawing an almost perfect five-pointed star and a square.

Figure 23.9 The Logo Turtle in action



Choosing the Proper Language

As a general principle, in this book we don't suggest any specific language for you to program your RCX. We provided examples in NQC just to find a common background with the readers to discuss programming issues; however, most of the projects in this book can be translated into any language of your choice. In this chapter, we make an exception—we are going to recommend a particular language, because we feel that the Logo Turtle project would benefit from a lot of interactivity.

A product specifically designed to marry LEGO to Logo does exist, it's the DACTA Control Lab Learning Environment. But it's been designed to interface with the Control Lab, not the RCX, and is a very expensive product. Staying in the range of free software, in our opinion the best choice for this project is pbForth, whose interactivity corresponds to the original Logo philosophy. PbForth uses the Reverse Polish Notation, so your statements will become reversed in respect to the Logo ones: FORWARD 10 will become 10 FORWARD, but is this a serious problem? With pbForth you can sit at your PC, type some commands on the console and watch the turtle execute them. Like in Logo, you can easily define new words to draw complex shapes, all this without being subjected to a compile-upload cycle.

If you don't want to approach pbForth, but still would like to make your turtle interactive, you have other options. One is sending commands through IR messages. You have 8 bits for each message, and can use three of them to code the main commands (**forward**, **back**, **right**, **left**, **penup**, **pendown**) and the remaining five for the command parameter. Five bits correspond to 32 different values, enough to code the angles, for example, in increments of 15°. Alternatively, you can design a two-byte messaging system, where the first byte corresponds to the command and the second byte to its parameter.

Another option could be to write an interface for your PC that performs all the computations and sends direct bytecodes to control the robot, while at the same time polling its sensors.

Variations

Let's examine what you can do in case you need some of the parts we used for this project, starting with the second differential gear. Using only one differential, you can make it subtract the motion of one wheel from the other, so that the differential case remains stationary when the robot goes straight (see Chapter 8). In this setup, you connect the rotation sensor to the body of the differential and ensure that during straight motion it doesn't rotate, slowing down the faster motor of the two when necessary. You have to rely on temporization to control the distance to cover during straight motion, but for turns, which are more difficult to control, you can still use the rotation sensor exactly as in the case we described.

If you have two rotation sensors, you can build your turtle on a simple differential drive, and concentrate your efforts on the software that would have to keep the wheels in phase both during straight motion and turns.

On the other hand, if you have no rotation sensor, you should dig back into Chapter 4 to see how you can make a fake one using other kinds of sensors.

Provided that the paper sheet provides a good color contrast against the floor, you can place a downward-looking light sensor to prevent your Turtle from accidentally writing off the edge of the sheet.

Tape Writer

The second project of this chapter uses an approach somewhat opposite to that of the Logo Turtle: Here it's the paper that moves, while the robot stays still. The principle is similar to the one used in ink-jet printers: A mechanism feeds the paper under a writing head, which by itself moves perpendicularly to the direction in which the sheet advances. From what you learned in the previous chapters, you can tell that such a system has two degrees of freedom, controlled respectively by a paper feeding motor and by the writing head motor (actually, our robot implements a third DOF, needed to move the pen up or down over the paper). This Tape Writer is also a Cartesian system, because the movements of the mechanisms are linear and perpendicular to one another.

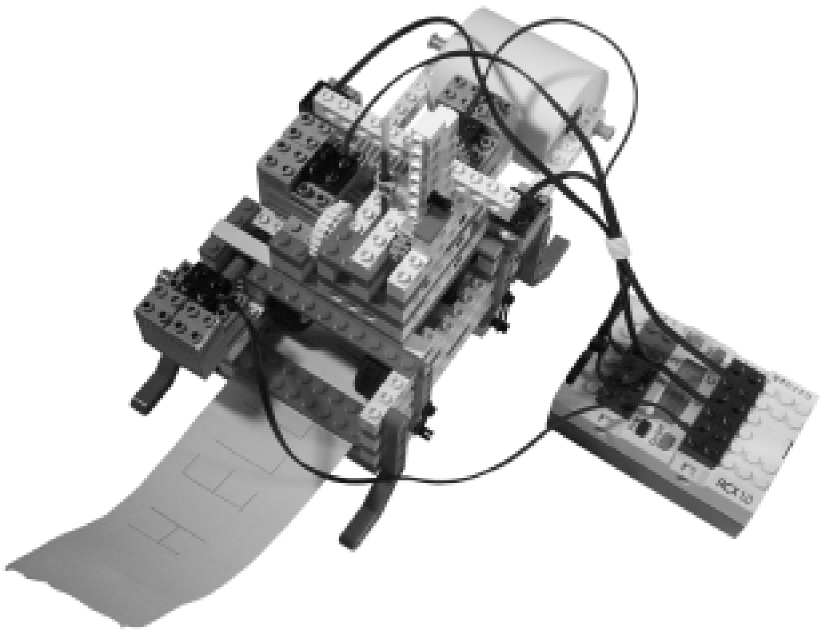
This robot requires some extra parts: a motor, some plates and beams, and many tiles; however, if you don't have the needed parts, there are many things you can do to downsize the project to keep within your inventory (we'll describe some of them).

Building the Writer

What we have in mind is a robot that writes on one of those common paper tapes made for printing calculators or cash registers. One motor moves the paper strip forward and backward, while a second moves the pen in a perpendicular (side-to-side) direction. The third motor controls the up/down pen movements.

Starting from the end, here's our finished robot that writes the traditional "Hello world!" welcome sentence (Figure 23.10).

Figure 23.10 The Writer Composes Its First Sentence



Analyzing the Tape Writer in detail, you can see that it's made of a body and three subsystems, all of them with one degree of a freedom.

- The body provides the main structures and hosts the paper transport system.
- There's a movable carriage over the body, which transports the pen in a direction perpendicular to the tape.
- Over the carriage, the pen assembly moves up and down.
- At the bottom of the body, there's the writing surface, a smooth surface that presses the paper against the wheels.

Looking inside the main body, you catch a glimpse of the transport wheels and the pen assembly (Figure 23.11).

The wheels are operated by a motor through a worm gear and three connected 24t gears (Figure 23.12). This latter geartrain is necessary to keep the two groups of dragging wheels turning in the same direction. You need the paper to go back to shape some letters, and this is the reason why there are wheels both before and after the pen.

Figure 23.11 Writer Side View

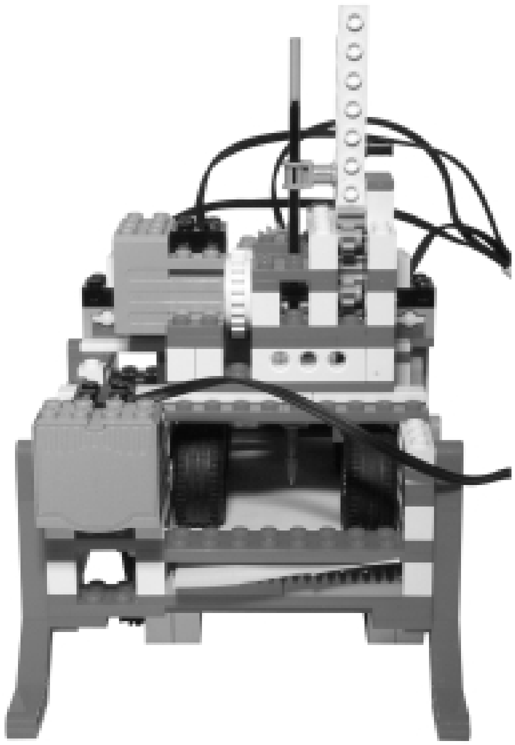
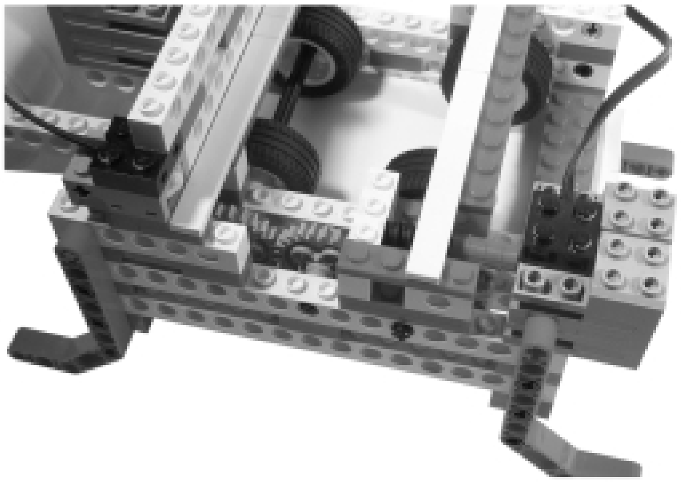
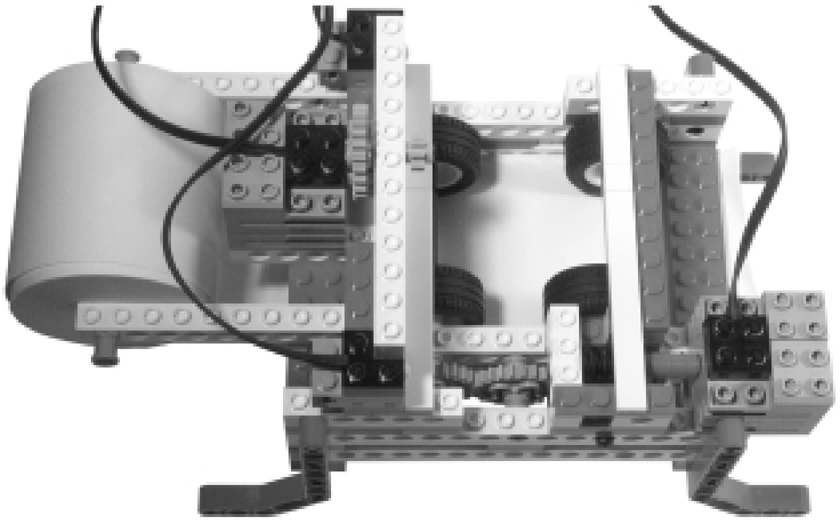


Figure 23.12 Writer Rear View



Removing the pen carriage, you see the wheels and the paper tape down below (Figure 23.13). The carriage is translated using a rack and pinion assembly, powered by a second motor on the body. We used two touch sensors to detect the carriage limits, but you could just as well employ a single sensor with two closing pegs, as we did for many other projects in this book.

Figure 23.13 Writer Top View, Pen Carriage Removed



A second rack and pinion system, operated by the third motor, controls the vertical movement of the pen (Figure 23.14).

We tried different ways to attach the pen, until we discovered that many common and cheap pen refills have a diameter close to that of the LEGO flex tubing. This simplified our lives a lot (Figure 23.15). Make sure you don't damage the refill, otherwise you'll end up washing a few LEGO parts!

The top is completely covered with tiles (Figure 23.16). The irregular surface covered with studs wouldn't work. In case you don't have tiles, or not enough of them, cover the plates with a smooth, thin support, like a glossy cardboard, an aluminum or plastic sheet or anything else similar that comes to mind. You can also build a top out of standard LEGO bricks laid on their side, which should provide an even more regular surface than tiles.

The writing surface is an independent part linked to the main body through short rubber bands (Figure 23.17). Those bands pull the surface up against the pen and against the wheels of the feeding mechanism.

Figure 23.14 The Writer's Pen Assembly

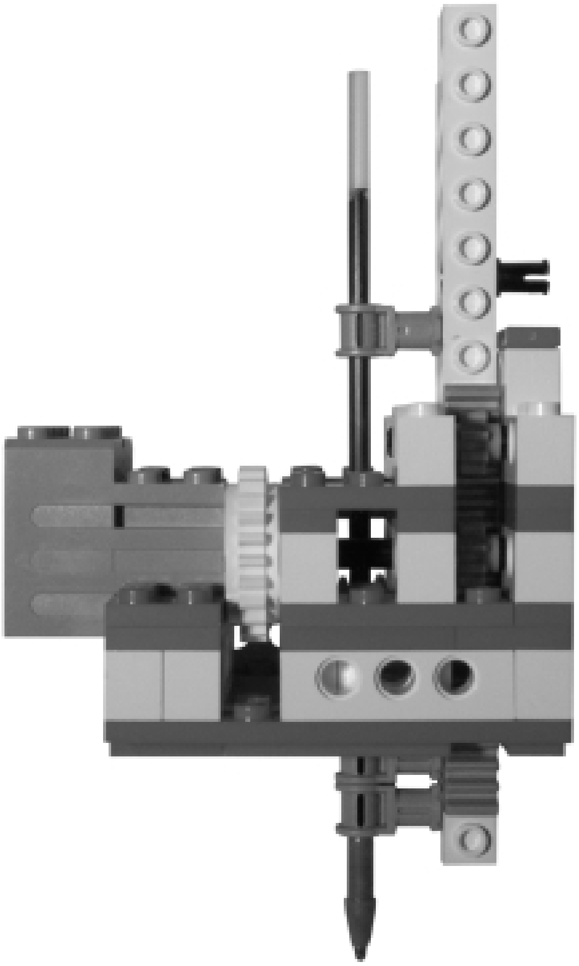


Figure 23.15 Close-Up of the Pen

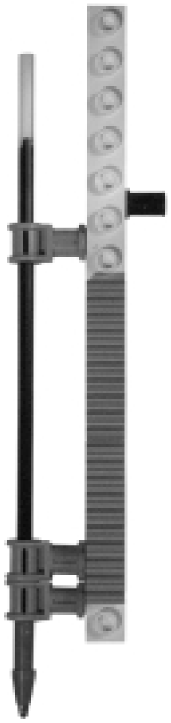


Figure 23.16 The Writer's Writing Top Taken Apart

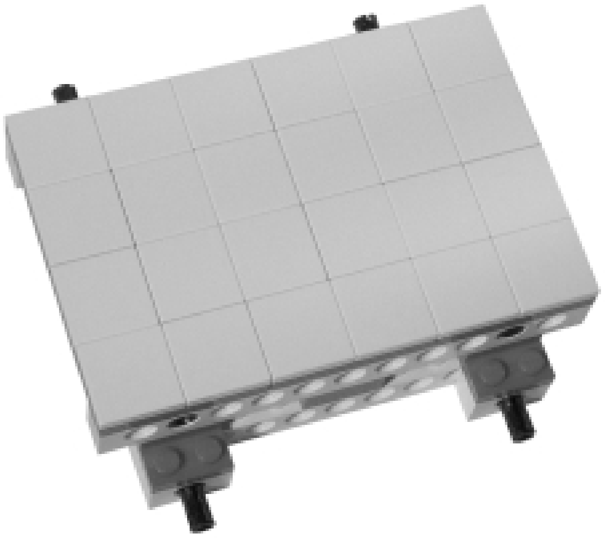
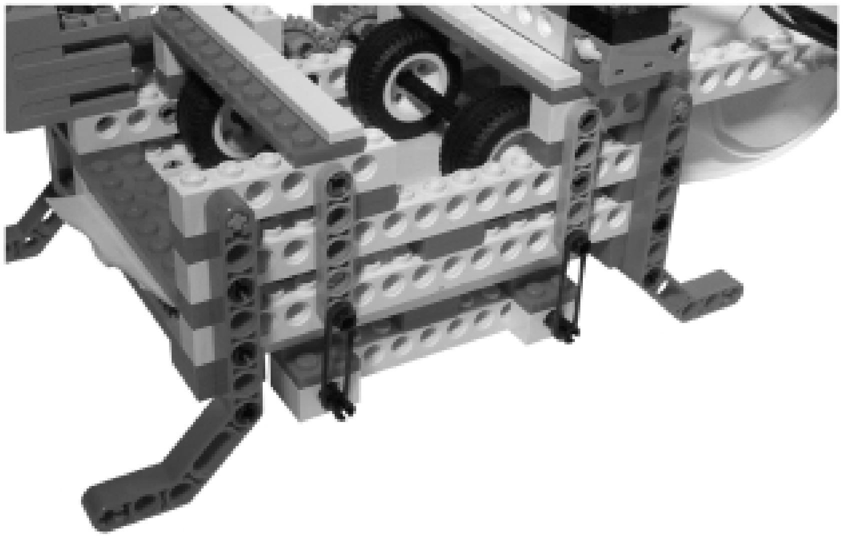


Figure 23.17 Writer Front View



Bricks & Chips…

Washing LEGO Parts

LEGO bricks are, generally speaking, not very difficult to clean. You can remove small ink spots using a cotton ball soaked in some alcohol. For large-scale cleaning, hand-wash your bricks in a tub of warm water with some dish detergent. Machine washing in a clothes washer on a warm setting is possible too, provided that you put your parts in a canvas bag.

Some warnings apply:

- Do not use any solvent, unless you're sure of what you are doing. Test it on a brick you don't care about.
- Transparent and printed bricks should not be cleaned with alcohol or any other solvent. They should only be washed by hand.
- Never use hot water, and never put your bricks in a *dish-washer* because the settings will be too hot and will warp the parts.

Programming the Writer

This robot is not very difficult to program, provided you choose a simple font, composed mainly of straight horizontal and vertical lines. Our own version contains few general use subroutines that draw standard length lines: Full length vertical lines get drawn using the two touch sensors as a reference, while half length vertical positioning is based on timing (always starting from the bottom). All the horizontal movements are based on time, but we have seen that for this application this isn't necessarily a critical limitation, especially if you optimize the path the pen follows when drawing the characters. Four more subroutines control the pen-up/pen-down positions, as well as the spacing between letters and between words.

Using those subroutines, you can now code all the characters you want to print in terms of pen movements. A tedious task, but not difficult. Study the path that minimizes the number of moves, and keep the pen up when you have to pass over an already traced line: It's almost impossible to get the pen to cover the same line exactly a second time, and a double line only makes things look worse.

Now you have all the elements to start writing. Simply call the proper character subroutines in sequence and you can write what you want. The two main options are: Coding some predefined or random behavior in the program to make it write some text, or use a communication protocol to interact with the system. As we said for the Logo Turtle, pbForth is probably your best choice for an interactive system, but this particular case is appropriate for the simple standard messaging scheme, too, being that you can send any single-bit ASCII character in a message.

What to Write

We had the idea of making this robot an Automatic Haiku Writer, but the truth is that you can make it write whatever you want. In the last part of the chapter, we will give you some hints about other possible uses of this robot: a label-writing machine, a graphing system, and more.

Now, what's a *haiku*? It's an ancient Japanese poem with a formal structure. Though not everybody agrees on all the rules involved, the most accepted form is a three-line verse where each line is composed respectively of five, seven, and five syllables. It usually contains a reference (even indirectly) to time, and is broken into two parts, like an introduction and a theme, or an action and its consequence. Here's our own example of a haiku contemplating the theme of this book (we ask your forgiveness in advance!):

The robot is on
Feel a bit worried, about
The things it could do

You can program your robot to produce a written haiku on request, generating it randomly from a database of predefined sentences. Or, using a more sophisticated approach, pluck random words from a (small) inner dictionary, combining them according to simple predefined grammatical structures (see Appendix A for some links to useful Internet resources in this regard).

Variations

On the technical side, you can make your robot more accurate by adding a rotation sensor to control the feed of the paper. Meanwhile, the two touch sensors for the carriage, as explained before, could become one.

It's possible to make a version of the writer fully independent from the PC, but still interactive. A sort of old fashioned label-writing machine where you choose a single letter at the time then print it. Add a rotation sensor with a wheel to make the user alternate the possible characters, and a touch sensor to confirm printing. You should show the characters on the display while the user rotates the wheel, but this is possible only if you installed an alternative firmware that allows full control of the display, including some alphanumeric capabilities (e.g., legOS). With the standard firmware, you should stick with numbers and show a code for each character.

A more radical variation on this project is the making of a graphing machine, like those that produce electrocardiograms or electroencephalograms, or monitor weather parameters like temperature or humidity. In this case, things get more simple on the mechanical side, because you don't need the pen mechanism anymore (the pen will be always down) and a single paper feeding direction is enough (the paper always goes forward). What you should add is a rotation sensor to make your software capable of converting the input value into a precise position. In typical applications, the paper will advance and the chart will update only once in a while, the length of this interval being related to the rate of change of the parameter you're monitoring.

What should you graph with such a system? Whatever variable your sensor can measure: temperature and light, for example. Using some simple assemblies or a few custom sensors, you can measure sound intensity, voltage, distance, force, weight, wind speed, or whatever else you can convert into a signal for an input port.

Further Suggestions

The writing and drawing theme offers many other ideas. The following suggestions are far from being exhaustive. Consider them starting points for your own creations.

Copying

This has become an almost classic project, but it's still interesting, instructive, and challenging. You need a feeding mechanism similar to those of our Tape Writer, but duplicated for two pieces of paper. It must be able to drag two sheets of paper: the one being copied and the blank one. Obviously, the whole machine will be much larger than the Tape Writer if you plan to use standard letter or A4 sheets.

The copying system is made of a translating assembly that moves a light sensor back and forth across the original sheet, and a pen in the corresponding position over the copy sheet. With the paper feeding motor stopped, the software scans a row with the light sensor, and depending on what intensity it detects, it puts the pen up or down. After each row, the paper will feed a bit for the next scan.

The requirements for this project are not very high: three motors, one light sensor, and one or two touch sensors for the carriage movements; but we suspect that in creating a standard sheet copier, you will likely need many additional beams and plates, because the structure will be rather large.

Emulating Handwriting

Using a completely different technique, you can build a robotic arm that writes with movements similar to those of a human arm. You need an arm with two degrees of freedom, similar to the one described in Chapter 20 when we discussed moving chess pieces: Two levers move on a horizontal plane, the first attached to the body of the robot and the second to the end of the first. At the end of the second lever, there's the pen with its lifting mechanism. We suggest you keep it very lightweight, using either pneumatics, the flex system, or a micromotor. Every lever will be rotated by a motor and controlled with a rotation sensor.

The software to control this beast is not very simple. Converting the angles of the arm into Cartesian coordinates on the sheet requires some trigonometry, and all that that implies (see Chapter 13)!

Learning by Example

To make the previous project even more interesting, and at the same time get rid of all the trig (yeah!), you can design your robot to learn from your movements. In this case, your robot will have a training phase, where you guide its arm to write or draw what you want, and a production phase where it replicates your movements.

Make the motors easy to decouple, so you don't have to move them while driving the arm during the training phase. The program will save the sensor readings at small intervals in order to reproduce those positions later in the production phase.

For the pen up/down movements, you can keep the motor connected and controlled by a touch sensor that you press when you want to flip from one state to the other.

The most challenging part of this project is the storing of the data collected during the learning process. You have basically two options: using a language that allows large memory structures like arrays (legOS, leJOS, pbForth), or doing the dirty work on the PC, leaving all the "intelligence" and data there and using the RCX as merely an executor.

Summary

In this chapter, we explored some techniques described in Part I that had not yet been applied to robots in this book. The Logo Turtle offers a good opportunity to find a use for the sophisticated dual differential drive of Chapter 8, which is capable of turning in place like a simple differential drive, but also of going perfectly straight. In fact, at the price of some mechanical complexity, it provides a way to separate straightforward motion and turning capabilities using two independent motors. Its advantages include the fact that you can monitor both kinds of movement with a single rotation sensor attached to one of the wheels. Using the dead reckoning math of Chapter 13, you can precisely control your Turtle. We went through those equations again, providing a concrete example of how to implement them in a NQC program.

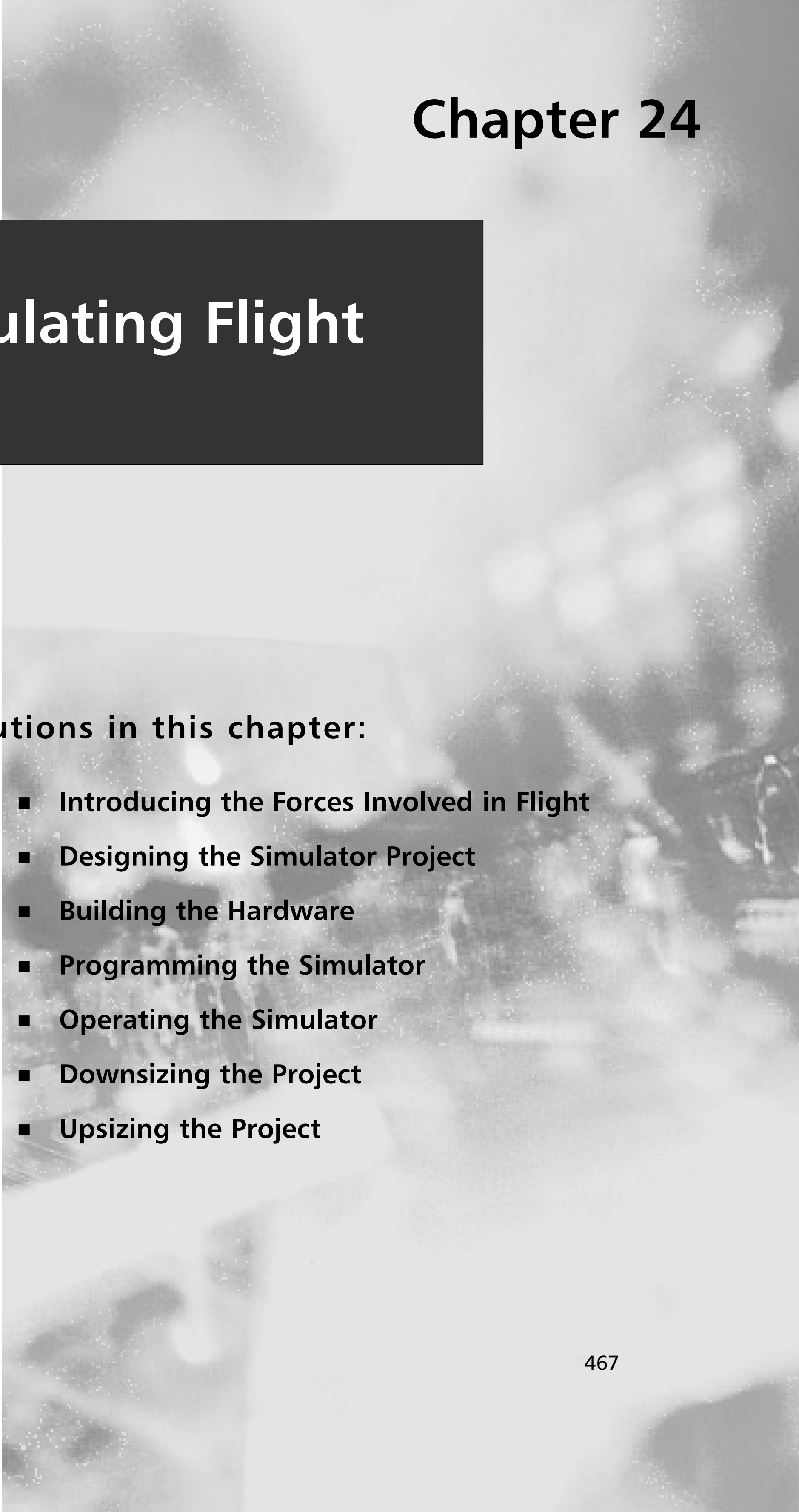
You can program your Turtle robot with any language of your choice; however, we discussed the advantages that this particular project could derive from an interactive programming environment like pbForth.

Though conceptually simpler, even the Tape Writer showed some construction tips. It is a Cartesian system not too different from those used in the robots of previous chapters (the Maze Solver and the Tic-Tac-Toe machine), but it does demonstrate once more that by reevaluating the terms of a problem, you can find an easier solution. For example, a Tape Writer built with a technique similar to the Maze Solver would have required very long rails; so, moving the paper instead of the robot, your construction results in a more compact design that is also capable of writing texts of unlimited length.

In the suggestions we provided at the end of the chapter, we described the possibility of emulating handwriting using an arm similar to the one used in the Broad Blue robot described in Chapter 20—not necessarily the same size, but based on the same principle. This includes a glimpse at how robots can learn by example, too; a feature used in many real life robots, including industrial robots. In a case where you want your robot to perform handwriting, you can guide the movements of the robotic arm to copy the shape of any written character; the robot "remembers" your movements, and then is able to replicate them and write by itself.

Chapter 24

Simulating Flight



Solutions in this chapter:

- **Introducing the Forces Involved in Flight**
- **Designing the Simulator Project**
- **Building the Hardware**
- **Programming the Simulator**
- **Operating the Simulator**
- **Downsizing the Project**
- **Upsizing the Project**

Introduction

LEGO robots cannot fly. This is a fact you have to accept—there's no way to build any kind of pure-LEGO self-powered flying machine. The project described in this chapter is a rather ambitious variation: We want to *simulate* flight.

What we're going to describe here is not technically a robot; we use the processing power of the RCX to simulate the simplified behavior of an aircraft. Even if you're not knowledgeable about flight or flight simulation, this project deserves some attention for its intensive use of the resources provided by the RCX: the in and out ports, the display, and the loudspeaker.

The simulator, as we conceived it, requires a lot of additional parts: three rotation sensors, four motors, some long cables, two polarity switches, and some extra beams and plates. Don't worry if you don't have all those parts, we also suggest some ways to reduce the requirements of the project.

Introducing the Forces Involved in Flight

Imagine an airplane in flight. The line that passes through the fuselage is called the *longitudinal axis*. When the plane rotates around this axis, one wing goes down and the other up. This movement is called *bank*, or *roll*, and in a real plane is controlled by the *ailerons*. The pilot operates the ailerons through the control yoke (see Figure 24.1). A movement of the yoke to the right banks the plane right (right wing down), and vice versa.

The same yoke also controls the *elevators*, the moving parts of the *horizontal stabilizer* (the "rear wings" of the plane). The elevators control the rotation of the plane around its *lateral axis*, a line parallel to the wings, and this affects the *pitch* of the plane (Figure 24.2). When you pull the yoke backward, the nose of the plane goes up, while pushing it forward points the nose down.

The *vertical axis* is perpendicular to the first two, and is essentially a vertical line that passes through the center of gravity (COG) of the plane. To rotate the plane around this axis, the pilot uses the *rudder pedals*. They control the *rudder* which is located on the *vertical stabilizer* of the plane. This rotation is called *yaw* (Figure 24.3).

Figure 24.1 The Ailerons Control Bank

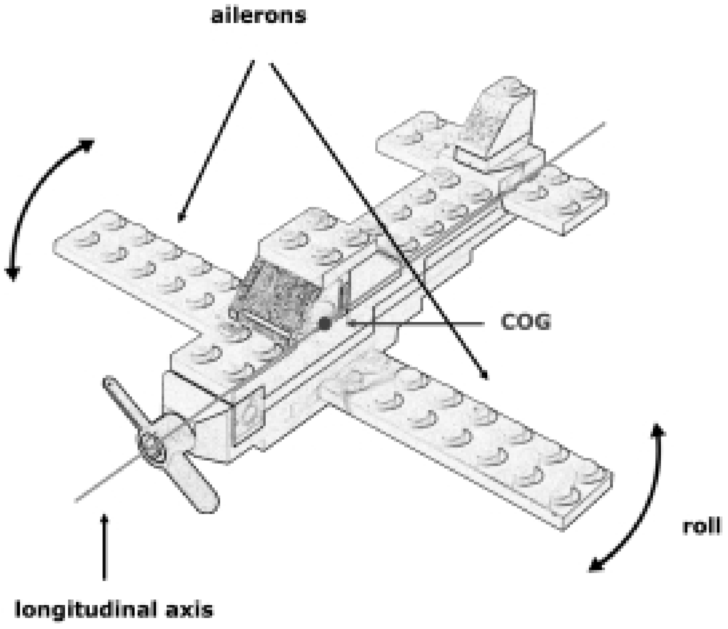


Figure 24.2 The Elevators Control Pitch

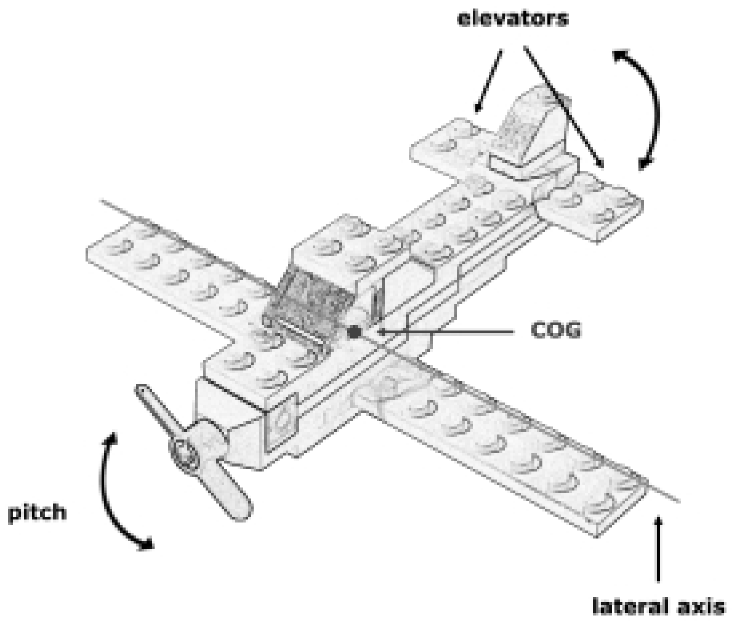
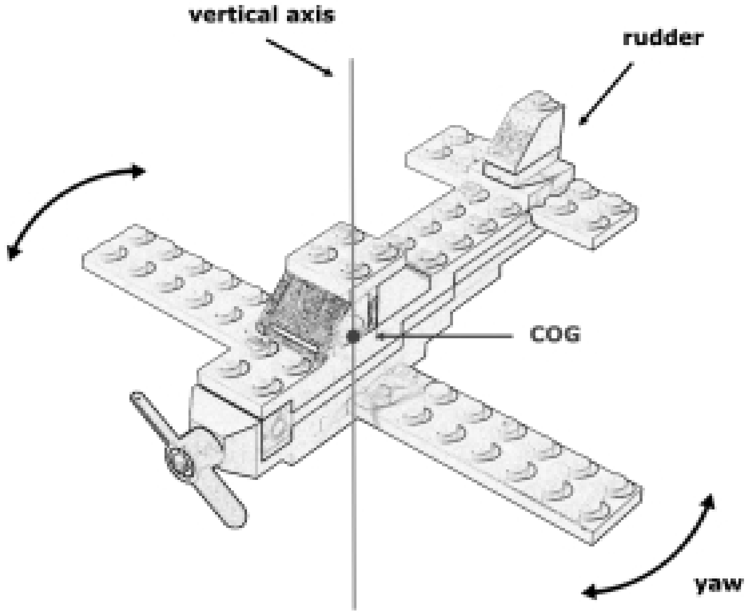


Figure 24.3 The Rudder Controls Yaw

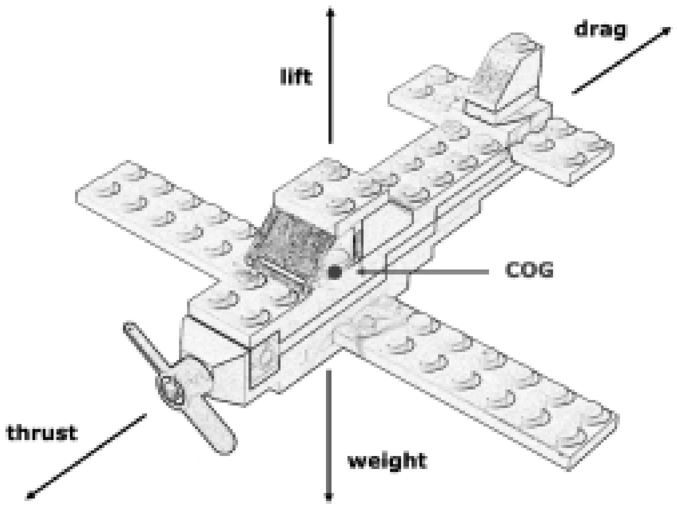


The fourth and last of the basic controls in a plane is the *throttle*, used to apply power. The engine of the plane converts this power into *thrust*, part of which affects the *speed* of the plane through the air, while another part gets transformed into *lift* by the wings. Lift is the force that opposes the weight of the plane and allows it to overcome the force of gravity. Another force, *drag*, caused by the friction of the air, increases rapidly with the increase in speed, thus limiting the maximum speed that the plane can reach (Figure 24.4).

There are other important controls in a plane, like the *flaps*, movable surfaces on the wings used to increase lift (and drag), or the *elevator trim*, but we're going to ignore them in our simplified simulator.

The behavior of a plane results from a complex interaction of its controls and the forces that come into play. We already explained that the thrust produced by the engine affects the speed of the plane and its vertical velocity. The pitch of the plane has an influence on this, too: Starting from straight and level flight, if you pull the yoke back, the nose will go up, causing an increase in lift and drag; the plane will reduce its air speed. and increase its climb rate. Eventually the plane will stall and fall out of the sky like a rock.

Figure 24.4 Drag Balances Thrust, while Lift Balances Weight



Turns are managed through bank and yaw. In normal turning, they must be coordinated. In a right turn, for example, the pilot applies the ailerons and rudder to make the plane bank right and turn right. The bank is necessary to compensate the *centrifugal force* introduced by turning, and the pilot banks the wings to make the lift oppose the resulting gravitational and centrifugal forces. In real planes, this coordination is very important for safety. This is why high-speed race tracks have banked turns. The resulting forces on the car and driver are directed downwards, which helps keep the car on the track and away from the wall.

Piloting an aircraft is very different from driving a car. Turning a car, for example, you keep the steering wheel turned throughout the length of the curve. In a plane, by contrast, you apply some yoke and rudder only until you get a new attitude and the plane starts turning, then center them again, similar to keeping a boat on course. The plane will continue in its turn until you operate the yoke and rudder again, in the opposite direction, to resume straight flight.

Designing the Simulator Project

Our flight simulator is made of two components: a mobile platform and a remote control. The idea is that from your remote control, a sort of portable cockpit, you manage three inputs of the system—pitch, bank, and throttle—and read indicators of speed, altitude, and possibly other parameters. We didn't succeed in having a further control for the rudder, but this doesn't affect the simulation too much since we assumed that bank and yaw are always coordinated.

The mobile platform features a symbolic airplane that represents the one you're flying. This plane actually rolls and pitches, giving you a visual indication of its attitude. Meanwhile, the platform moves and turns according to its speed and direction. As the remote control is connected with cables to the platform, you'll have to walk behind it while it moves. It's more or less like driving a remote-controlled car, the difference being that the vehicle here responds like an aircraft.

Let's explore the design details. Two motors on the platform control pitch and bank; these are not driven by the RCX, but rather by you through the yoke on the remote. The yoke works with two polarity switches. What the RCX reads are the values of pitch and bank that come from two rotation sensors connected to those motors. The third input, throttle, arrives from a third rotation sensor placed on the remote.

From these inputs, the RCX computes the outputs: thrust, lift, drag, acceleration, speed, heading, and altitude of the plane. The speed is reflected in the motion of the platform, and the same applies to the heading: When the simulated plane turns, the platform turns, too. The RCX drives the platform, a simple differential drive, with two of its output ports. The altitude appears on the display, and you can program the simulator to show other values, too. In our version, altitude and speed appear at the same time using two digits each. The loudspeaker outputs the noise of the engine, and the stall alarm. (*Stall* is when, because of too low a speed or too high a pitch, the airflow detaches from the wings causing a sudden drop in lift.) Functions and connections are summarized in Table 24.1.

Table 24.1 RCX Resources and Functions

RCX Resource	Function
IN 1	Reads Pitch.
IN 2	Reads Bank.
IN 3	Reads Throttle.
OUT A	Controls the left motor of the differential drive platform.
OUT B	Controls the right motor of the differential drive platform. Outs A & B together reflect velocity and direction of the plane.
OUT C	Always ON; used just to supply power to the polarity switches that control pitch and bank motors.

Continued

Table 24.1 Continued

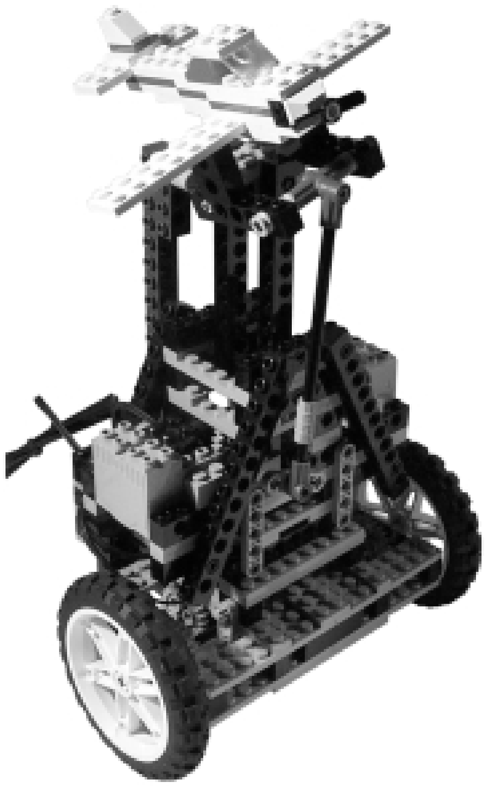
RCX Resource	Function
Display	Shows altitude, speed, and possibly other flight status variables.
Loudspeaker	Reproduces the simulated noise of the engine and outputs the stall alarm.

Building the Hardware

Figure 24.5 shows a view of the whole mobile platform. It's organized into three sections:

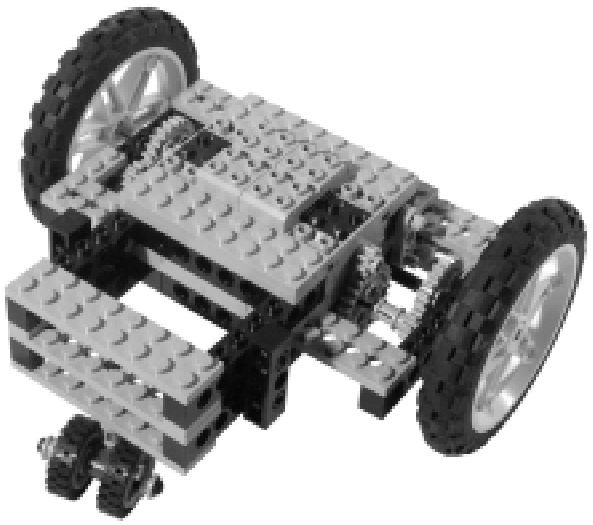
- At the bottom, a differential drive.
- In the middle, the assembly that controls pitch and bank.
- At the top, the small plane that is your visual reference for flight.

Figure 24.5 The Mobile Platform



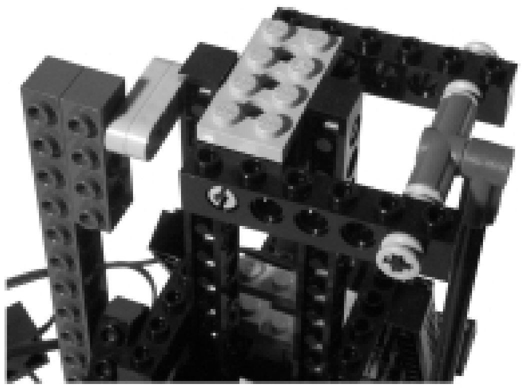
Let's dissect the platform and analyze its components. This particular differential drive (Figure 24.6) is really nothing unusual. We used a high reduction ratio (1:9) to keep the simulator slow. This design is actually very similar to that of chapter 14, with the only relevant difference being that the motors are placed behind the main wheels to keep the COG between them and the pivoting wheel.

Figure 24.6 The Differential Drive that Controls Motion



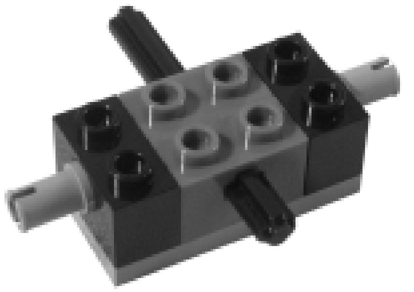
The most notable part of the pitch and bank control assembly is the small chassis at the top, the one that supports the plane (Figure 24.7). This is able to move with two degrees of freedom (DOFs) in two orthogonal, or perpendicular, directions that correspond to the lateral and longitudinal axis of the plane. In other words, this mechanism controls the pitch and bank of the plane independently.

Figure 24.7 The Pitch and Bank Chassis



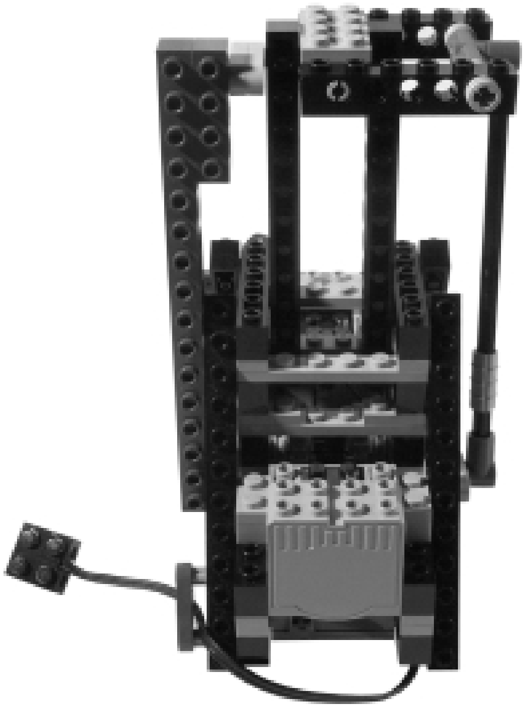
Under the 2 x 4 plate in the middle are two 1 x 2 beams with axle holes, intersected by the axle that controls bank. The other two 1 x 2 bricks hold gray pins that represent the pivoting points of the lateral axis, the one that controls pitch (Figure 24.8). The center of this assembly is the virtual COG of the plane.

Figure 24.8 The COG of the Plane



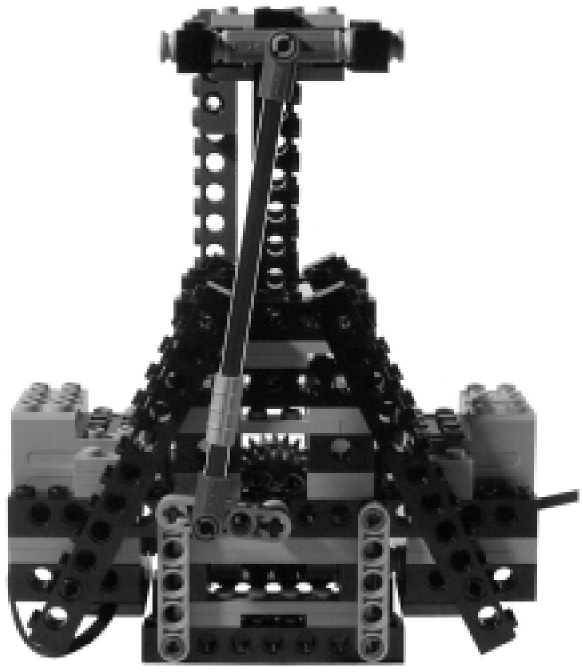
This jointed chassis connects to two arms that operate the pitch and bank movements. The front one adjusts pitch, while the rear one changes bank (Figure 24.9).

Figure 24.9 Side View



Those long arms are linked to liftarms, each one mounted on an axle that goes right in the middle of the assembly and ends on a 24t gear (Figures 24.10 and 24.11). The two diagonal beams that lock the assembly are not a perfect match: They form a triangle with sides 4 and 8, which correspond to a hypotenuse of about 8.94 instead of the 9 we used. This small difference—less then seven parts per thousand—is close enough for a solid connection.

Figure 24.10 Front View



Going inside the assembly, you'll notice it's perfectly symmetrical. The two 24t gears are moved through worm gears. The axles that carry the worms also carry pulleys to receive motion from the motors, and pass through the rotation sensors that measure pitch and bank (Figure 24.12).

The mechanism is easier to see in the cross-section of Figure 24.13, which shows just one side of the assembly that controls pitch and bank. The motor shafts mount half bushings used as pulleys to get the highest reduction ratio against the medium pulley.

Figure 24.11 Rear View

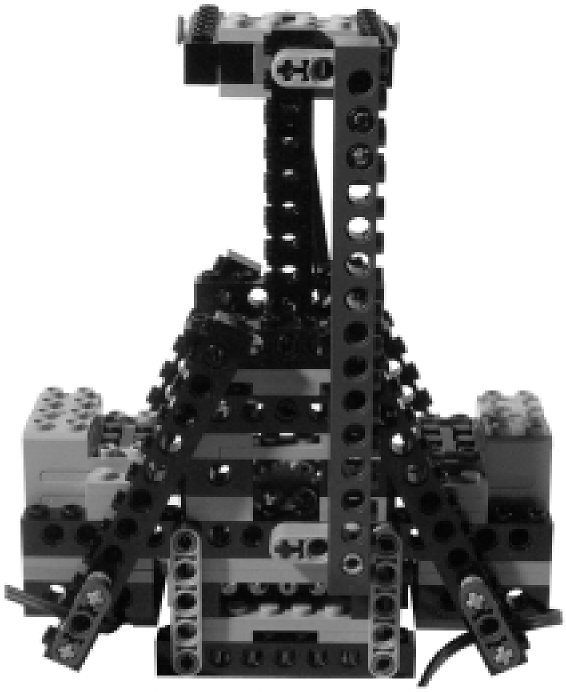


Figure 24.12 Bottom View

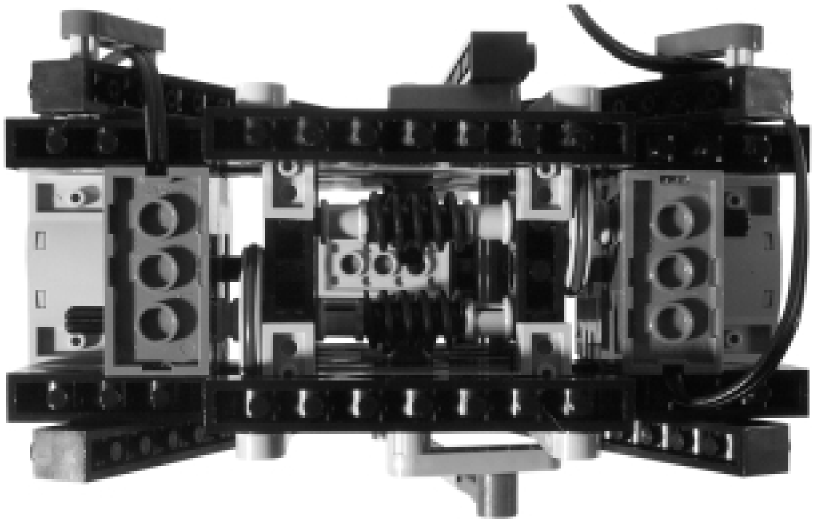
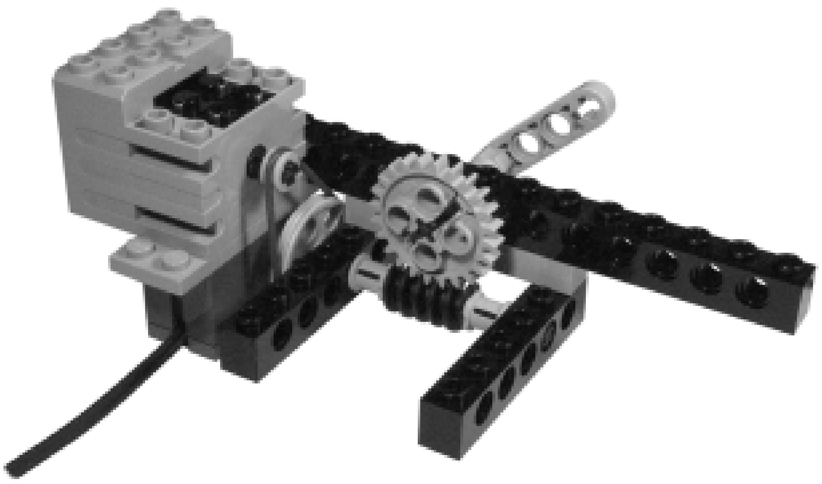
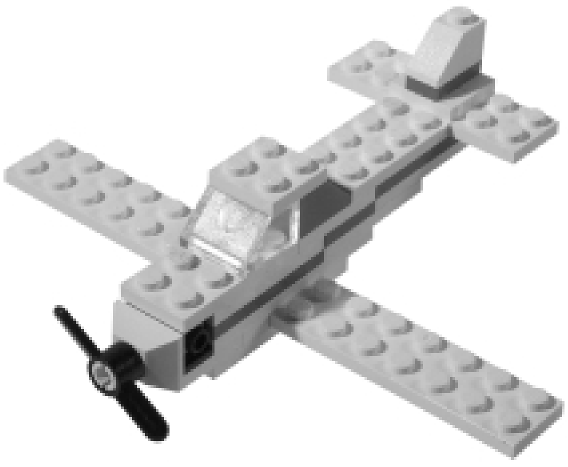


Figure 24.13 Cross-Section of the Pitch/Bank Mechanism



The model of the plane itself is representational (see Figure 24.14). Build any plane of your choice with the parts you have available, just don't make it too heavy because this design is not suitable to support large loads. If you want to use a larger model, put your plane's real COG as close as possible to the two DOF joint. (It's much better below it than above it.)

Figure 24.14 The Plane



Our remote control contains the RCX, the yoke, and the throttle (Figures 24.15 and 24.16). The yoke is made with two polarity switches: The central one (bank) is mounted over two 1 x 2 bricks with axle holes, whose axle goes into the second switch (pitch) on the left of the picture. The throttle is a simple wheel and axle assembly that passes through a rotation sensor.

Figure 24.15 The Remote Control

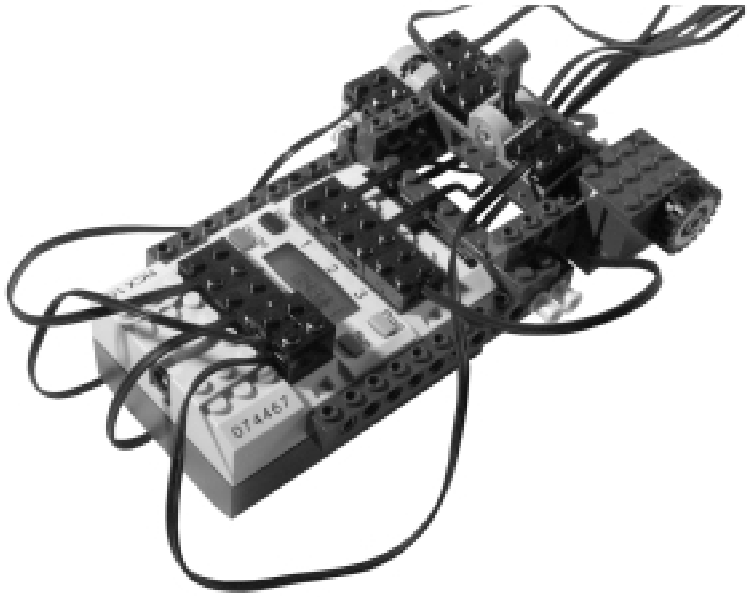
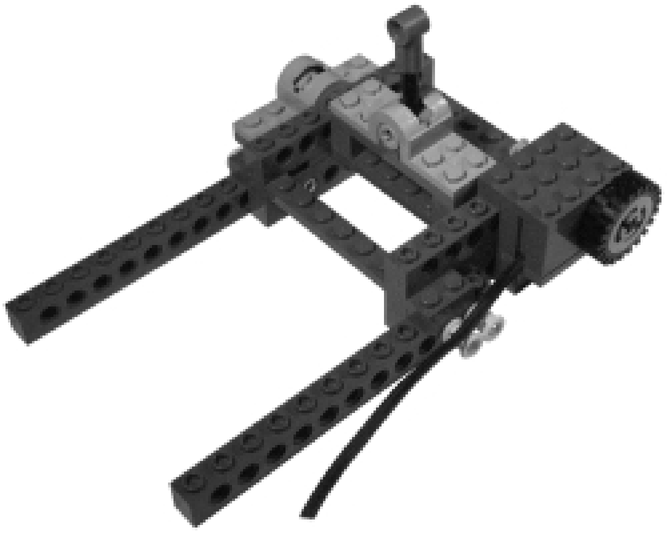


Figure 24.16 The Yoke and Throttle Controls



There are six long wires that run from the remote to the platform: two from output ports A and B to the motors of the differential drive, two from input ports 1 and 2 to the rotation sensors for pitch and bank, and two from the polarity switches to the pitch and bank motors. The throttle rotation sensor connects directly to input port 3, and you need two more short cables to bring power from output port C to the polarity switches.

Programming the Simulator

This is the hardest part of the job. There's an impressive quantity of material in literature and on the internet about the physical equations that explain flight, but:

- They are not easy to understand if you don't have a solid background in physics and math.
- They are not easy to implement on your RCX unless you use some alternative programming environment that allows high precision math and trigonometry functions.
- Making a good simulator, realistic enough but enjoyable at the same time, is something that goes beyond the understanding of the principles behind flight. The process requires some experience and a lot of patience in testing all the details.

We developed a simple model that, though largely simplified, has the advantage of requiring very simple math and working with the limited 16-bit precision allowed by the standard firmware. Our NQC version works quite well and makes the simulator fun and instructive to use with less than 200 lines of code. Using legOS, leJOS, or pbForth, you can get even more from your simulator. (We'll give you some hints about this later in the Upsizing the Project section.)

The program starts by configuring the sensors and resetting the variables. Afterward, the main cycle begins:

```
while(true)
{
  ReadInputs();
  ComputeOutputs();
  UpdateDisplay();
  Wait(INTERVAL);
}
```

You see the structure is quite simple. The program reads the inputs, the three rotation sensors, then computes the variables that represent the output of the system (altitude, speed, and direction), and finally updates the display of the RCX. The conversion of speed and direction into motion of the platform is performed by a separate task, but we'll discuss this later.

INTERVAL is a constant that reflects the simulation step. We modeled our equation on an interval of 1 second, meaning that the model is realistic (as much as it can be) when the status is updated once a second. As all the computations require some time, *INTERVAL* will be something less than 1s in order to make the cycle last almost exactly one second. We placed a sound click in the loop and trimmed the value until we found one that made the RCX click exactly 60 times a minute. In our case, it was 85 hundredths—which implies that the processor inside the RCX was actually only doing work for 15 hundredths of a second!

The **ReadInputs** subroutine polls the sensors and converts the readings into proper values for pitch, bank, and throttle. Pitch and bank are expressed in degrees, 0 represents level flight, while positive values mean nose up for pitch and right wing down for bank. The bank control is built with a 24t driven by a worm gear, the latter attached to the rotation sensor. This means that the 24t makes a full turn, 360°, every 24 turns of the worm. As the rotation sensor ticks 16 times every turn, it will count 24 x 16 = 384, so the bank angle will be the sensor reading multiplied by 360/384 (reduced, this becomes 15/16). We are now ready to compute the *bank* variable:

```
bank=(SENSOR_BANK*15)/16;
```

Notice the use of parentheses: When you are not sure how your compiler optimizes expressions, use parentheses to be sure the computation follows a specific sequence (see Chapter 12). Notice also that we called the sensor `SENSOR_BANK` instead of, for example, SENSOR_2. This is not only possible, but very helpful in making your code self-explanatory. You simply need to define a new constant whose value is the name of the sensor port you want to map with a new name:

```
#define SENSOR_BANK SENSOR_2
```

The pitch control assembly is slightly different: It doesn't actually rotate the lateral axis but rather acts on a lever. To convert this movement into an angle, you should use trig functions. But relying on the fact that useful pitches won't go over +/-16°, and that in that range the behavior of our assembly is almost linear, we introduce a small simplification and use a linear conversion for this case, too.

The important thing to remember is that the arm of the lever at the top is double the length of that at the bottom, meaning that in respect to the bank sensor, you need double the ticks from the rotation to cover the same angle. This leads to the formula:

```
pitch=(SENSOR_PITCH*15)/32;
```

We chose for the throttle a scale of ten values from 0 to 9. To make the rotary control a bit less sensitive, we counted an increment in throttle every four ticks of the sensors, and added some code to ensure its value stays in the right range.

```
temp_thr=throttle+(SENSOR_THROTTLE-old_sensor_throttle)/4;
old_sensor_throttle=SENSOR_THROTTLE;
if (temp_thr<0)
  throttle=0;
else if (temp_thr>9)
  throttle=9;
else
  throttle=temp_thr;
```

Now we have all the elements to determine the new flight status variable, which derives from the previous status combined with the effect of the input controls.

NOTE

We use the metric system for all the variables in the Flight Simulator, thus altitude is expressed in meters, speed in m/s, and acceleration in m/s^2. Nothing prevents you from converting the final output values to feet and knots, however. 1m corresponds to 3.28 feet, and 1m/s to 1.94 knots.

Let's start with acceleration, which is essentially the variation in speed. It comes out of three components: applied power (throttle), drag, and pitch. We used a simple linear relation for throttle:

$\text{acceleration}_1 = \text{throttle} / 2$

Drag is the force that contrasts the applied power, and it increases with the square of the speed. Therefore, our formula is:

$\text{acceleration}_2 = \text{speed}^2 / 1250$

Pitch also affects speed: the higher the nose of the plane, the greater the portion of power that is transformed into lift instead of speed:

$\text{acceleration}_3 = \text{pitch} / 5$

The final equation is:

$\text{acceleration} = \text{throttle} / 2 - \text{speed}^2 / 1250 - \text{pitch} / 5$

The maximum speed that our plane can reach during level flight (pitch = 0) is 75m/s, or about 145 knots, a typical value for a small propeller aircraft. At that speed, drag equals the thrust produced by the engine at maximum throttle.

Let's create an example using real numbers. Suppose you are applying a throttle of 8, and that your airplane is flying at 65m/s with a positive pitch of 3 degrees. Acceleration then becomes:

$\text{acceleration} = 8 / 2 - 70 \times 70 / 1250 - 3 / 5 = 4 - 3.92 - 0.6 = -0.52$

If you want to simulate what happens inside your RCX, remember that you are limited to whole integers, thus you should calculate the expression truncating the results of every division to an integer number:

$\text{acceleration} = 8 / 2 - 70 \times 70 / 1250 - 3 / 5 = 4 - 3 - 0 = 1$

The difference between this (wrong) result and the previous (correct) one is quite significant: What should have been a negative acceleration—the plane slows down—becomes a positive one—the plane speeds up! To keep this problem to a minimum, recall some of the suggestions given in Chapter 12, and rearrange the expression to get as small a loss in precision as possible. For example, you can group the numbers so you have only one final division:

$\text{acceleration} = (\text{throttle} \times 625 - \text{speed}^2 - \text{pitch} \times 250) / 1250$

Using the numbers of our example, and truncating the result of the division, this expression becomes:

$\text{acceleration} = (8 \times 625 - 70 \times 70 - 3 \times 250) / 1250 = -650 / 1250 = 0$

The new result, 0, is better then the previous one, that is, it's closer to being correct. A further improvement comes from the rounding of the last digit (rounding 5 into 10). This is actually the solution we adopted in our NQC code:

```
speed2=speed*speed;
```

```
acceleration=throttle*625;  // thrust
acceleration-=speed2;        // minus drag
acceleration-=pitch*250;     // minus vertical component
acceleration/=125;           // scale appropriately
if (acceleration>0)          // round +/- 0.5
  acceleration+=5;
else
  acceleration-=5;
acceleration/=10;
```

How does this trick work? Let's again use the numbers of our example. Dividing by 125 instead of 1250, the result is:

$-650 / 125 = -5.2$

As acceleration is negative. The code performs the rounding by subtracting 5, then dividing by 10 again:

$\text{acceleration} = (-5.2 - 5) / 10 = -1$

This is the best approximation you can get. The sign of the acceleration is the correct one, and its magnitude is rounded to the nearest integer.

Now we'll figure out the constants for lift, the force produced by the flow of air over the wings. Lift is somewhat similar to drag: They increase together at a similar rate. The pitch of the airplane affects the lift, too: the higher the pitch, the greater the lift. This is the equation we introduced in the model:

$\text{lift} = \text{speed}^2 \times (1 / 422 + \text{pitch} / 3500)$

Pitch is not the main way to control lift, and cannot be increased to arbitrary values for at least two reasons. The first is that an increase in pitch reduces speed, thus limiting the generated lift. The maximum climb speed in our simulator results with maximum throttle and a pitch of 10°, though temporary higher values are possible when increasing pitch, until the speed drops down at a stable level.

The second reason is that there's a physical limit to the pitch the plane can tolerate before stalling: When the pitch passes a critical value, the airflow detaches from the wings and the aircraft experiences a sudden drop in lift that goes to zero. Stall can occur for another reason: too low a speed. For our model, we chose a maximum pitch of 16° and a stall speed of 27m/s (52 knots).

There's another negative component to be considered when computing lift: bank. We use bank as an absolute value since any bank other than zero reduces the climb rate of the airplane because part of the force is used to compensate the centrifugal force. Our final equations becomes:

lift = $speed^2$ x (1 / 422 + pitch / 3500 – |bank| / 35000)

You see that bank has a negative effect on lift equal to one tenth of pitch. Thus one degree in pitch compensates the loss in lift produced by ten degrees in bank. The following code includes the test for the stall condition. The computation has been again rearranged to maximize the precision:

```
if (pitch>16 || speed<27)   // stall!
{
  lift=0;
  stall=1;
}
else
{
  lift=speed2 / 422;                               // effect of speed
  lift+=((speed2 / 10) * pitch) / 350;         // effect of elevators
  lift-=((speed2 / 10) * abs(bank)) / 3500;  // effect of bank
  stall=0;
  flying=1;
}
```

The flying flag, set to 0 at the beginning of the simulation, makes the simulator remember that the lift value became positive at some time, in order to activate the stall alarm if the lift becomes zero again after the takeoff.

There's just one thing missing in the model: the effect of bank. We used a very simple relation to obtain the change in heading (angular velocity in degrees per second) from bank and speed:

```
turn=(bank * speed) / 453;
```

Now that you have all the elements that affect your flight attitude in the previous time interval or step in the simulation, you can proceed to update the corresponding status variables. Notice that altitude cannot be less then zero. We didn't include any landing or crash test. We leave this exercise to you.

```
altitude+=lift-g;
if (altitude<0)
  altitude=0;
```

```
speed+=acceleration;
heading+=turn;
if (heading>360)
  heading-=360;
else if (heading<0)
  heading+=360;
```

The constant g introduces the effect of gravity. The real value should be 9.8m/s^2, but in our world of whole integers it becomes 10m/s^2. Remember to declare it somewhere in your code:

```
#define g 10
```

You are ready to display some of the calculated values. If you're using the standard firmware, remember that you need version 3.28 or later to control the display (it's free at the LEGO site and perfectly compatible with RCX 1.0 and 1.5).

For our version, we chose to split the display into two groups of two digits, the first for altitude, in tens of meters, and the second for speed, in m/s:

```
tmp_display=(altitude/10)*100+speed;
display=tmp_display;
```

Two digits are perfect for speed expressed in meters per second—always in the range 0 to 99 during our simulation. If you want to use knots as in real planes, you have to multiply speed in m/s by 194 and divide by 100, or by 1000 to show tenths of knots and remain in the two digits range.

The same goes for altitude. Our simulator shows tens of meters, but you can easily convert them to feet to adopt the units used by real aircraft. Multiply meters by 328, then divide by 100 to get feet, or by 1000 to get tens of feet (see the Upsizing the Project section later in the chapter for a more sophisticated usage of the display).

Into the same routine, we also placed the instructions that control the loudspeaker:

```
if (stall==1 && flying==1)
  PlayTone(1760,30);            // stall alarm
else
  PlayTone(80+throttle*12,105); // engine noise
```

In case of stall, the routine plays a high tone. (In normal operation, it plays a low sound whose frequency is proportional to the throttle.)

At this point, there's one last thing you have to code: the motion of the platform. We used a simple system to control speed: In a given period, the motors stay on for a time proportional to speed. These instructions are in a separate task that loops with a tighter interval than the main one so motion results more smoothly than it would in a loop of 1 second. Our program cycles every 20 hundredths.

```
#define PERIOD 20
while (true)
{
  if (speed==0)
  {
    Off(OUT_A+OUT_B);
  }
  else
  {
    on_left=(speed*PERIOD)/100;
    on_right=on_left-turn*3;
    on_left+=turn*3;
    OnFwd(OUT_A+OUT_B);
    if (on_left==on_right)
    {
      Wait(on_left);
      Float(OUT_A+OUT_B);
      Wait(PERIOD-on_left);
    }
    else if (on_left>on_right)
    {
      Wait(on_right);
      Float(OUT_B);
      Wait(on_left-on_right);
      Float(OUT_A);
      Wait(PERIOD-on_left);
    }
    else
    {
      Wait(on_left);
```

```
        Float(OUT_A);
        Wait(on_right-on_left);
        Float(OUT_B);
        Wait(PERIOD-on_right);
      }
    }
}
```

The two variables *on_left* and *on_right* contain the time each motor must stay on. This time is proportional to speed, and affected by the turning of the plane: for each degree in angular velocity we transfer 3 hundredths of a second from one motor to the other (quite an experimental parameter).

The code starts the motor, and if they are expected to run for the same time, stops them simultaneously. In case the aircraft is turning, one motor will stop before the other.

Operating the Simulator

Be sure that the small plane on the platform is perfectly level. Place it on the runway (that is, on an open space on the floor) and start the program. Apply full throttle and look at the speed on the display. When it reaches about 45m/s (87 knots) pull gently on the yoke for a while to raise the nose about 10°. You're taking off! Your altitude should start increasing every second.

When you reach the altitude of your choice, level the aircraft, pushing the yoke for a while until pitch goes to about 0. In this attitude, with maximum throttle, the plane continues to climb: reduce throttle a bit to obtain straight and level flight.

Now you can experiment with turns. Push the yoke right or left for a while until the plane banks about 30°, then center the yoke again. The platform starts turning slowly. You should notice that during turns you lose some altitude, but you can apply the elevators a bit to compensate for this. To exit the turn, push the yoke to the other side until the plane levels again. Remember to reset the pitch.

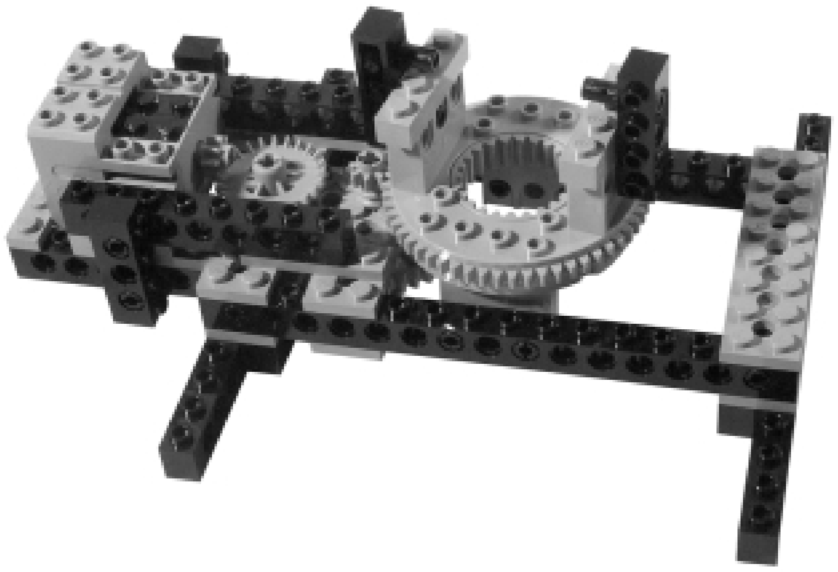
You can experiment with nose-ups and dives, also. Remember that the maximum positive pitch your aircraft can bear is 16°, with higher values it will stall. On the other hand, there's no limit, other than the physical structure of the simulator, to do a negative pitch.

Downsizing the Project

Let's explore what you can do to reduce the requirements of the project. In the following paragraphs, we suggest some options that can be used alone or combined together.

You can make a static version of the simulator—something that stays on your table instead of navigating the room—that only turns when the plane does. In this case, you substitute the differential drive platform with a static support. Figure 24.17 shows a possible rotary platform that combines with the pitch and bank assembly of Figure 24.7 to create the static simulator of Figure 24.18.

Figure 24.17 A Static Base for the Simulator



The complete simulator of Figure 24.18 can be built using only MINDSTORMS parts plus the turntable, a motor, and the rotation sensors. Be sure to pass all the cables inside the hole of the turntable for maximum turning capability.

If you don't have the turntable, you can build a rotating support using a 40t gear as shown in Figure 24.19.

In this static simulator, you only need three motors, thus you can connect them directly to the RCX out ports and avoid the polarity switches, using either the LEGO remote or some software on your PC to drive them via the IR interface.

You could even remove the pitch and bank motors and replace them with mechanical couplings. Just one motor needed in this case!

Figure 24.18 The Static Simulator Assembled

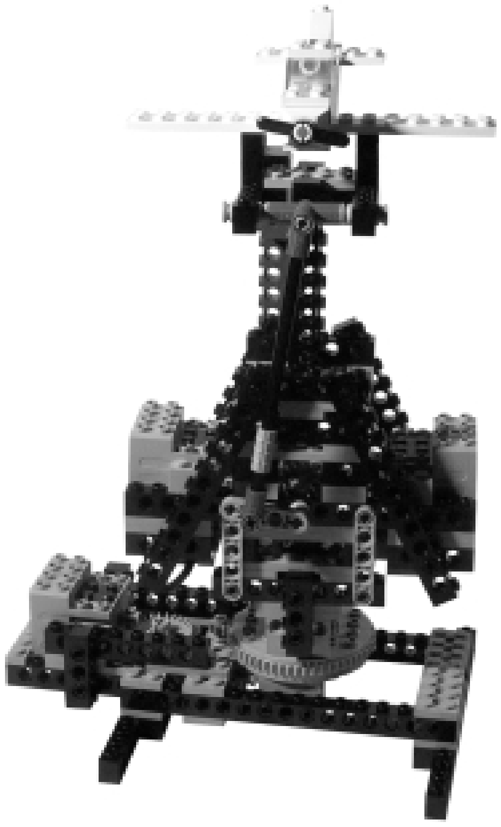
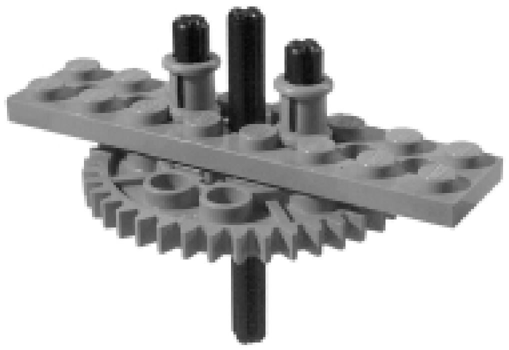


Figure 24.19 A Homemade Turntable



If you want to maintain the movable platform but use only three motors, you can replace the differential drive with a steering drive, using one motor to drive the main wheels and connecting the bank system to the steering wheel so that when you bank the aircraft right, the platform steers right, too.

In regards to sensors, you can replace them with light sensors that look at a white-black gradated surface that moves according to the inclination in pitch or bank. Experiment with readings and create a function that converts them to a reasonable approximation of the angle.

Even touch sensors can be used in place of one rotation. For example, you can insert two of them to read pitch with a simple scheme like: front switch pressed means pitch –10°, rear switch pressed means pitch +10°, none of them pressed means pitch 0°. This works, but requires two input ports.

The throttle rotation sensor can be replaced by IR messages sent by the LEGO remote (or the PC)—for example, message 1 to increase throttle and 2 to decrease it.

Upsizing the Project

There are many things you can do to make the flight simulator more sophisticated and complete. Starting on the software side, you can program it with one of the alternative languages that allow better control of the display and, more importantly, the push buttons.

The display has a single digit on the right (program slot) that can be used, for example, to show the throttle value. The small arrows that in the standard firmware show the status of in and out ports can effectively be employed as pitch and bank indicators, so you can more easily level the plane.

A whole new world of possibilities comes from the push buttons: The **View** button, for example, has its natural designation in allowing the display to exhibit different data: altitude, speed, heading, and any other value you would like to keep under control. The **Prgm** button, on the other hand, can be used to apply flap for takeoff and landing. With all digits available for view, you can use the basic display for tens of meters, then hit the **View** button for more accurate readings.

Or you could assign two buttons to throttle, one to increase it and the other to decrease it, so as to free one input port and use it for the rudder control, or just to save a rotation sensor.

With a language that allows variables with higher precision and trigonometric functions, you can redesign the mathematical model to make it more accurate and realistic. We deliberately ignored many parameters that influence flight, like the air density and mass of the aircraft, just to name a few.

Moving all the software on the PC is another possible radical approach. You could place the RCX on the platform and control it via IR, piloting the plane from a virtual cockpit on the screen of your PC. The software then informs the RCX only about the expected actions the platform must perform.

And why not put a small video camera in place of the plane in order to view on screen what you'd see if you were a LEGO figure inside the plane?

There are so many possibilities. Now it's your turn!

Summary

Tough subject, isn't it? Why might you approach a difficult project like this? For the opportunity to learn something about flight, or for the challenge to succeed in making such a complex machine work? In our case, both reasons were important, but we must confess that the biggest reason was because we thought it would be fun.

The enjoyment of piloting a plane in your living room using furniture as obstacles is not just for kids. You can imagine the side of the couch as one wall of the Grand Canyon or the coffee table to be the Golden Gate Bridge. The reassuring noise of the engine coming out of your RCX, one eye to the instruments and the other to the landscape slowly flowing below you, can make you feel like the next Charles Lindbergh.

Apart from being a lot of fun, the Flight Simulator project contains some lessons for you. The conversion of our mathematical model into actual NQC code provides many good examples of the techniques described in Chapter 12 about minimizing the loss of precision during calculation with integer numbers. You noticed how much care we put into translating any single equation into program instructions: If you don't attentively consider the domain of the numbers you enter in your formulas, you take the risk of running into unexpected results.

This project also teaches you that you can use your MINDSTORMS kit to emulate complex machines that you cannot actually build. In fact, in the introduction to the chapter, we explained that though it's not possible to build a flying LEGO airplane, you *can* simulate one. Similarly, you can build a simulator for a submarine, or a spaceship, and learn a great deal about the principles that control their navigation.

Chapter 25

Constructing Useful Stuff

Solutions in this chapter:

- **Building a Floor Sweeper**
- **Building a Milk Guard**
- **Building a Plant Sprinkler**
- **Designing Other Useful Robots**

Introduction

Sooner or later you will be asked the fatal question: "Couldn't you build something *really useful*?" The right answer is rather obvious: "All of my robots are useful to me. They provide me with a creative outlet and help broaden my thinking." However, be prepared to face the fact that this answer probably won't satisfy most people—after all, if they have to ask in the first place, they're not going to get it. If you really want to stop them from pestering you by building something truly useful, you'll find some suggestions in this chapter.

The projects we'll describe here only need a few parts not contained in your basic MINDSTORMS kit. The Floor Sweeper, for instance, requires some foam and tissue paper, the Milk Guard employs a temperature sensor, and the Plant Sprinkler needs a small pneumatic pump, a short piece of pipe, and a plastic bottle like those water and soft drinks come in.

Though these robots may be simple, they have their merits in applying some of the concepts discussed in Part I and in introducing some new ideas. For example, the Floor Sweeper covers room navigation, suggesting approaches that refer to the absolute positioning methods of Chapter 13. The Milk Guard describes a possible application for the temperature sensor, while the Plant Sprinkler explores the possibility of using LEGO elements to pump water. We'll explain the simple physics involved in the indirect method it uses.

Building a Floor Sweeper

This robot is based on the principle that a vehicle moving randomly about a room will eventually touch every point of the floor. You might point out that this random (*stochastic*) approach is not very efficient, but it's a good navigation exercise—plus, the request was just for *useful* stuff, remember!

Constructing the Sweeper

For the proposed sweeper technique to work, you need a hard, smooth floor. It definitely won't work on any kind of carpeting.

Our robot is simply a differential drive that in place of supporting casters mounts a relatively large wiper, wrapped in a piece of tissue paper (Figure 25.1).

You can see that there's really nothing different here than anything else we've featured in the book, except for the two very large bumpers designed to detect most common objects that occupy a room. Each bumper has its own touch sensor (Figure 25.2).

Figure 25.1 The Stochastic Floor Sweeper

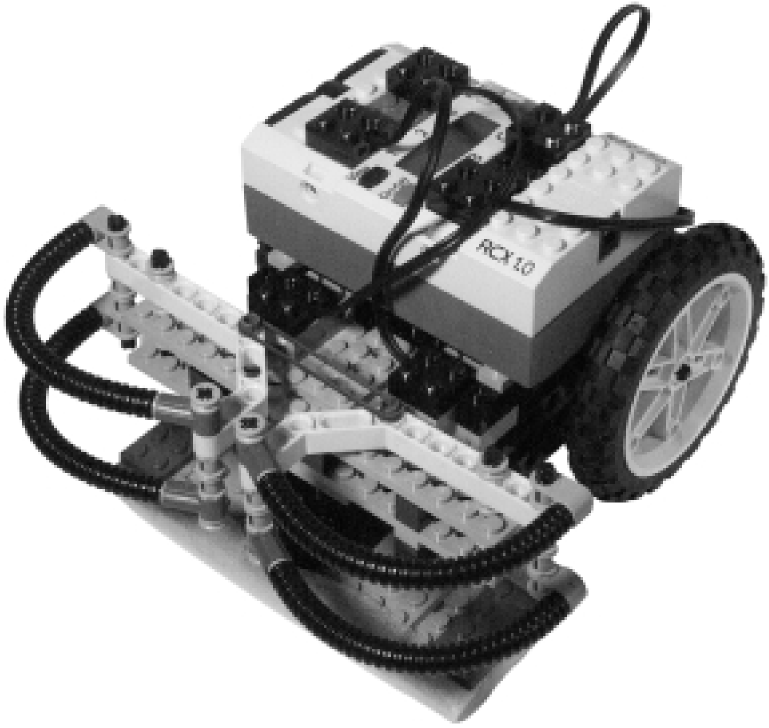
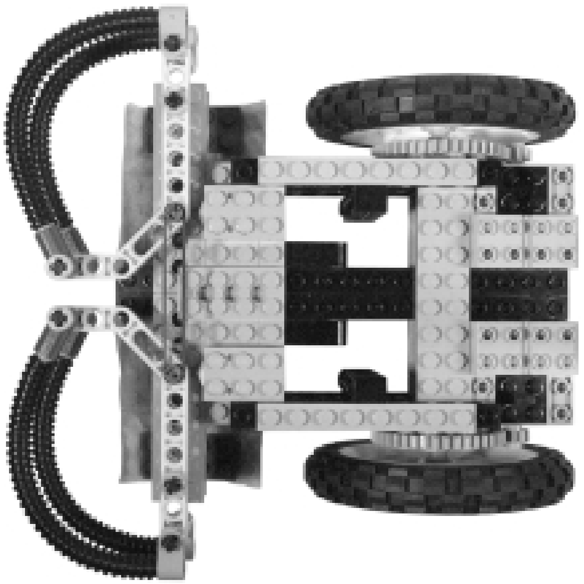
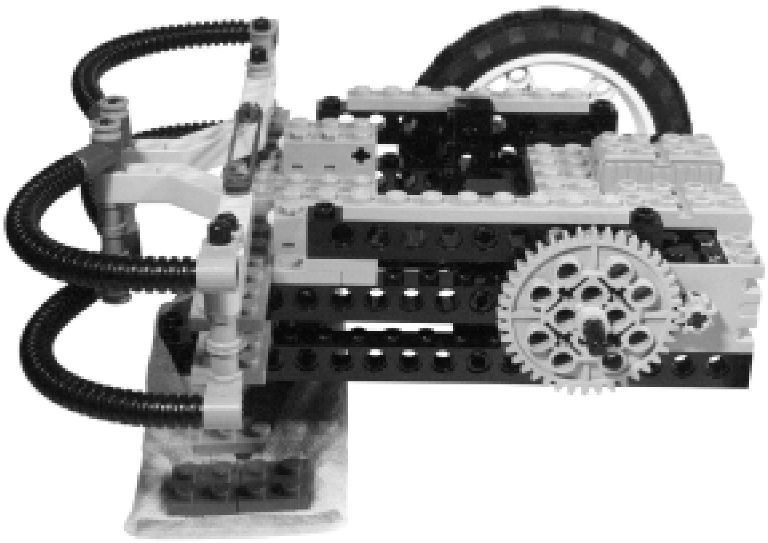


Figure 25.2 Top View of the Floor Sweeper (RCX Removed)



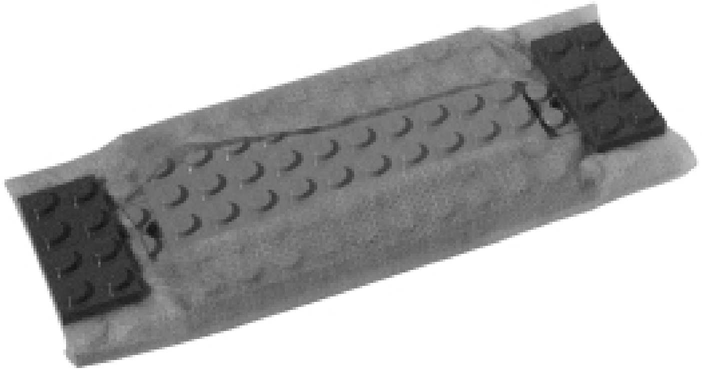
The motors drive the wheels through a direct 1:5 (8:40) gearing (see Figure 25.3).

Figure 25.3 Left Side View of the Floor Sweeper (Wheel Removed)



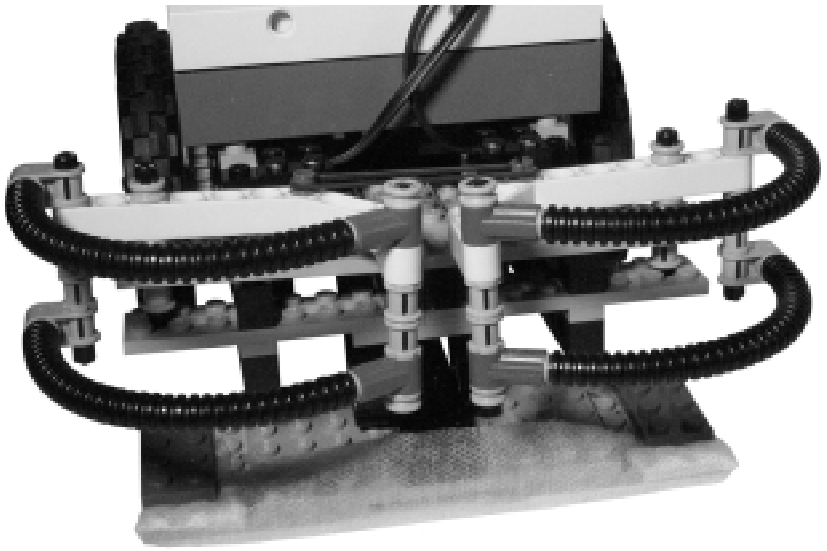
We made the wiper from three layers of different materials: LEGO plates (on top), a thin sheet of foam rubber, and tissue paper. The foam rubber is attached to the plates with a small piece of double-sided adhesive tape. The tissue paper, easily replaceable, covers the foam rubber and then folds over the top of the plates to be pinned between them and four 2 x 2 plates (see Figure 25.4).

Figure 25.4 The Wiper Component



The wiper is jointed to the body of the robot in order to make sure the entire surface is in contact with the floor (Figure 25.5).

Figure 25.5 Close-Up of the Floor Sweeper Bumpers and the Wiper



Programming the Sweeper

Our Floor Sweeper is stochastic, that is, it moves in a random pattern, so just program your robot to go in whatever direction and turn at whatever angle you want it to when it runs into an obstacle. For the sake of simplicity, let's say your robot will go straight until something happens, maneuvering to change direction only when one of the bumpers is closed.

The length of the turns can be purely random, or they can be controlled by a random factor combined with some form of "intelligence." For example, when the robot detects several hits in a short period, it is probably stuck in a blind alley. A wider turn can help it find a clear path.

Improvements on the Floor Sweeper

This robot is so limited in intelligence that almost any change can improve it! Start with the bumpers. They could probably use some adjustments to better detect the kind of furniture that occupies your own room. A very nice improvement is the adoption of one (or better yet, two) rotation sensor(s) to implement indirect collision detection as described in Chapter 4.

The rotation sensors open the way to dead reckoning and thus to position control. This is a *big* improvement, but be aware that rough navigation won't help you much, especially if it accumulates errors. Either your robot knows where it is and where it goes with controllable precision, or you'd better stick with random navigation, which, all things considered, isn't such a bad option.

If your floor has obvious grooves or markings (like the grout lines between floor tiles), you have the opportunity to utilize them as natural landmarks, using the absolute positioning methods described in Chapter 13 for navigating a grid. The best equipment to help the robot find its way on this grid are two light sensors. Place them on the front of the robot, one centered and the other at the extreme left (or right). This way your robot can follow a longitudinal line with its side sensor, while the other helps detect perpendicular grooves. When a collision occurs, it should turn 180 degrees and start following the next line or the other side of the previous one, alternating the two rules.

This, of course, is easier said than done, but brings great satisfaction when it actually works. If you succeed in reliable tile navigation, the next step is to implement a map (where each tile represents a unit) so that your robot can circumvent obstacles and find the proper row of tiles again.

Building a Milk Guard

Milk has wonderful physical properties that are no less surprising than its nutritional ones. Did you ever try to warm up a pan of milk on your stove? If you stand there watching it, the temperature of the milk seems to rise incredibly slowly, about one degree per hour (or at least it feels that way), then as soon as you look away or get distracted by something, the milk instantly boils over and makes a mess on your stove! Seriously, when heating fresh milk, it should never be allowed to boil because this destroys many of its nutrients. The Milk Guard robot that we'll build in this chapter lets you quietly watch your favorite TV show, or build your latest robotic creature without worry, sounding an alarm when your milk reaches the desired temperature.

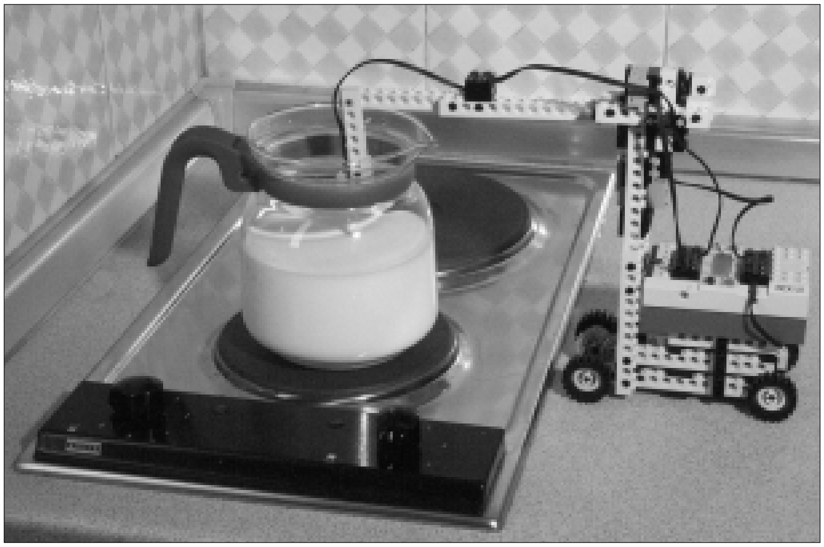
This robot is actually something more than a temperature sensor for milk, since it features a self-protection mechanism which prevents possible damages. In case you don't hear the alarm or don't get there in time to avoid the catastrophe, when the programmed temperature is reached, the robot pulls the sensor out of the milk and retreats a few inches, just in case!

Making the Milk Guard

This is the only project in the book where we employ a temperature sensor. It's also the only part of this robot that's not included in the MINDSTORMS kit.

Our Milk Guard is so simple, only a short description is necessary. Essentially, it is this: a wheeled chassis carries the RCX and a vertical support that ends in a lifting arm. At the end of the arm the temperature sensor hangs down, its metallic cylinder touching the surface of the milk (Figure 25.6).

Figure 25.6 The Milk Guard



The chassis has no turning ability at all, as this robot doesn't need it (Figure 25.7). The gear ratio is 1:9, obtained from two 1:3 stages. Together with the small wheel diameter, this ratio makes the robot very slow. There's no particular reason for such a geared down configuration, except that the robot is not in a hurry, and that there are all those nice gears in the MINDSTORMS box that would be a pity to leave unused!

The second motor operates the lifting arm through a worm gear—24t gearing and a pair of small pulleys. The upper 16t gear serves as a knob to manually lower the sensor into the milk when starting the operation (Figure 25.8).

There's a touch sensor that detects the uppermost position of the arm, thus allowing the robot to stop the lifting operation (Figure 25.9). Just activating the motor for a fixed time period wouldn't work, as the starting position of the arm may change from time to time.

Figure 25.7 Top View (RCX Removed)

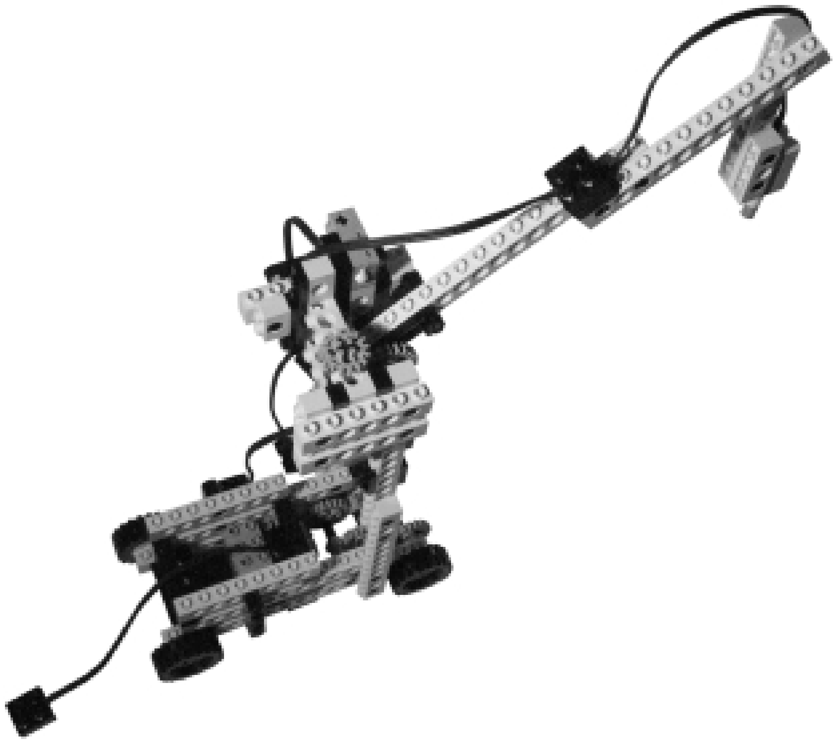


Figure 25.8 Side View

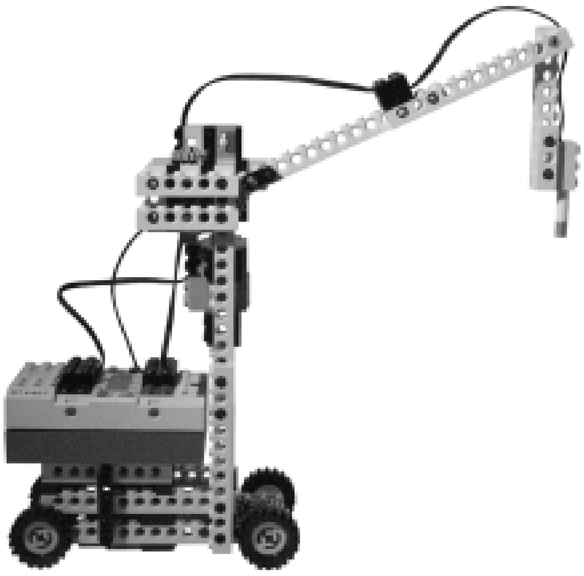
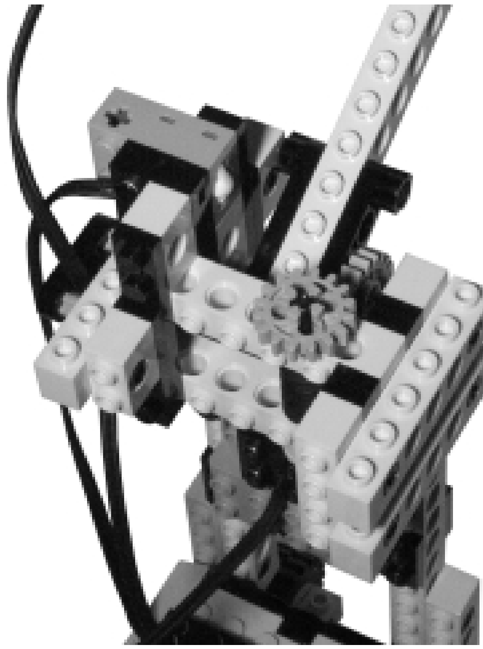


Figure 25.9 Detail of the Lifting Mechanism



Programming and Using the Milk Guard

When in action, the Milk Guard does nothing but wait for the temperature to reach a programmed value. As soon as this happens, it starts a task that sounds a short tone every second. Then it lifts its probe from the hot liquid and, when the arm is fully vertical, moves away from the site of the impending boil-up. The alarm stops only when the user stops the program.

The setup of the system is similarly simple. Be sure the metallic head of the sensor is well cleaned, then turn the knob to lower it into the milk. Keep the RCX switched off during this phase, so the motor offers less resistance (it's in float mode instead of braked).

WARNING

Despite being one of the simplest robots in the book, the Milk Guard has a long list of warnings:

- Hot milk and stoves can cause burns. To avoid injury, children should be supervised by adults when experimenting with this robot.

- Do not use this robot with a gas stove. The Milk Guard is designed for electric stoves. Gas stoves generate a flow of hot air that rises around the container, and this will damage (melt) your LEGO parts.
- Do not put the robot, or any LEGO part, into a traditional oven or microwave.
- The LEGO temperature sensor ranges up to 70°C (158°F), enough for the purpose of this robot. Do not expose the sensor to higher temperatures.
- Do not submerge the sensor into the milk. It could get damaged. Test the robot under safer conditions—for example, with a programmed temperature just above the ambient temperature, so you can simply warm the sensor with your hands and see what happens.
- Be sure that everything that comes in contact with the milk is perfectly clean so as not to contaminate it.

Improvements on the Milk Guard

Our version of the Milk Guard has the target temperature coded into the program. To make your own robot more versatile, you can add some input mechanisms so the user can set the desired temperature without having to modify and reload the program. A rotation sensor is perfect for this: make the RCX display the current target value and increment/decrement it according to the movements of the sensor. The touch sensor that controls the arm can also be used to confirm the value and start the monitoring process.

Instead of the rotation, you can use a plain touch sensor with some simple communication protocol of your choice—for example, a short touch to increase temperature, a long touch to decrease it, and a double-click to confirm.

The physical structure of the robot may require some changes to make it fit your own situation; typically its distance from the stove and the height of the support might need some minor adjustments.

Building a Plant Sprinkler

You are about to leave for a long-awaited vacation, but are worried about the fate of your endearing house plants: Will they survive your vacation if not watered? Fear not. The robotic Plant Sprinkler is the answer to your problem.

Making the Sprinkler

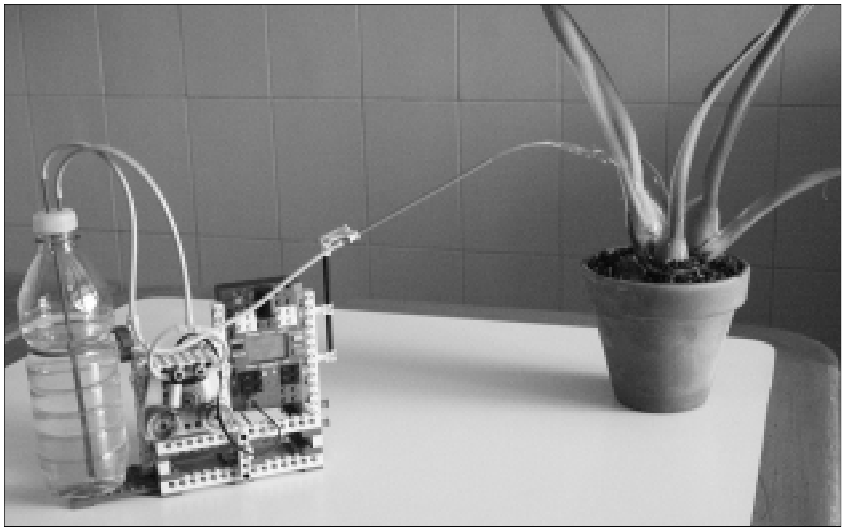
What's interesting about this robot is that it introduces the idea of pumping fluids with LEGO components. Is the pneumatic system suitable to be used with water instead of air? Yes, it is, but we strongly discourage you from trying this technique because it will spoil your precious pneumatic pumps and cylinders. They have not been designed with water in mind, and the guys that experimented with water then had to open the cylinders, clean their insides, and lubricate them again.

This stated, you need some other way to pump water indirectly. For our part, we adopted a solution that aspires to Michael Brandl's Adam der Gärtner robot, with minor modifications. There are other possible ways to do this, too, things we'll later explore in the Improvements on the Plant Sprinkler section.

Let's use the following principle: If you create pressure in a sealed container filled with water, and provide an outlet for the water to exit, it will flow out until the inner pressure balances against the atmospheric pressure. To state it in simpler terms, the incoming air pushes the water out.

As often happens, this is more complicated to describe than to see in action. Figure 25.10 shows our Plant Sprinkler; note that the core component of our system is a plastic bottle typically used for soft drinks or water.

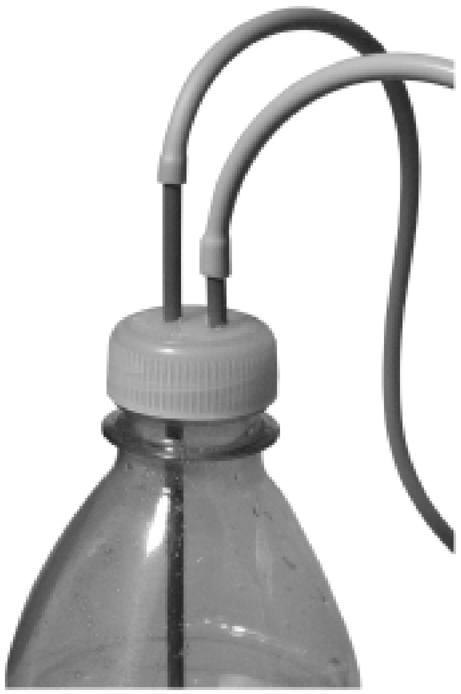
Figure 25.10 The Plant Sprinkler



Any bottle or container will work, provided you can make an air-tight seal. Drill two holes in the cap to insert two LEGO tubes, the shortest being the air inlet and the longest the water outlet. The water pipe must reach the bottom of the bottle.

In our case, the pipes fit the holes well, friction keeping them airtight. We didn't need any additional sealant (Figure 25.11), but you can use silicone to seal possible slits if you suspect air is escaping.

Figure 25.11 Detail of the Modified Cap



We used a double-acting compressor, but any one will work (Figure 25.12). Efficiency is not a concern in this project.

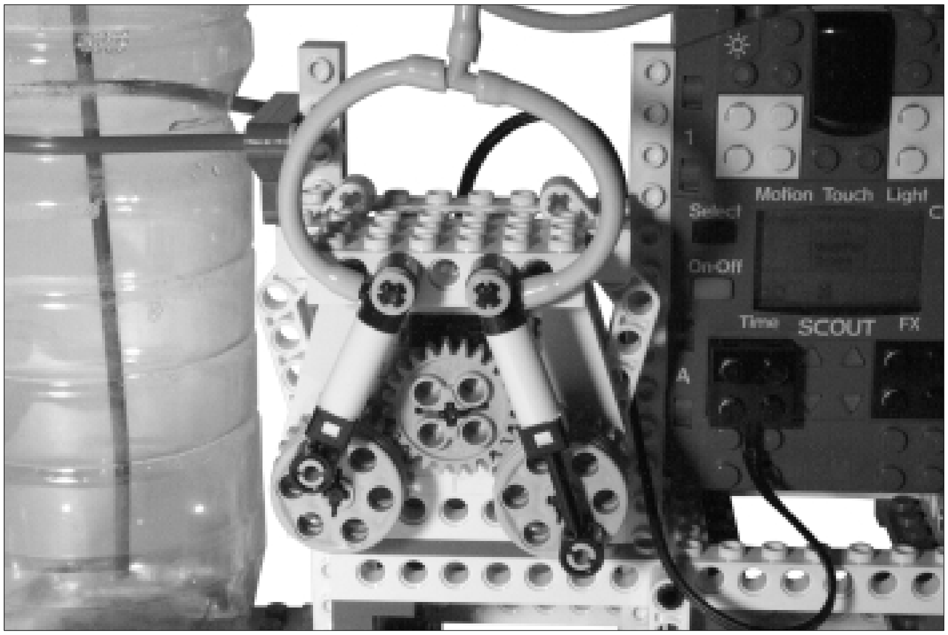
Our robot features the Scout programmable unit, but nothing prevents you from substituting it with the RCX.

Programming and Using the Sprinkler

All that this robot requires to work is that you switch on the compressor when you want to pump water. You can program your RCX to work as a timer that starts sprinkling at a given hour.

Because plants should not be watered during the hottest hours of the day (when the sun is high in the sky), the light sensor can be used to trigger sprinkling when light dips below a certain level. You can even combine time and light to make your system more reliable.

Figure 25.12 The Double-Acting Compressor



To decide how long your compressor must stay switched on, measure the average quantity of water expelled in a given period. Be aware that the water will drip for a while after you've stopped the compressor, at least until the pressure inside the bottle equals the pressure outside it.

Improvements on the Plant Sprinkler

This robot needs many improvements to really be usable. The first and most serious of its limitations is the power supply. If you leave your RCX turned on, it will drain its batteries flat in a few hours. The RCX included in the first version of MINDSTORMS included a socket for external adapters (a very nice feature that has unfortunately been removed from more recent versions). If you own the first version, just find a proper adapter and the problem is solved. If, on the other hand, your RCX is a later design, a possible workaround can be found in the sidebar, "Supplying Your RCX with an External Power Source."

The second problem that affects our robot comes from the limited reserve of water. You can apply the principle we used to a larger container, but when the volume of air inside it becomes big, the compressor will take a long time to reach the pressure required to pump the water out. In this case, you better adopt a different solution—for example, gravity.

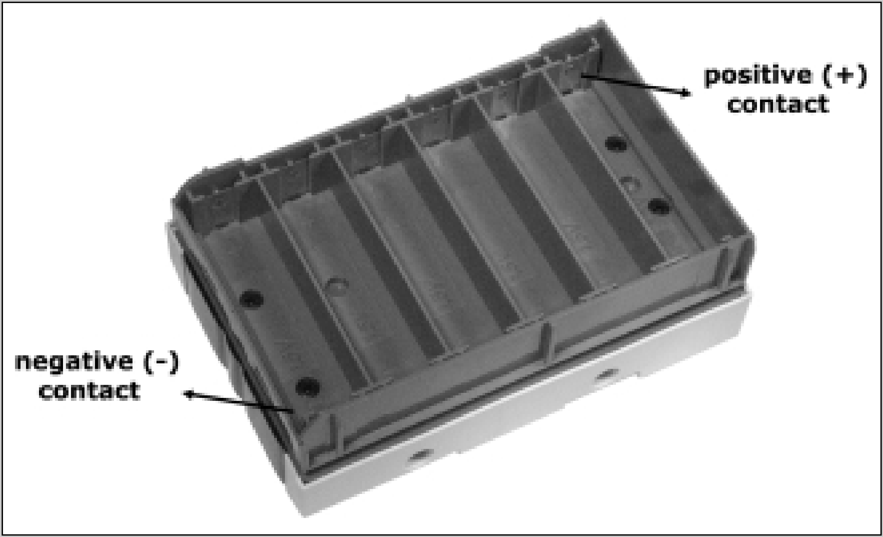
Bricks & Chips…

Supplying Your RCX with an External Power Source

Supplying a later RCX version with an external source is actually possible, but we discourage you from attempting it unless you have a good deal of electrical experience. Wrong voltages or polarities may permanently damage your RCX.

1. Make two fake batteries using two small wood cylinders of appropriate length and diameter (other insulating materials will do the trick as well).
2. Mount a small screw on one end of each cylinder, and attach a wire to it either soldering or tightening it between the head of the screw and the wood.
3. Open your RCX, and leaving the six batteries inside, identify with a tester the contacts where the first and last batteries supply a full nine volts (Figure 25.13). Replace those batteries with your fake ones and remove all the others as well.

Figure 25.13 Opening Your RCX



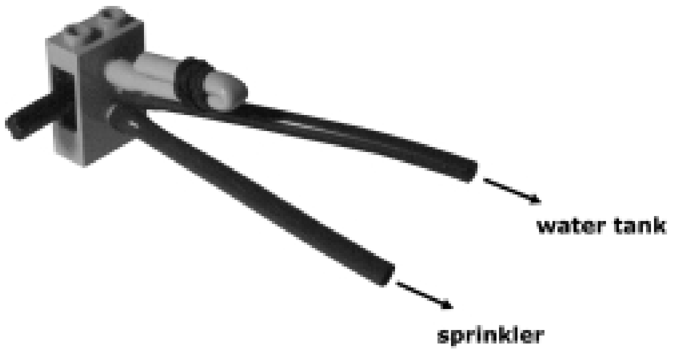
4. Route the wires outside the RCX—there's a notch in the body that seems designed for this purpose—and close the cover.

Continued

5. Connect the wires, respecting polarity, to a 9V DC, 1.5A *regulated* adapter.
6. Supply power to the RCX and turn it on to check if it works.
7. Reload the firmware.

To use gravity you don't even need the compressor: Simply put your water tank high enough so the water flows naturally down the pipes. A pneumatic valve switch can work as a tap, so all you need is to drive it with a motor as described in Chapter 10. Connect the central hole of the valve to the hose coming from the water tank, one of the side holes to the sprinkler, and the other side hole to a short closed piece of pipe (Figure 25.14).

Figure 25.14 The Water Tap



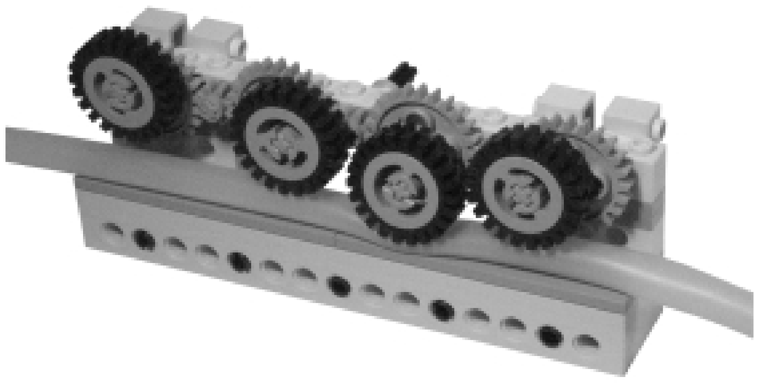
The pressure depends on the difference between the height of the tank and the sprinkler, each meter producing theoretically one tenth of an atmosphere, which is strongly reduced by the friction of the fluid in the pipes.

WARNING

Do not connect your valve system to the tap or connect it in any way to the water network of your house. The pressure would be too high. If for any reason a pipe junction breaks, you'd come back from your vacation to find your house converted into an aquarium.

There's one last indirect pumping technique we would like to show. If you press a wheel down on a soft rubber pipe until you close it, then move the wheel forward along the pipe, any fluid inside will be pushed forward, too. You apply this principle every day when you squeeze toothpaste out of a tube, and it's also used in medical procedures for regulating bloodflow. Figure 25.15 shows a possible setup that requires a larger diameter and much softer pipes than those supplied with LEGO pneumatics. The pipe remains fixed in place, while the wheels roll over it. Make your actual version more solid than this prototype, whose *raison d'etre* is to explain the principle.

Figure 25.15 A Possible Setup for an Indirect Pumping Technique



The obvious solution when faced with watering more than a one plant is to make a network of pipes that reaches every pot. A less practical but more impressive way would be to use a mobile robot. It could travel along a black line that borders all the pots, reading special marks that tell it where to stop and activate sprinkling. Or it could run on train tracks, again reading stops from some external reference.

Designing Other Useful Robots

If you still haven't found what you consider a "useful" robot, the following tips might provide you with the proper inspiration.

- **Alarm clock** The RCX, as is, can be used as alarm clock in many different configurations.
- **Baby entertainer** This is something to hang over your baby's cradle, like a mobile to which you could attach some puppets or your child's

favorite toys. The RCX can slowly turn the mobile, while at the same time playing a suitable melody. Lots of warnings apply here: Make sure everything is solid so there's no risk of it falling into the cradle; use long wires to keep the RCX (the heaviest part) far from the cradle; keep everything out of the reach of the baby; small LEGO parts represent a choking hazard; always use batteries instead of power supplies.

- **Pet feeder** If your favorite pet is fed with some kind of dry food, you could devise an automatic feeder that supplies its bowl at predefined times. Please don't rely on this system for the survival of your pet during vacations!
- **Dog trainer** Most dogs love to return a ball. You can make a robot that throws the ball anytime your dog drops it in a specific receiver. Be prepared to clean your LEGO parts of a certain amount of drool.

Summary

Despite their limitations, which actually make them rather charming, the robots of this chapter demonstrate that your MINDSTORMS kit can be turned into many "useful" devices. Making robots interface with the real world involves many difficult tasks, which can lead to either funky but not-so-effective machines, or to very practical but uninteresting projects. But, hey, did you really buy a $200 kit to build a kitchen alarm clock?

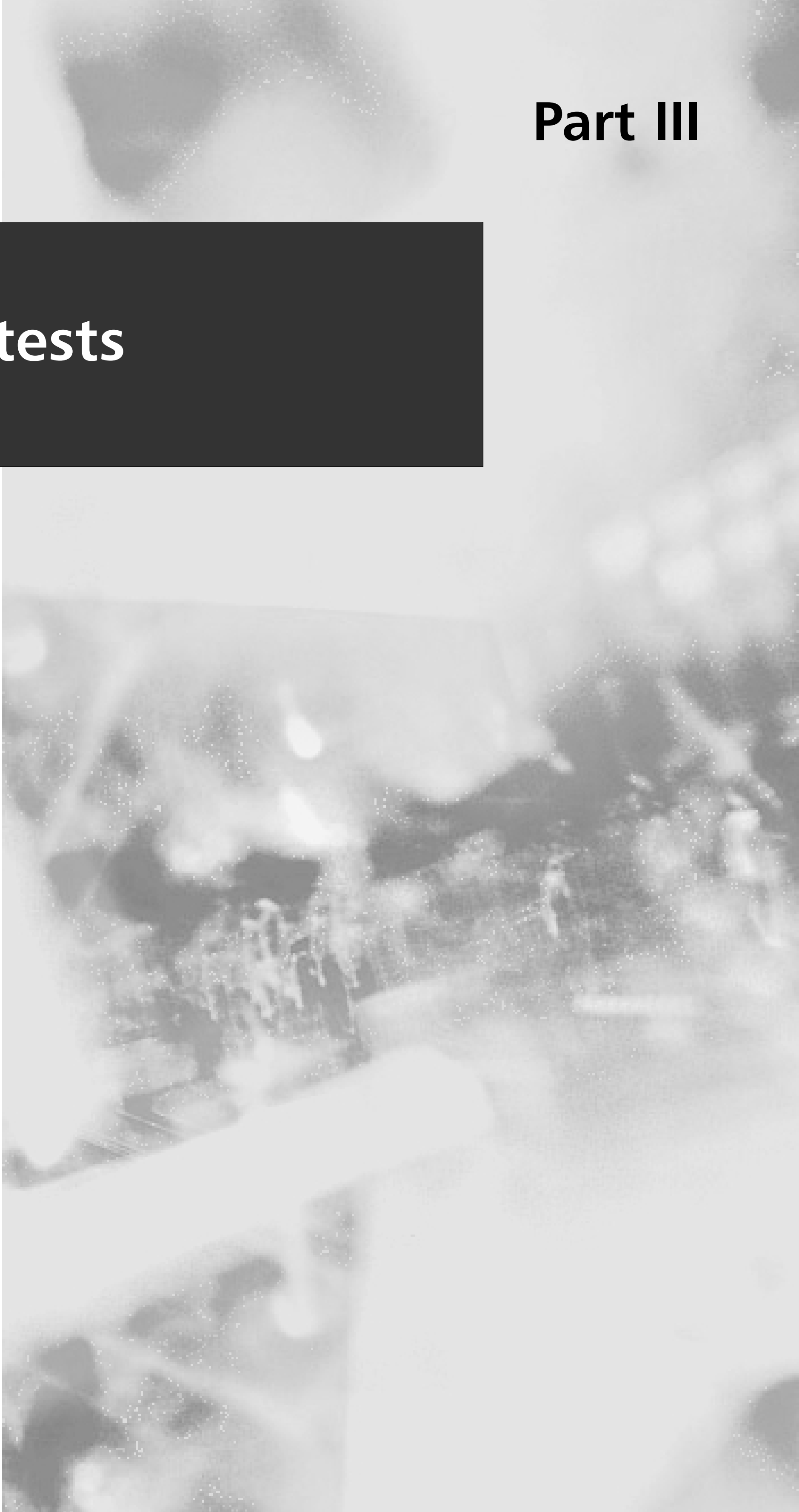
The projects we devised for this chapter, the Floor Sweeper, Milk Guard, and the Plant Sprinkler, do provide a starting point for talking about some offbeat capabilities, like pumping water. In fact, the Plant Sprinkler shows an unusual application of the LEGO pneumatic system. Since the LEGO pumps and cylinders are not suitable for direct use with water, we went around the problem, and used an approach whereby the power of compressed air pushes the water.

The Floor Sweeper explores more problems involved in exploring a room and navigating the obstacles therein. This robot is conceptually very similar to the differential drive described in Chapter 14, and faces the same difficulties—however, this time we looked into using tiled floors, the ideal situation to experiment with the absolute positioning methods illustrated in Chapter 13. The grooves between the tiles turn your floor into a grid your robot can navigate using light sensors.

Before someone at home gets too enthusiastic about the stochastic Floor Sweeper, please tell him/her the truth about the time it takes to clean even a small room, and how many batteries a week are consumed in the process!

Part III

Contests



Chapter 26

Racing Against Time

Solutions in this chapter:

- **Hosting and Participating in Contests**
- **Going as Fast as Possible**
- **Combining Speed with Precision**

Introduction

This chapter opens the third section of the book, where we explore the world of MINDSTORMS robotics contests and challenges. The three chapters that make up this section are mainly based on our direct experience, accumulated while attending competitions organized by the Italian LEGO Users Group (ItLUG). We won't be discussing the specific details of the contests we participated in, instead we'll be providing you with a good starting point for more general considerations.

The first section of this chapter is about robotic contests in general. We will explain what robotics contests are all about, from the definition of the rules up to the course of competition. For those of you interested in participating in LEGO robotics contests, we'll give you some hints about how to find a LEGO Users Group not far from where you live.

In the later sections of the chapter, we will introduce contests related to pure speed, as well as those demanding great amounts of mechanical and programming acumen. There are many different kinds of contests and challenges. Because of this, we grouped them into three categories: contests based on speed, contests based on strength, and contests based on ability. These categories are not absolute, because most of the competitions require a mix of these capabilities. For example, a line following contest is mainly about speed, but each robot is also required to run without departing too much from the line. Nevertheless, we tried to sort a few typical contests into the categories previously mentioned because in our opinion this helps in focusing on their key points.

Hosting and Participating in Contests

A contest offers many opportunities to learn new concepts and build some experience. We can identify at least four main phases of participating in a contest, each one requiring extensive usage of your know-how while contributing to your knowledge base. They are:

1. **Defining the rules** Participating in this phase depends on whether you are the one who organizes the contest, or part of a group that does. Unless you're deciding on your own, this will prove a very creative moment, where the group develops a list of rules, adjusting them until it feels they are meaningful and consistent. A set of rules always has a specific purpose (whether declared or not), which has been chosen to test the ability of the competitors on a specific field. The "legislator" should take care to close any possible loopholes that might allow a contestant to

escape the main difficulties of the contest, which requires that he/she imagine all the possible approaches to the problem.

2. **Studying the rules and deciding on a strategy** From this moment on, you are in the competitive arena, and must find a strategy to beat your competitors. Don't limit your choices to what the organizing committee expects you to do. In our experience, most contests have been won by people who found a very original way to interpret the rules without violating them.
3. **Building the robot** This phase will very likely present some surprises to you. Implementing your desired strategy, you'll discover new constraints and opportunities you hadn't thought of while imagining your robot. As for programming, we strongly suggest you stay with simple but solid strategies. Only when you're sure the basic behaviors work as expected, should you add in the more sophisticated components, making sure not to introduce bugs in the previous code. You can't imagine how many matches you can win by simply not getting too fancy!
4. **Attending the contest** This is the most exciting moment—on the field, testing your ability against your competitors'! It's also the moment to learn: study the other robots and their strategies; observe the course of the matches. Don't be frightened to ask for explanations and details, most of the builders are usually more than happy to describe their creatures. All that you learn will be useful for other contests, whether run on the same set of rules or not. One last suggestion: never throw in the towel before the end, anything can happen during the event. The strongest competitors aren't always crowned winners.

We imagine some of you are thinking right now: okay, very interesting, but where do I find a contest that isn't light years from home? Remember, the Robotics Invention System is a tremendous success. With a bit of luck you may find an already organized LEGO Users Group in your area. Many exist in the U.S., Canada and Europe, covering most regions and major metropolitan areas.

Use the Internet to search for other MINDSTORMS fans. The best resource is the LEGO Users Group Network (LUGNET), which lists dozens of local groups. Many of them also have their own Web site, which shouldn't be difficult to find using any search engine. Once you've found a group, or some individual users, there's no certainty that anyone's going to leap up and organize a robotics contest from time to time. But you, yes you, can be the one to get the ball rolling (or robots rather).

Last but not least, try to attend some *remote* challenges, contests that don't require your presence in person. Usually all you send are some pictures of your robot, a copy of the software, and a short description of how it works. In code-only contests, this concept is pushed to the extreme limit: the organizing committee distributes the plan of a standard robot, and all the competitors send their own code via e-mail. The design of the robot is usually simple and doesn't require any special or rare parts, so that a large number of competitors can replicate the robot at home to debug and test the code.

LUGNET is the best place to find information about contests of all sorts, as most local groups advertise the contests they organize there. Usually they refer you to a Web site where you can find all the details about the time, place, and rules of the contest. Some user groups require a small admission fee for each robot, which funds the prize for the winner. Events are characterized by a very friendly atmosphere, and you'll be welcomed even if you just go to watch.

Optimizing Speed

The first challenge we'll describe here concerns pure speed. Don't make the mistake of thinking speed is purely trivial and poses few challenges in terms of robotics. We've been proven wrong on this score ourselves. Even a straight-out speed race promises surprises.

Drag Racing

A starting line, a finish line, the fastest robot to cover the distance wins. Described in these terms, the race sounds a bore. But stay tuned, and take a closer look at the implications of this definition.

The speed of a vehicle is affected by a number of factors: motor power, gear ratio, mass, friction. Using electric motors, the maximum power you can apply to your racecar depends on the kind and number of motors, and the current you supply them. There's only one kind of motor in MINDSTORMS, which forces upon us a simple rule: use as many motors as allowed.

As for their supply of energy, the rules should outline some restriction, like the adoption of the same kind of batteries for all the competitors (e.g., standard commercial alkaline batteries). Should this not be carefully stated, someone could take advantage of a custom battery setup—which is exactly what happened during our dragster race!

Inventing…

A Very Special Battery Combination

Marco Berti won the first ItLUG robotic competition using eight NiMH batteries fitted inside the RCX. He needed eight because NiMH cells are rated 1.2V instead of 1.5V like the standard AA batteries. With eight cells he got 9.6V, a voltage just a bit higher than those supplied by fresh alkaline batteries, but tolerated by the RCX and the motors. What's unusual about rechargeable NiMH batteries is that they supply more current than alkaline ones. Electric power is proportional to voltage multiplied by current, so he definitely got more push for his motors. To fit eight cells in six slots, he used four standard AA size, and two 2/3 of AA combined with two 1/3 of AA. Those smaller batteries supply the same voltage and current as their bigger brothers, only for a shorter time.

Be very cautious in experimenting with custom battery setups: voltages higher than what the RCX is rated for can permanently damage your unit.

As for the gear ratio and mass, which have a strong influence on the acceleration rate of your vehicle, here is a short list of tips:

- The shorter the gear, the shorter the time it takes to reach the maximum speed. The problem is that a short gear also has low top speed. You have to balance the two effects, and the optimal choice depends also on the length of the race: favor acceleration on short tracks, and maximum top speed on longer ones.
- Build your robot in a way that allows easy replacement of the gears, so you can experiment with different ratios in a time-efficient manner.
- Keep the gearing pared down to the essentials—remember that each stage adds some friction. There's no need for a differential gear, since the dragster travels on a straight run.
- The diameter of the wheels has its role in the conversion of power to speed. If you substitute the wheels of your car with ones half the diameter in size, you get the same effect as if you had reduced the gear ratio by a factor of two.

- Acceleration is also influenced negatively by the mass you have to move: under the same power, the higher the mass, the lower the acceleration. This is due to inertia (see Chapter 5), which explains why it's harder to get a car rolling than it is to push a child in a stroller. So a very important thing to do is to keep the mass at a minimum. Build a lightweight structure.

Up to this point, the challenge is essentially electro-mechanical. No need for an RCX, a vehicle supplied by a battery box would perform the same, or even better (recall that the RCX has an inner current-limiting device, the battery box doesn't). To create the necessity of at least a few lines of code, in our dragster race we introduced the rule that the dragsters had to stop a very short distance after the finish line. As a result, we had three lines: start, finish, and braking limit. In our particular case, the run was 5m (about 5.5 yards) with a braking distance of 20cm (less than 8 inches).

The robots had been equipped with a face-down light sensor to detect the finish line, and a portion of code, the same for all the robots, to time the run and register it in the datalog.

Every competitor quickly discovered that braking inside the limit was not so easy. The fastest dragsters covered the distance in about 3 seconds, at a speed of 6 km/h (3.6 mph). We initially tried to switch the motor off at the finish line, but this wasn't enough. Then we reversed the motor to increase the braking effect, but it still wasn't enough braking power. Nobody wanted to reduce the speed of their dragster just to keep up with the braking limit. Surprisingly, most competitors developed the same solution: start braking *before* the finish line. Timing the performance of our dragsters without any braking, we registered the time they needed to cover the distance. Then we wrote some code to reverse the motors a few moments before reaching the finish line, trimming this advance until the robot stopped exactly at the extreme limit.

Combining Speed with Precision

When you move from races based on pure speed to those that require additional skills, your projects become more complex. All the considerations listed in the previous section still apply—batteries, motors, gear ratio, mass—but you must also take new variables into account. Speed will actually become what makes your task more difficult: when you design a robot for yourself, you usually feel satisfied when it works; but when you have to build and program it to be as fast as possible, some techniques that worked at a slower pace prove unsuccessful at higher speeds.

Sometimes you reach a point where you cannot increase speed without compromising the reliability of your robot. This is the time when a further improvement can come only from a *paradigm shift*, a change from one way of thinking to another. A good example of this was described in Chapter 14. You probably remember we initially approached line following using the same differential drive platform we used to navigate a room. It worked, but when we tried to achieve better performance, we had to move to a different architecture: a steering drive.

This principle can be summarized in a few words: don't set your heart on a particular solution, try to look at the problem from different angles and keep your mind open to any idea—even those which initially seem strange or unpractical may lead to a winning configuration.

Line Following

Don't worry, we won't start discussing line following again! Jump back to Chapter 14 if you feel compelled to revise some of those concepts. We just couldn't ignore line following in a chapter that talks about races against time, since it presents many interesting discussion points.

If you are the one who decides the rules, don't underestimate the importance of the details. State the number and kind of the allowed parts, motors, and sensors in particular. More importantly, be very precise regarding the nature of the path, informing competitors about the width of the line and the minimum radius of the turns—the latter having a strong influence on the structure of the robots. Very tight curves will favor differential drives, while a less winding run is surely the ideal terrain for steering drives.

Line following contests are usually judged only by speed. Evaluating accuracy, though theoretically possible, is not a very practical option. However, if you want to try this option, you can use a paper pad and attach a pen to each robot so they draw a line as they move. At the end of each run, you measure the maximum distance between the course of the robot and the main line, and apply greater penalties to greater distances.

Line following allows for many interesting variations, including:

- **Round trip** When the line ends, the robots must return to the starting point.
- **Short interruptions in the line, specified by number and length** For the robots, it's like hanging in mid-air for a while!

- **Small obstacles to overcome** The robots should detect these with bumpers, suspend line following, pass the obstacle, and resume line following again.
- **Obstacle removal** Similar to the previous variation except that objects of a specific size and shape must be removed instead of climbed over.
- **Specific robotic architecture** Specifying that a particular type of architecture be incorporated into the robot design. For example, all the robots must use legs instead of wheels.

Wall Following

Conceptually similar to line following, in this challenge, the competing robots must follow a wall instead of a line. The software is actually very similar to what works for line following, with only a few adjustments to reflect the difference in sensors.

If you decide to organize a wall following competition, remember that the walls used need not be real walls. You can create temporary walls with wood, cardboard, or any other material of your choice. Wall following can be as simple to set up as having the robot find its way around the perimeter of a large cardboard box. For example, you can state in your rules that the robots must run around the MINDSTORMS kit box; the fastest robot being the winner. Most of the participants will likely own the box, which will help them in setting up and testing their robots. As we've said before about line following, it's important you put a lot of care in specifying the details, including:

- The height of the walls, their color, and the material they are made of.
- Whether the robots are required to remain in constant contact with the wall, or if they can move apart from it for a while.
- The shape of the course, or at least what kind of angles the robots should expect.
- Whether or not the robots are allowed to "hook" the upper edge of the walls.

Moving to the point of view of the participant, the hardware configuration required to follow walls can be very similar to that shown in Chapter 19 with regard to maze solving (maze solving actually being a sophisticated variant of wall following). However, this is one of those cases where an increase in speed brings new difficulties. Similar to what happens in high-speed line following, the critical factor here is the reaction time of the robot. In fact, any time it loses contact with the wall and needs to undertake a corrective action, that longer reaction time entails a stronger correction.

As we mentioned in Chapter 14 when discussing how to optimize line following, this is easier said than done. To recapitulate, the elements you have to consider include:

- **The mechanical configuration of your robot** Type of drive, number of motors, position of the sensors, gear ratio, and backlash within gears.
- **The firmware you installed on your RCX** We explained that some alternative firmware offers faster code execution.
- **The algorithms used in the software** Strategies adopted to keep the robot on course as much as possible.

The mechanical configuration of your robot is something you have to experiment with. You can use the Maze Runner of Chapter 19 as a starting point, but the optimal solution also depends on the set of rules you'll use to race with. As for the firmware options, this is an opportunity to study a new language and install a new system, though not everyone will want to do that just to attend a contest.

As for the strategies, some of you may recall that in Chapter 12 we introduced hysteresis as a technique aimed at improving the efficiency of a system, because it reduces the number of corrections it has to make. It was definitely an interesting option for line following, but is it applicable to wall following, too? The answer depends on the configuration of your robot. If it relies on a touch sensor to "feel" the wall—like the Maze Runner of Chapter 19—hysteresis will be of no help, because all you can determine from the robot is whether it's touching the wall or not. To take advantage of hysteresis, you need finer information—you need to know the *distance* from the wall, so you can make your robot decide when and how much to correct the route. This implies that you have to replace the touch sensor with some more sophisticated device. For example, you could arrange a bumper, or antenna, connected to a rotation sensor in such a way that the count of the sensor is proportional to the distance. Or, if the rules allow for custom sensors, you could successfully use one of the distance sensors described in Chapter 9.

Other Races

There are many other type of contests that require your robot to perform some action as quickly as possible. As we explained in the introduction, most of them require some additional ability rather then just speed. In Chapter 28, we will describe contests where speed is important, but this is usually in the background when compared to other factors, like the efficiency in finding and gathering objects. In the following list, we suggest a few ideas for competitions in which speed is the most important component:

- **Car racing** Car racing is similar to drag racing, but the robotic cars run on a circuit that is more complex than just a straight track. The circuit may be delimited with colored tape on the floor, or with side walls. Avoid reducing the contest to line or wall following; instead, design the circuit so that a robot that follows one of the sides takes a longer route than those that run inside the track. If the circuit is delimited with real walls, encourage the competitors to use sophisticated detection techniques, like proximity sensing, by applying a penalty for every collision with a wall.
- **Fast painting** Each robot is equipped with a felt-tip pen and is asked to paint a given area on a sheet of paper. The robot that covers the surface fastest wins. Consider basing the results of each competitor on a combination of the elapsed time with the comprehensiveness of the coverage. The panel could be provided with a robot designed to scan the sheet and evaluate the result!
- **Wall climbing** Prepare a climbing wall equipped with special holds that a robot can seize (this could be as simple as a grid of horizontal bars); the fastest robot to reach the top wins. You can keep the competition open to ideas, allowing any kind of technique to reach the top, including lifting mechanisms and the launching of ropes.
- **Monkey bars** The Toronto Users Group (rtlToronto) is very active in organizing robotic contests. Their recent proposals include a monkey bar race. The competing robots are required to traverse a horizontal ladder, racing against another robot. The first one to reach the end, or the one who goes the furthest, wins (see Appendix A for a link to the rtlToronto Web site).

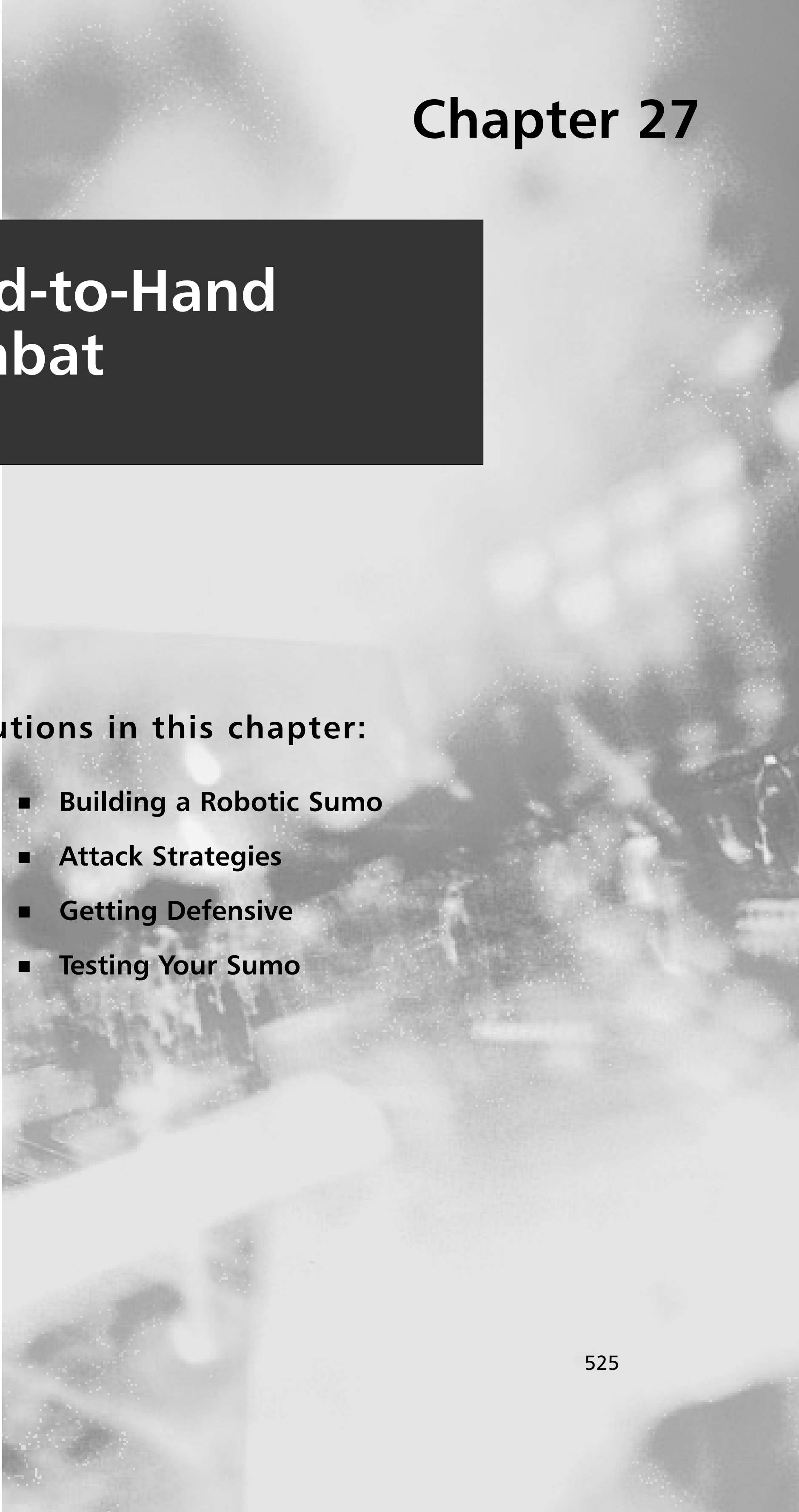
Summary

This chapter introduced you to the world of contests that represent a great opportunity to expand your knowledge, stimulate your creativity, and compare your ideas with others'.

Even races that seem the least "robotic" of all the possible types of competitions can spur you to find new solutions or improve old ones. During contests, the details are very important. Your robot should not only work, but work better than its competitors. For this reason, an apparently simple task like going straight and fast requires thoughtful planning of your project: batteries, motors, gear trains, wheels, weight of the vehicle… these elements are all crucial to success.

The simple addition of a limited braking space can make drag racing much more interesting, forcing the competitors to devise efficient braking solutions. Similarly, when you move to contests that involve highly specialized abilities, like navigation, the problems become much more complex. Tasks as simple as line following and wall following require a tremendous effort when your purpose is to design, build, and program a robot tuned for optimal performance. This is a process which proceeds by trial and error, and which will test your skills, your experience, your creativity and, most of all, your patience!

We encourage you to participate in contests. They can really be a great experience. Be humble enough to learn from your mistakes, or from more effective techniques rather than completely different approaches adopted by other robots. Take everything very seriously during preparation: Try different solutions, perfect the details, test your program thoroughly until you feel satisfied. But don't take the final rankings too seriously—remember, it's all in fun!



Chapter 27

Hand-to-Hand Combat

Solutions in this chapter:

- **Building a Robotic Sumo**
- **Attack Strategies**
- **Getting Defensive**
- **Testing Your Sumo**

Introduction

The contests described in Chapter 26 are the kind where each competitor has its turn, and the results compare the individual performances. In this chapter, we'll talk about competitions where the rival robots fight face to face in a more spectacular way.

In our experience, Sumo is one of the most suitable kinds of competition for small robots, offering the opportunity to test an incredible range of techniques that may prove useful in all your projects, not just during contests. We will take a look at variations on some familiar solutions—like bumpers and proximity detection—and will introduce some new ones. For example, we will explain how to alternate the use of a single light sensor to look down to detect the edge of the playing field and to look ahead to search for the opponent, and we will illustrate a transmission which behaves like a sort of automatic gear switch.

Although the technical aspects of building a successful Sumo robot are important, the design requires much more than simply putting together a few mechanical solutions: it requires a strategy. Will your robot be very aggressive, or do you prefer a defensive approach? It could be robust and slow, or lightweight and fast. It could be designed to actively search out its opponent, or to react when it's under attack. You cannot work at the mechanical configuration and decide how the robot should behave after it's finished. On the contrary, you have to pick up a strategy and design both the mechanics *and* the program according to it. This principle applies to any robot, but it is particularly important for Sumo robots, and it is the key to understanding this chapter: We want you to devote the proper attention to the connections between the planned behavior of your robot and the solutions you can adopt to effectively implement it.

Building a Robotic Sumo

We explained in the Introduction that when you start building a robot for a Sumo contest, you must have a strategy in mind. The process starts before building your robot. It begins by examining the rules carefully, understanding what you can and cannot do, and deciding your line of action. You must try to imagine what the opponents' strategy can be, and plan your robot to be able to resist their attacks and take advantage of their weak points. Obviously you cannot really know how the other competitors will strategize and behave, but this exercise helps you to focus on a well-defined strategy. Remember that any strategy is better than no strategy at all!

This section starts by describing a typical set of rules, which will help you in framing what a Sumo contest is, and provide a starting point in case you want to organize your own. Then we'll describe how you can tune your robot to produce maximum force, which is undoubtedly a very important component in a Sumo competition. We will also explain how to configure your robot to take advantage of some important offensive and defensive behavioral strategies.

Setting the Rules

During our Italian LEGO Users Group (ItLUG) meetings we organized robotic Sumo tournaments based on two separate sets of rules. The first set of rules states that the robots can be made out of any original LEGO piece, in any desired quantity, but that they must be within a maximum size of 32 x 32 studs and a maximum weight of 1.5kg (3lbs). In the alternative set of rules, which we called Mini Sumo, each robot may be built using only parts from a single MINDSTORMS set; there is therefore no need for size and weight constraints.

For most other aspects the two sets of rules are almost the same:

- The field is a circular or square pad with a contrasting external strip of 20cm (8 inches). Usually the pad is white and the strip black, or vice versa.
- Only two robots can fight on the field at a time. Should one robot for any reason find itself outside the field boundaries, that is, any portion of it touches a point beyond the external strip, the robot loses the round. If neither robot is eliminated within a chosen time limit (e.g., 3 minutes), the match ends in a draw.
- A robot may also be eliminated if it is overturned by its opponent or it finds itself in a situation where it can no longer maneuver.
- No "violent" behaviors are allowed. A robot can only push or lift its opponent. It is in no way allowed to damage its opponent's structure or parts.
- A robot cannot drop any part or subsystem in the field either deliberately or involuntarily. Any part found loose on the field will be removed by a member of the panel.
- The robots must be fully autonomous; any kind of remote control is forbidden.

- Every robot must comply with the limits in size and weight at the beginning of a match, but once the match starts, it can modify its own structure, perhaps extending parts so itself so its dimensions become larger than the initial specified size limits.

There are many other less important rules covering items like batteries, composition of the panel, pre-match test time, and more. Some Sumo competitions require that your robot pass an admission test: It should be able to push a block of wood out of the fighting ring. If it can't beat a block of wood, it has little chance against another robot, and this rule is meant to screen out robots too weak to enter the contest. We never enforced this rule during our Italian Sumo contests, and have to admit that it's quite possible a block of wood might have been able to win a few matches!

Maximizing Strength and Traction

The making of a strong Sumo robot requires much more than just brute force, but we cannot deny that maximizing the generated push will increase your chance of winning some matches and maybe the tournament.

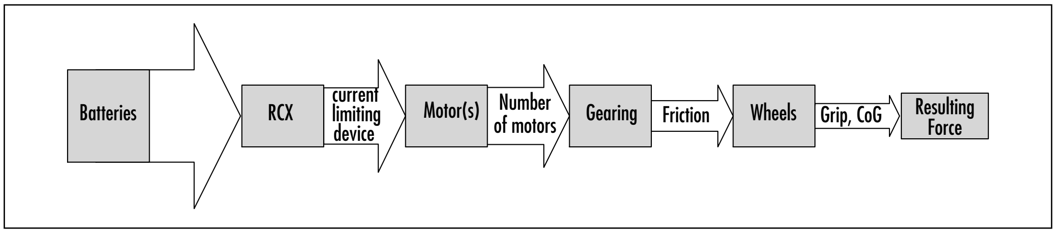
When optimizing the pushing power of your robot, the first thing you need is an objective way to measure it. Without measuring the force, the improvements you make are subjective and as a result are very inaccurate. During the preparation for the first ItLUG robotic Sumo contest, our friend and robot builder Sergio Lorenzetti suggested a simple trick based on a very common object: scales, like those used in many kitchens to weigh flour, sugar, or other ingredients.

You have to place the scale on its side on the table or the floor, possibly removing the upper tray, and hold it firmly while your robot pushes against it. You're not interested in the absolute value that the scales indicate, but rather in comparing the push produced by different setups.

There are many factors that affect this force; you can imagine a sort of path of power that goes from the batteries to the wheels, passing through the motors and the gearing, decreasing in accordance with the variables that affect each part along the path (see Figure 27.1).

We already talked about batteries in Chapter 26; the rules will hopefully specify that all competitors use the same kind of commercial batteries. Between the batteries and the motors, there's the RCX. It's worth reminding you once again, that the RCX incorporates a current-limiting device to protect the motors connected to its output ports. If the rules allow the use of extra parts and you have them, you can consider the option to connect the main motors to a battery box and a polarity switch, thus implementing the indirect control described in Chapter 3.

Figure 27.1 Limitations on Force



The number of motors influences the generated power. Simply use the maximum allowed by the rules and by your own inventory. As for the mobility configuration, the differential drive allows for the highest combination of maneuverability and simplicity. The fact that it doesn't go perfectly straight is not relevant to Sumo fighting, and the dual differential drive has no advantages in this case. On the contrary, the ability to use one motor to turn and the other to move reduces the maximum generated force.

The optimal gearing is, as always, easier to determine by experiments than by calculations. Generally speaking, the higher the reduction ratio, the higher the push, but this doesn't mean you should gear down too much. Speed has its importance (we'll explain why later in the chapter), and very high reduction ratios introduce too much friction, which uses up precious power.

Now we come to the part where you have to convert the produced torque into actual push. The wheels are a critical component: if they don't grip the pad well, the rest of your efforts will prove fruitless. This is when the scales we mentioned earlier prove an enormous benefit. By testing different kinds of LEGO wheels, you'll discover that there are significant variations in grip. The ones from the 8462 Tow Truck work particularly well, as well as the large spoke wheels contained in the MINDSTORMS kit. On no account should you use tracks. They offer extremely low grip, and almost no grip at all in the direction perpendicular to its motion. You'd have little hope at all if your opponent broadsided you—an eventuality more probable than a head-on collision.

If possible, try to test your robot on a surface similar to the contest's official pad. Different materials require different wheels. For example, the wheel having the best grip on a smooth tabletop is not necessarily the one with the best grip on a rough plywood surface.

The position of the center of gravity is also very important when it comes to friction and your wheels. Keep the COG as close as possible to the main drive axles.

Designing & Planning…

When Air Is Power

Our first robotic Sumo tournament confirmed the success of the brute force approach. Antonio Ianiero and Mario spent the night before the contest building Eolo, a monster based on the pneumatic engine of Chapter 10, supplied by the compressed air of seven air tanks manually loaded before the match. Barely a robot, Eolo was not able to turn, nor stop before the end of the pad should it miss its opponent on the first try. To reduce the possibilities of such a disaster, Eolo featured an extra large front shovel that stayed vertical until the beginning of the match (to comply with the 32 x 32 studs size limit). Thus all the RCX had to do was lower the shovel and open the valve switch. Easily the shortest program ever written for a contest!

Built mainly as a joke, Eolo won the tournament. At that time our rules stated that the robots had to start facing each other, and this is what made such a stupid machine able to overcome most of its opponents. From then on, our rules introduced a side-by-side start, with random orientation drawn by the panel.

Attack Strategies

We anticipated that force wouldn't always make the difference in a robotic Sumo contest. There are many different strategies that can affect the result and cause a robot to win out against a more powerful competitor. These include finding the enemy first, using speed as a force, using a gear switch for maximum speed and push, and other offensive tricks.

Finding the Enemy

A very important rule is: find your enemy before he finds you. This basic military principle applies to Sumo robots as well, for the simple fact that the first one to engage the other has a good chance of attacking it on a weak side. Sumo robots are generally designed to push forward, and offer much less resistance when attacked from the side or rear. In fact, they often don't even realize they're under attack, because oftentimes they're not designed to detect the enemy from behind or from the side. In such cases, you can say that three sides out of four are generally weak.

The problem is that finding the opponent is more easily said than done. Unless your rules allow custom sensors, what you have in your toolbox isn't much. Proximity detection is a good option (see Chapter 4), but remember that you also need the light sensor to detect the outer strip so as not to commit "suicide" by going outside the circle. When a single light sensor is allowed, you should alternate it face down and face front, depending on the situation. Guido Truffelli successfully implemented this trick in his robot in order to win our first Mini Sumo tournament, using only two motors as required by the rules. Figures 27.2 and 27.3 show a small assembly that explains how this works: one of the motors of the differential drive is connected to a differential gear instead of directly to the wheel. When the robot goes forward, the differential gear rotates the light sensor downward until it gets stopped. From that moment on, the sensor cannot rotate anymore, and all the power goes to the wheel. Reversing the direction of the motor—for example, to turn in place—the sensor offers less resistance than the wheel and comes up until blocked again. Guido's robot thus had two chief states: a search phase, when it turned in place with proximity detection active, and a motion phase, when it advanced with the sensor facing down.

Figure 27.2 Flipping Light Sensor Assembly (Top View)

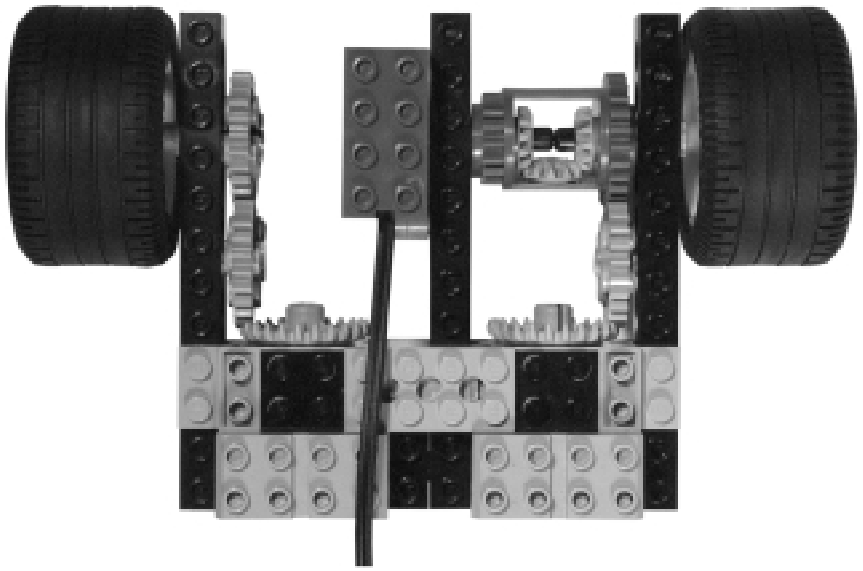
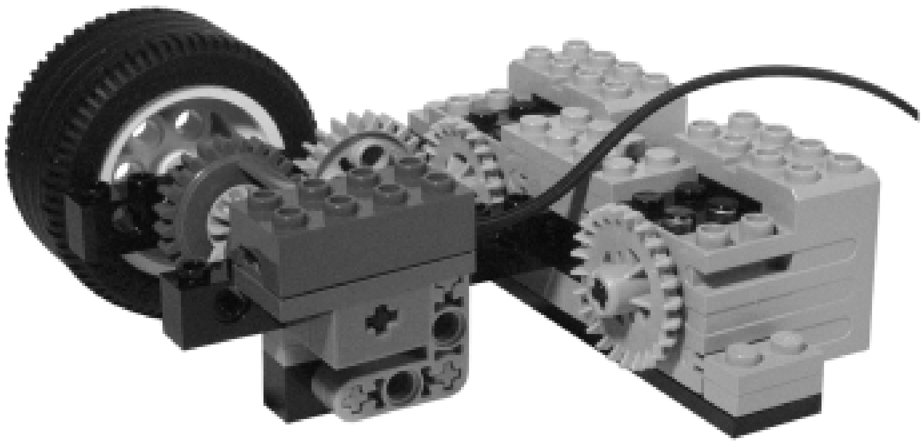


Figure 27.3 Flipping Light Sensor Assembly (Side View, Left Wheel Removed)



A simpler, but just as effective technique employs contact sensors, either in the form of bumpers or antennas. Bumpers don't require any particular trick. You simply program your robot to turn toward the obstacle instead of avoiding it. Design compact and smooth bumpers devoid of any unnecessary protrusion, to reduce the chances of getting caught on an enemy robot and dragged off the playing surface. With antennas you can use either touch or rotation sensors, the latter being able to tell you more about the direction of the opponent.

Using Speed

Speed is an extremely important factor in the search for the enemy. Imagine two robots running freely on the Sumo field, simply going straight until they find the border and change direction randomly. Supposing that they have different speeds, the faster of the two has a much greater chance of intercepting the other. For this reason, it's important not to have too a slow robot. Find a compromise between pushing ability and speed.

Carrying this to the extreme, Roberto Francia made speed the main weapon of his robot Lancillotto (Lancelot). Crashing into the opponent at high speed, the robot used its momentum instead of its strength. The energy released in the impact made the opposing robot lose contact with the ground, pushing it back a short distance. One assault after the other, Lancillotto charged repeatedly, like a ram, until its poor victim was pushed off the field. (Incidentally, Roberto's robot was fast, but not so fast as to be considered illegal in terms of the rule against destroying the opponent!) Ranking second at his first contest, Roberto's robot demonstrates that even beginners may teach the "experts" something.

NOTE

Momentum is a physical quantity defined as the product of mass times velocity. You can understand what it means through an example: You face a person of your same weight and build that's trying to knock you down. If you're both stationary, you have a good chance to resist. If, on the other hand, you are stationary and the other is running towards you, you will very likely go down.

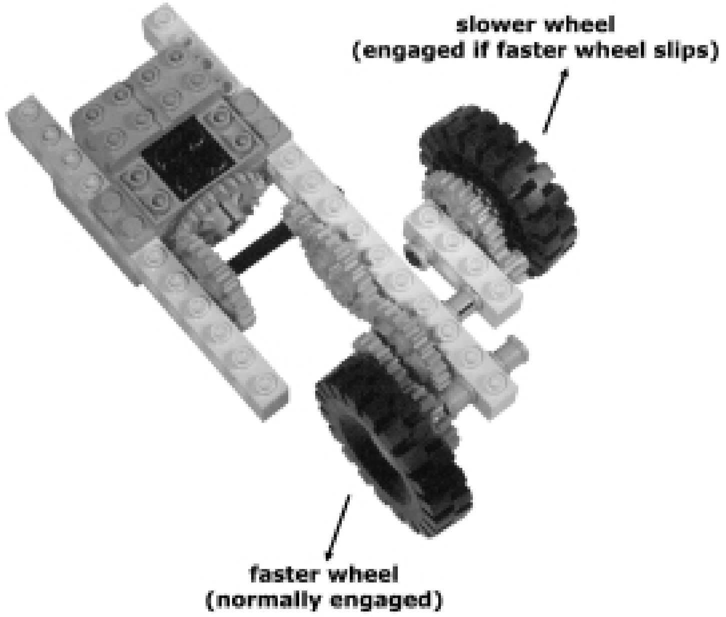
Using a Transmission

Other robots use a transmission to get the best of both worlds: fast speed during the search phase, and maximum push after the engagement. Our robot Golia II used a transmission very similar to that described in Chapter 14, based on the special transmission ring. But Sergio Lorenzetti demonstrated during a contest that it's possible to make a sort of automatic gear shift even inside the strict rules of Mini Sumo. Look at the assembly in Figure 27.4, it's not very solid, but explains the principle: The wheel on the right in the picture is geared with a shorter ratio than the main one, and during normal motion it slips a bit because the robot is moving faster than the speed of the idler wheel. When the robot slows down for any reason, the faster wheel slips, and at that point, the slowest one grips. Since it's mounted on a short independent beam with a free end, part of the torque pushes the wheel down and consequently lifts the robot. Remember to add a part to stop that short beam when almost vertical. (We didn't include it in the picture because we wanted to keep it clearer.)

Other Sumo Tricks

There are many other tricks that prove useful during a Sumo contest. The ones most often used are meant to lift the opponent, thus getting two positive effects: reducing or canceling the grip of its wheel and transferring part of its weight on your robot. This class of method includes at least two large families, one based on inclined planes and the other on counter-rotating wheels.

Figure 27.4 An Automatic Gear Switch Assembly



An inclined plane works like a wedge that slips under the enemy robot. It can have the shape of some small slopes placed at the front side of the robot, or of a large inclined surface that covers the whole robot. In this latter case, a LEGO baseplate is the better choice: mount it studs-down and you'll have a very smooth top surface to wedge under your opponent.

Counter-rotating wheels are very effective, too, but require an additional motor to operate them. Be sure they don't touch the ground though, otherwise they'll counteract the forward motion of your own robot! The combined effects of the front wheels with the push of the robot may even overturn the opponent, a spectacular but rare event.

Getting Defensive

So far we have discussed attack strategies, but protecting the weak sides of your robots is important as well.

Every active defense system relies on the fact that you know what's happening around you, and require some sensor to detect a possible attack. Depending on the rules of the tournament, you might find yourself dealing with a limited number of input ports, requiring that you carefully plan out how to allocate them in regards to your navigation, attack, or defense subsystems. The simplest detecting system is a sort of large bumper that covers a whole side of the robot. If you have enough touch sensors, you can connect them in parallel so as to monitor three sides with a single port. In this case, you'll know you're under attack but won't be able to tell what side it's on.

When you detect you've been tackled, you have the option of either escaping or facing your enemy. The first choice is best when fighting a slow, strong opponent, while the second works well when it's your robot that has a strong push (though it's not always easy to turn in place when being pushed). Some rules allow the competitors to use more than one program. Take advantage of this opportunity by preparing different versions to implement different strategies, then select the one most suitable when you know which robot you'll be facing in a given match.

Also consider passive defense systems, the kind that doesn't require any sensor or port. The more obvious defense mechanisms revolve about the shape and size of the robot itself. A smaller robot offers less surface area to an opponent than a larger one, and though a triangular shape is more difficult to build, it's also more difficult to catch. Make the perimeter somehow convex if you can, so as not to offer any holds that will help your opponent. Clearance from the ground is important for the same reason: it reduces your enemy's chances of wedging itself under your robot.

More sophisticated passive defenses include protruding beams or axles meant to keep the enemy away from your robot's vital organs, freewheeling vertical wheels on the sides to neutralize lifting wheels, and free horizontal wheels to allow your robot to slip away when engaged on one side.

Testing Your Sumo

This phase is crucial to a good result. Start testing your robot on a pad similar to the tournament's to make sure it doesn't do stupid things in the most common situations. It should detect the edge of the field when reaching it from any angle: You can't imagine how many robots won a match because their opponents killed themselves!

When everything works well, you can start more advanced testing. You really need a sparring partner, but it need not be a second robot. Many reasons suggest you use a fake robot as a sparring partner, something you can move by hand to create any situation you want. (Using a real robot, you'd end up testing both instead, plus you risk not being able to control specific scenarios.) A simple box does the trick, or a heavy book. Start by leaving the fake robot still in the middle of the field, and see what happens. Your robot should find it, sooner or later, and push it off the pad. When this works, move the fake robot yourself to test the defensive strategy of your robot, and its behavior at the edge of the pad, the most dangerous area.

Remember that the perfect robot doesn't exist. For any winner of a contest, it's possible to design an "antidote" robot capable of beating it. You just have to accept some compromises in your project and make some assumptions about your opponents, hoping they won't prove too far from reality.

Summary

If you have no previous experience in robotic Sumo, you may think of it as a competition based solely on brute force. We must confess that we also had many preconceptions our first time out at a competition of this kind, but had to change our mind. Force is indeed important, but it typically proves useless when up against a good deal of intelligence.

These competitions have nothing in common with the kind of events that feature radio-controlled machines, called "robots," that try to destroy each other. These are not robots, simply because they totally lack a distinctive robot property: autonomy.

The first important lesson that this chapter teaches is that you must design your robot with a strategy in mind, choosing the configuration that best suits your goal. Start examining the rules, then make a hypothesis about your opponents and devise a strategy to beat them. Your opponents may be very different from how you imagined them, but this is not important—what's important is that you build and program your robot to be consistent with the strategy you chose. A perfect robot doesn't exist; in fact, situations in which robot A beats robot B, which then beats C, which in turn actually beats robot A, are very common in contests. And they're what make contests so interesting and instructive.

We hope you also understand the second important message of this chapter: When building and programming your robot, make reliability your first priority. If you can beat a block of wood in a Sumo match, you're halfway to success!

Chapter 28

Searching for Precision

Solutions in this chapter:

- **Precise Positioning**
- **Finding and Collecting Things**
- **Playing Soccer**

Introduction

This third and last chapter of Part III is dedicated to contests based on some specific ability. Occasionally, speed is important, too, but not as much as in the competitions described in Chapter 26, and while two or more robots may perform at the same time on the same field, physical contact is not the main goal as was the case in the competitions of Chapter 27.

These abilities include what in Part I we described as the most challenging tasks for MINDSTORMS robots: finding and grabbing objects (Chapter 11), and knowing precise positioning (Chapter 13). The need to use them in a contest makes your mission even more demanding: You must consider the interference that comes from sharing the playing field with other robots, that may voluntarily or involuntarily disturb the action of your robot. The recipe for success is the same as proposed in the previous two chapters. This applies to any kind of contest: study the rules, define a strategy, make a few assumptions about the opponents, build a prototype, experiment with it, test the software carefully … and rebuild everything from scratch until you are satisfied. In other words, you need some ideas, some skills, and lots of patience!

The last challenge described in the chapter—Soccer—shows an interesting variation on the theme of object finding: It is the object itself—the ball—that guides the robot to its position, through the emission of IR light. You will discover that this change in the nature of the problem is enough to simplify the robot's requirements considerably, to the point where its software isn't so different from that used to implement the simple light following algorithm of Chapter 18.

Precise Positioning

The challenge of precise positioning requires that your robot go, or return, to a specific point. The robot whose degree of error is smallest, wins. You can define many implementations of that simple statement, each one with its own peculiarities. As always, even a small change in the rules can have radical effects on the difficulty of the challenge. A very simple version is: Starting from a predefined point, the robots must move forward until they hit an obstacle, then turn in place 180° and return to the spot where they began. The obstacle will be the same, at the same distance from the start for all the robots, and the contest may require many runs with different distances. It's important that the rules specify that the robots must turn 180° before returning to the starting point, otherwise most of them will simply go in reverse!

If you're the one who decides the rules, calibrate the difficulty of the contest by setting the limit on the number of parts admitted. For example, a dual differential drive like our LOGO Turtle can be very precise, but requires two differential gears and a rotation sensor. Limiting the equipment to just the MINDSTORMS set will make the contest fair to a larger number of participants, but more difficult.

Have you any initial ideas about how you would make a precise run-and-fetch robot with only MINDSTORMS parts? At this point in the book you should have many ideas, however, let's do this exercise together. Starting from the mobility configuration, you can proceed by a process of elimination: steer, tricycle, and synchro drives don't turn in place; skid-steer does but introduces track slippage, which is very bad for precise positioning; dual differential drive won't work because of the lack of a second differential gear, thus you end up with the tried-and-true differential drive, with its handicap in going straight.

The first approach that comes to our mind requires you emulate two rotation sensors with the touch sensors, and monitor the turns of the right and left wheels so as to keep them synchronized. (You still have the light sensor for the bumper!) Getting ideas from Chapter 8, you can also use a differential gear to monitor the difference in speed between the two wheels (Figure 8.2). The light sensor could face a sort of black and white disc connected to the differential, so you will drive the robot more or less like a line follower, slowing one of the motors when it reads too black and the other when it reads too white.

With either one solution or the other, you may manage to go straight, but you still have to turn precisely 180°. This is the most critical point, because even a small error in the angle will leave your robot very far from the starting point. Do you remember what we said about tuning the turning ability of the Logo Turtle in Chapter 23? Use the distance between the wheels to adjust the turning angle so the U-Turn is equal to a whole number of counts of your sensor. Thorough testing is, as always, your ticket to success.

A challenge based on positioning may be made significantly more complex by simply adding more segments and checkpoints to the route. For example, instead of a round trip you can prepare a triangular path—ask the robots to stop in any vertex and measure the deviation between their actual position and the expected one. Each robot should have an easily identifiable part to use as a reference point for measuring the starting and ending points of the journey—for example, a vertical axle with one end very close to the ground.

Finding and Collecting Things

In 1999, Joel Shafer proposed a tough challenge unlike any that had been presented in the LEGO community previously. In it, a robot had to navigate the room, pick up three empty soda cans, and return to the starting point. Active navigational aids, like beacons, were not allowed. Joel's was a remote challenge, that is, there wasn't any specific place or time for the event, rather the participants had to send pictures and documentations of their robot and its program.

Joel's challenge was new and tough; it created a lot of traffic in the lugnet.robotics newsgroup, the place where MINDSTORMS fans virtually meet to discuss ideas and techniques. The challenge was at the same time engaging and daunting because of the involved difficulties, and originated many interesting discussions about navigation and search techniques.

A few months later, Richard Sutherland was declared the winner. The challenge aside, Richard proved that managing those kinds of difficulties were not beyond the range of MINDSTORMS robotics, which opened the way to similar competitions of a very demanding nature. We'll explore some variations on these types of competitions in the following sections, and describe some of the navigation and search techniques you could employ.

Maxwell's Demons

David Schilling devised a very interesting robot competition he called "Maxwell's Demons," paying homage to James Clerk Maxwell's famous physics scenario in which a demon separates hot molecules from cold ones in a room, supposedly contradicting the second law of thermodynamics. In David's challenge, each robot must distinguish black LEGO cubes from white ones, and push as many white cubes as possible to its side of the playing field while at the same time pushing as many black cubes as possible to its opponent's side.

This is a very complex challenge that requires your robot be able to:

- Navigate the playing field.
- Find the cubes.
- Recognize their color.
- Capture them.
- Push them into the proper territory.

Though a very attractive challenge, we at ItLUG decided to start with a slightly simplified version where the colors of the cubes don't matter, just their numbers.

Stealing the Cube

During a given time period (three minutes) each robot must try and accumulate as many cubes as possible on its side of the playing field, taking them either from the central border line where they are placed at the beginning or from the opponent's field. The robot can use only one RCX and one light sensor, but there is no limit to the number of other original LEGO parts and sensors that can be used; non-LEGO custom devices are not allowed.

As in David's original rules, the cubes are made of six 2 x 4 bricks arranged in three interlaced layers of two bricks. The cubes are topped with tiles so as to make them perfectly smooth and cubic.

Our field has a white side, a black side, and a gray border. We decided on a very restrictive size limit for the robots: They must fit into a space 20 x 20 studs square. Other rules state that the robots must push or hold just one cube at a time; they can push more than one if this happens by chance, but cannot be explicitly designed to do this. Each match starts with seven cubes placed along the borderline and the robots situated on their respective sides.

This challenged immediately proved very difficult. The first obstacle we had to face was navigating a three-color playing field. As we explained in Chapter 4, the LEGO light sensors reads the average reflection of a small area, thus making it impossible to tell a black-white borderline region from a gray one. How would you have done it? The simplest answer required that the competitors increase the complexity of the code a bit and test two or more closed readings of the sensor, accepting the value only when they were stable. Theory is one thing, practice another: keeping the robots inside the field was not easy, but almost everybody succeeded in the task.

The next difficulty came from the cubes. Chapter 11 describes how to use proximity detection to search for objects, but this technique is not applicable to two robots at the same time, because their IR emissions interfere with each other and alter the reliability of the readings. Add to this the fact that only one light sensor was allowed, which was necessary to navigate the field, and this definitely excludes proximity detection. Everyone decided to use touch sensors to monitor collisions with the cubes—in other words, the robot would navigate the pad until they ran into a cube. Unfortunately, the objects couldn't be detected with a simple front bumper, because they were too lightweight and slippery to exert any pressure on the touch sensor. We saw many different techniques offered as solutions to this: Our robot, the one shown in Chapter 11 (Figure 11.12), used a top bumper activated by the height of the cube. Other competitors adopted different harvesting systems—for example, a sort of gate that closed when a cube was inside, or a lever that blocked the cube against one side of the robot.

Most of the robots based their search for the cubes on a purely random navigation, while the more sophisticated ones used a limited knowledge of their position in regard to the borderlines of the field, applying the dead-reckoning techniques described in Chapter 13. The idea of using relative positioning methods was good, and the colors of the different areas of the field provided some external references to reset the errors that internal odometry would inevitably accumulate. If a robot knows its position, it can more efficiently scan the field, passing precisely once—and only once—through any point. Unfortunately, the approach didn't end up working very well, and proved no better than random navigation. The problem came from collisions: each time a robot was hit by the opponent, it got its position and orientation slightly changed, and this was enough to make the internal position information totally unreliable.

Inventing…

An Amazing Strategy

For the Stealing The Cube competition, Guido Truffelli once again devised a winning strategy nobody else had thought of. His robot went straight in order to find the borderline between the fields where the cubes had been set up, then turned 90 degrees and followed the line at the same time, kicking the cubes into its field. After a few seconds, most of the cubes were inside its territory, and the opponent had to fight a hard battle.

Not satisfied with this initial advantage, Guido's robot continued relentlessly combing the opponent's field with lethal precision, promptly carrying any captured cube to its side. It proved by far to be the strongest competitor.

This once again proves how different sets of rules affect a competition: If the starting position of the cubes had been random, or if launching the cubes had been forbidden, Guido couldn't have used his trick.

Variations on Collecting

The theme of finding and collecting objects admits infinite variations. The size and shape of the objects have a strong influence on the architecture of the robots. Marbles, for example, are quite different from empty soda cans or LEGO cubes, because they tend to roll away at the slightest touch, thus requiring a very cautious approach.

Also, instead of placing two robots in the playing field at the same time, you can organize the challenge so just one robot runs at a time, evaluating its performances against time. For example, counting the number of objects collected during a prearranged interval, or measuring the time it needs to harvest all of them.

Playing Soccer

Soccer, in an extremely simplified form, is a rather suitable game for small robots. By "simplified" we mean that the robots don't actually kick the ball but instead push it toward the goals. Other simplifications include the absence of fouls, throw-ins, and all other rules except for the one that says each goal scores a point.

The required abilities would be similar to those explained earlier about finding and collecting objects, with the difference being that there's only one object, the ball, for two teams who must take it and score a goal. However, with the adoption of a special field and a special ball, you can trim down the essential skills to a level where even an inexperienced child, with some tips, can program a basic robot to play the game.

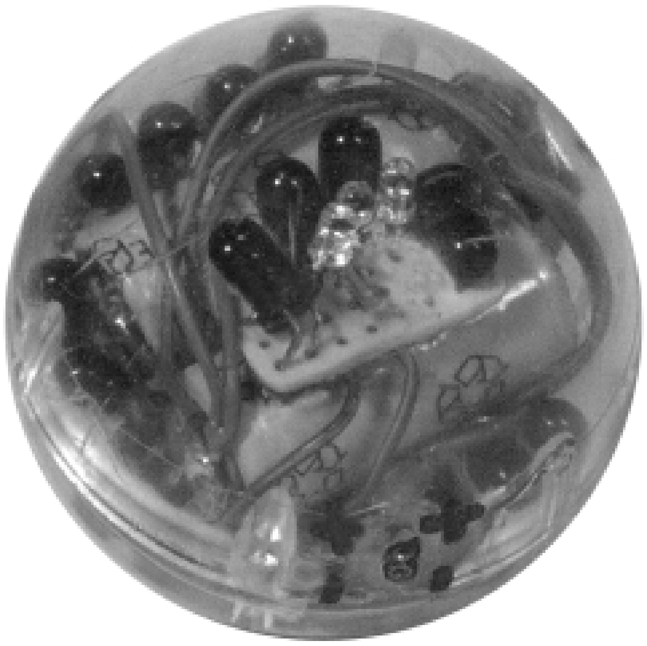
In 1999, during the first Mindfest at MIT, we saw a very entertaining version implemented by Henrik Hautop Lund and Luigi Pagliarini from the LEGO Lab at the University of Aarhus. The field was covered with a simple linear gradient, black at one end and white at the other, and there were raised edges all around, with gaps to represent the nets. The players featured a single RCX with five sensors: three light and two touch (the light and touch sensors were combined in pairs on the same port as described in Chapter 4). One light sensor, facing down, allowed simple navigation on the field, while the other two, facing forward, were aimed at finding the ball. The touch sensors were connected to bumpers in order to detect collisions with the edges or with other players.

The ball was a special, active ball: two to three inches in diameter, made of clear plastic, it was filled with rechargeable batteries and IR LEDs so to be easily detected by light sensors. The reasons that Lund and Pagliarini used two front light sensors is that they were able to program a simple method of finding the ball. With two sensors, they could make the robot turn in place until both the sensors read a very high value, and that was the direction of the ball. The standard chassis of the robot and simple commands allowed young children to program their soccer players with different strategies without having to worry too much about the details.

We replicated this setup with Marco Berti a few months later, scheduled to be shown during an exhibition in Italy. Like the originals, the robots were differential drive, but with two light sensors instead of three. The front of the robots was shaped so as to push the ball while going forward.

Building your own soccer player is not a difficult task—you'll find the greater challenge is in the construction of the ball. We used one of those clear plastic balls found in toy vending machines for children, the kind that usually have a little prize inside. The making of the inner electronics requires some experience in soldering and in assembling small parts. Fit the inside with as many IR LEDs as possible, and with rechargeable batteries of your choice. Drill small holes in the surface of the ball to include a female connector and a small switch so you can turn it on and off and recharge the batteries without opening it. This is the most difficult part of the job, because you must keep the surface smooth and the COG as close as possible to the center of the ball so it rolls smoothly (Figure 28.1).

Figure 28.1 Marco Berti's IR Ball



The field may be improved using the two attractors gradient described in Chapter 13. This geometric pattern has the property that, if you follow the darkest path from any of its points, you arrive at the black attractor, while if you choose the clearest path, it drives you to the white one. Using such a pattern, it's very easy for the robots to reach the goals that correspond to the attractors by employing a very simple navigation algorithm.

The program is not too difficult to write. Make your robot turn in place searching for the ball until it finds it (the algorithm is actually very similar to the one we described in Chapter 18 to implement light following). If it doesn't, make it move a bit in any random direction and look around again. When it finds the ball, it moves forward to catch it and then starts going toward the opponent's net.

Summary

The competitions we talked about in this chapter require some abilities that, in Part I of the book, we described as the most challenging to implement: finding objects and knowing where you are.

If these activities demonstrated here prove difficult to implement when you build a robot for yourself, situating them in the context of a competition makes your mission even more difficult. This happens because you must push the performance of your robot to its maximum. You have to consider all the details, optimize the software, and reach the highest possible level of reliability. However, most of the hitches derive from the fact that your robot is not alone in the field, and the interference with its opponent will disturb its behavior. For example, IR proximity detection can't be used by two robots at the same time, and dead reckoning calculations to estimate the position of the robot may be frustrated by collisions against the opponent.

The Soccer competition we described in this chapter is a good example of how a few changes can radically affect the solution to a problem. It also shows the practical application of two techniques described in Chapter 13 regarding absolute positioning: the use of an IR beacon, and a pad with a special pattern that eases navigation. In fact, the IR ball guides the robot to it, and after the robot gets the ball, it can find the way to the goal following the gradient on the pad. This simplification is so effective, that the software of the robot becomes similar to a simple algorithm we covered earlier in the book: light following.

In this kind of competition, the contest sponsor can also suggest, or impose, a standard chassis built with just MINDSTORMS parts. This would move the challenge of the contest completely to the software side.

Cube collecting or soccer playing requires a complex behavior made of many different actions that need to be coordinated together well. If you decide to take up the challenge, we suggest you think both your hardware and software in terms of subsystems. This way they will be easier to test, debug, and maintain. Write your program with a top level supervisor that manages small subroutines corresponding to the basic actions the robot has to perform: navigation, object detection, and object collection. Mastering this kind of challenge won't be easy, but as with most difficult things in life, your satisfaction will be directly proportional to the effort you expend!

Appendix A

Resources

Introduction

There's quite a large amount of reference material to be found regarding MINDSTORMS inventions, including some very good books, and hundreds of Internet sites that cover specific topics and show interesting models. In this appendix, you'll find a section about books, another one about links of general utility, and a section specific to each chapter of this book (many of the quoted sites pertain to more than a single chapter topic, so browse through them all). We apologize in advance for the significant number of interesting sites that we surely (and unintentionally) omitted from the list.

Every link of this appendix has been checked, but as you know, the Internet is a dynamic animal and we cannot guarantee they will be still valid at the time you read the book. If you find any broken links, use the descriptive information we provided beside each site address to hunt for it using your favorite search engine.

A few of the links point to commercial sites, or to sites that, besides providing information about the making of some custom part, also sell a kit or the finished product. We have no direct or indirect interest, nor any connection with them; we included the links simply to help the reader.

Bibliography

The Unofficial Guide to LEGO MINDSTORMS Robots, by Jonathan B. Knudsen; O'Reilly & Associates, 1999. The first to appear on the market, Jonathan's book is still a very good resource for introducing readers to the MINDSTORMS world. It covers many topics, from construction techniques to programming with different languages.

Dave Baum's Definitive Guide to LEGO MINDSTORMS, by Dave Baum and Rodd Zurcher (Illustrator); Apress, 1999. Dave is the creator of NQC, the most successful alternative programming environment for the RCX. In this book, he not only explains how to use NQC, but also explores many building and programming techniques.

Extreme MINDSTORMS: An Advanced Guide to LEGO MINDSTORMS, by Dave Baum, Michael Gasperi, Ralph Hempel, Luis Villa; Apress, 2000. Four gurus of the independent MINDSTORMS community introduce you to the secrets of NQC, legOS, pbForth, and to the making of custom sensors.

Creative Projects with LEGO MINDSTORMS, by Benjamin Erwin; Addison-Wesley, 2001. Ben invites the reader to be creative, to explore different approaches, and even use different materials. He also covers topics like ROBOLAB, not covered in any other book.

Joe Nagata's LEGO MINDSTORMS Idea Book, by Joe Nagata; No Starch Press, 2001. Joe is without a doubt a great designer. In his book, he steers you step by step through the building of some instructive and efficient models.

LEGO MINDSTORMS: The Master's Technique, by Jin Sato; No Starch Press, 2001. This is a great book, containing both general building suggestions and programming tips. It also includes step-by-step instructions on how to replicate MIBO, his famous robotic dog.

General Interest Sites

LEGO MINDSTORMS (http://mindstorms.lego.com)
The first site to mention is, of course, the LEGO MINDSTORMS official site. It contains tons of stuff: technical tips, a gallery of inventions, events, contests, answers to frequently asked questions (FAQ), and more. The official LEGO MINDSTORMS FAQ site is: http://mindstorms.lego.com/products/whatis/faq.asp.

LUGNET (www.lugnet.com)
The LEGO Users Group Network (LUGNET) is the most comprehensive Internet resource for LEGO, and it's difficult to describe in a few words. It features a database containing all the LEGO sets ever released, as well as a reference list citing all the single LEGO parts. But, more importantly, its newsgroups are the meeting point of LEGO fans of any age and from any part of the world, and it's one of the friendliest places on the Internet. Don't miss the LUGNET robotics newsgroup (http://news.lugnet.com/robotics), the place where you can ask any number of questions and be answered with completeness, competence, and patience.

LEGO Parts Reference (http://guide.lugnet.com/partsref/)
Created by Steve Bliss and hosted on LUGNET, this database contains information, images, and links for many LEGO bricks and other elements.

NOTE

The official Web site for this book is at www.syngress.com/solutions. Check it out for additional MINDSTORMS-related features, resources, and downloads, including more NQC code, MIDI conversion files, positioning grids, film clips, and lots of new photographs. You can also post questions to the authors and editors, see the front page of this book for details about the site.

Brickshelf (www.brickshelf.com)
Brickshelf is a site that offers everybody the extraordinary opportunity of having free space to show off his or her own LEGO models.

Fred Martin's Unofficial Questions and Answers about MIT Programmable Bricks and LEGO MINDSTORMS (http://fredm.www.media.mit.edu/people/fredm/mindstorms/index.html)
Fred Martin tells the story of the Programmable Brick and provides some other useful information about the RCX.

LEGO MINDSTORMS Internals (www.crynwr.com/lego-robotics/)
Russell Nelson maintains a page that contains many technical details about the MINDSTORMS system as well as many useful links.

Artificial Intelligence and Machine Learning (www.bvandam.net)
Bert van Dam's site is a mine of information about artificial intelligence in general. If you find the subtle link to Miscellaneous | General Information, you will discover a whole world of LEGO projects!

Andy Bower's LEGO Robotics Wiki (www.object-arts.com/wiki/html/Lego-Robotics/FrontPage.htm)
Andy Bower's dynamic site is where anybody can look for answers or add his own contribution. Quoting Andy, "The important thing to remember is that this Wiki site is a growing body of knowledge and you are responsible for how useful it becomes."

Chapter 1 Understanding LEGO Geometry

LEGO On My Mind (http://homepages.svc.fcj.hvu.nl/brok/legomind)
Don't miss Eric Brok's site, filled with explanations and suggestions. LEGO geometry is just one of the many topics covered.

Length of Diagonal Spreadsheet (http://news.lugnet.com/org/us/smart/?n=37)
Gustav Jansson created a spreadsheet that shows the length of the diagonals in terms of LEGO units.

The Brick Bakery (http://web2.airmail.net/sjbaker1/lego)
Steve Baker's site contains two useful pages about LEGO dimensions and gear spacings.

Chapter 2 Playing with Gears

Fred Martin's The Art of LEGO Design (ftp://cherupakha.media.mit.edu/pub/people/fredm/artoflego.pdf)
A very good primer about LEGO geometry in general and gearings in particular. It also contains many useful design ideas.

Sergei Egorov's LEGO Geartrains (www.malgil.com/esl/lego/geartrains.html)
Sergei Egorov has prepared a useful table showing possible combinations of gear wheels, with resulting ratios and working distances.

Mike Fusion's TECHNIC and MINDSTORMS Page (http://odin.prohosting.com/mrplanet)
In Mike's site, you will find a differential made with ordinary gears, an adder-subtractor and a solution for a small planetary gear.

Chapter 3 Controlling Motors

Motor Mis-Match (http://news.lugnet.com/robotics/?n=13927)
Steve Baker's report about his test on a group of TECHNIC motors.

LEGO Dacta eLAB (www.lego.com/dacta/elab/default.htm)
Contains the technical specifications of the new TECHNIC motor (follow the link to Technical Specs and then to Motors).

LEGO Motors (www.enteract.com/~dbaum/lego/motors.html)
Dave Baum's page about LEGO motors and their features.

Chapter 4 Reading Sensors

MindStorms RCX Sensor Input Page (www.plazaearth.com/usr/gasperi/lego.htm)
Michael Gasperi's super-site about MINDSTORMS sensors—the starting point for any investigation about this component. It also contains Brian Stormont's suggestion to combine a touch sensor and a light sensor on the same port, and Tom Schumm's trick to connect touch sensors in the AND configuration.

MINDSTORMS Light Sensor Trick (www.hempeldesigngroup.com/lego/lightsensor/index.html)
Here, Ralph Hempel explains a way to improve the reading range of the LEGO light sensor.

LEGO Robotics (www.mop.no/~simen/lego.htm)
The LEGO section of Simen Svale Skogsrud's site is definitely worth a visit. It contains details about using the FOS unit as a rotation sensor, discusses proximity detection employing the light sensor, and outlines some interesting projects.

Rotational Sensor & Gearing Down (http://news.lugnet.com/robotics/?n=14074)
Steve Baker's original post reporting his test about rotation sensors.

Chapter 5 Building Strategies

LEGO Engineering (http://british.nerp.net/lego/index.html)
This helpful site includes a Building Tips section, too.

Reinard's LEGO Building Tips (http://british.nerp.net/lego/index.html)
Reinard van Loo's page contains building tips and tricks.

Ldraw.org (www.ldraw.org)
LDraw is a freeware program that can create LEGO models in 3D on your computer screen. Did you ever dream of working with an unlimited supply of any LEGO part in any color?

MLCad (http://mlcad.ldraw.org)
Michael Lachmann's MLCad is a great (and free) CAD program for creating LEGO-like building instructions of your own models. The MLCad site has been recently incorporated into the Ldraw.org domain.

Chapter 6 Programming the RCX

RCX Internals (http://graphics.stanford.edu/~kekoa/rcx)
Kekoa Proudfoot documents all the internals of the LEGO firmware and ROM routines. He made the development of firmware like legOS and pbForth possible. In his LEGO MINDSTORMS Internals page (www.crynwr.com/lego-robotics), Russell Nelson properly lists him under the heading *Heroes*!

NQC – Not Quite C (www.enteract.com/~dbaum/nqc/index.html)
Dave Baum's NQC site contains the compiler and the documentation.

legOS (http://legos.sourceforge.net)
The legOS homepage.

pbForth (www.hempeldesigngroup.com/lego/pbFORTH/index.html)
Ralph Hempel's programmable brick FORTH (pbFORTH) for MINDSTORMS page.

Gordon's Brick Programmer (www.umbra.demon.co.uk/gbp.html)
With its graphic-textual interface GBP is a sort of bridge between RCX Code and the pure textual programming environments.

Bot-Kit (www.object-arts.com/Bower/Bot-Kit/Bot-Kit.htm)
An interface to programming the RCX in Smalltalk (based upon Dolphin Smalltalk).

QC (http://digilander.iol.it/ferrarafrancesco/lego/qc/index.html)
Francesco Ferrara's QC, a mini OS (no multitasking) meant as an interface between C code and the ROM routines of the RCX.

Brick Command (www.geocities.com/Area51/Nebula/8488/lego.html)
A simple textual programming language that incorporates a complete IDE.

ADA for MINDSTORMS (www.usafa.af.mil/dfcs/adamindstorms.htm)
An ADA pre-processor to NQC. Also consult some of the documentation at www.faginfamily.net/barry/Papers/AdaLetters.htm.

LEGO Robot Pages (www.cs.uu.nl/people/markov/lego)
The site of the original RCX Command Center, a very good IDE for NQC originally developed by Mark Overmars but not updated to the current version (see Bricx Command Center).

Bricx Command Center (http://hometown.aol.com/johnbinder/bricxcc.htm)
Formerly known as the RCX Command Center, and based on Mark Overmars' original source code, John Hansen's BricxCC supports all the LEGO Programmable Bricks and introduces many new and interesting features. If you use NQC on a PC platform, this is a "must have."

VisualNQC (http://home.hetnet.nl/~myweb1/VisualNQC.htm)
Ronald Strijbosch's Visual NQC has its roots in the RCX Command Center, but is completely rewritten in Visual Basic. A very functional and complete IDE to NQC.

NQCEdit (http://hem.passagen.se/mickee/nqcedit)
Another front-end IDE for NQC, written by Mikael Eriksson. Currently less sophisticated than the RCX Command Center and Visual NQC, it's indeed an effective and solid alternative.

NQC API Programmer's Guide (www.cybercomm.net/~rajcok/nqc)
Mark Rajcok's guide to NQC API lists all NQC functions, their syntax, their supported programmable bricks and a few examples.

MindScope (http://baserv.uci.kun.nl/~smientki/Lego_Knex/Lego_electronica/Mindscope.htm)
Stef Mientki's graphing utility is able to continuously monitor the sensors and produce a chart from the sampled values.

Programming the LEGO Microscout (http://eaton.dhs.org/lego)
Doug Eaton explains how to program the Microscout through bar codes.

General Paranoyaxc RCX Tools (www.rainer-keuchel.de/rcx/rcx.html)
A package that contains a port of the NQC compiler, a simple editor, and a remote control program to access your RCX from WinCE platforms.

EmulegOS (http://sourceforge.net/projects/emulegos)
A LegOS emulator, which lets you run and debug your LegOS programs on your Win/Linux PC. Started by Mario Ferrari and Marco Beri, emulegOS is currently an Open Source project managed by Mark Falco.

LegoSim (www.informatik.hu-berlin.de/~mueller/legosim)
LegoSim is a Unix-based Simulator for LegOS with an Applet-GUI, written by Frank Mueller, Thomas Röblitz, and Oliver Bühn.

WinVLL (www.research.co.jp/MindStorms/winvll/index-e.html)
A simple tool by Shigeru Makino to control and program the MicroScout from a PC.

TCL – RCX (www.linux.org/docs/ldp/howto/mini/Lego/tcl.html)
Laurent Demailly and Peter Pletcher's TCL RCX can either compile a TCL script into RCX bytecode, or it can remotely control the robot via either a script or an interactive TCL shell.

Reactive Languages and LEGO MINDSTORMS (www.emn.fr/richard/lego)
Martin Richard's Web site about using synchronous languages (Esterel, Lustre, Grafcet) to play with the LEGO MINDSTORMS kit.

Chapter 7 Playing Sounds and Music

Guy's LEGO page (www.aga.it/~guy/lego.htm)
Guido Truffelli's page contains his MIDI2RCX and WAV2RCX utilities, as well as their new graphics interface RCX Music Studio.

Music-Robots (www.daimi.au.dk/~ocaprani/SmallCar.dir/Main.html)
A car that makes musical sounds through a speaker connected to the output port of an RCX.

Note Names, MIDI Numbers and Frequencies (www.phys.unsw.edu.au/~jw/notes.html)
A table that gives the frequency of any standard keyboard note and its midi number.

Chapter 8 Becoming Mobile

Robo-Rats Locomotion Page (www.cs.dartmouth.edu/~robotlab/robotlab/courses/cs54-2001s/locomotion.html)
This complete excursus about robotic architectures describes the pros and cons of each platform and also covers a few types not discussed in this book (e.g., the pivot drive, the articulated drive).

The Straight and Narrow (www.oreillynet.com/pub/a/network/2000/05/22/LegoMindstorms.html)
Jonathan Knudsen's article about using a differential drive to go straight.

Doug's LEGO Robotics Page (www.visi.com/~dc/index.htm)
Many special mobility configurations appear in Doug's site: a tri-star wheel drive, a Killough's platform, and a couple of synchro drives.

LEO & LEGO (http://carol.wins.uva.nl/~leo/lego.html)
Leo Dorst's LEGO page contains many useful tricks and explains how to build a Killough's mobile robot platform.

Synchro Drive (www.restena.lu/convict/Jeunes/SynchroDrive.htm)
Part of the Boulette's Robotics site (see heading later in this appendix), this page describes the steps in building a LEGO Synchro Drive.

S18 Details (www.geocities.com/mario.ferrari/s18/s18.html)
From our site: Our second synchro drive features a rotating bumper.

Macs Robotics Page (http://homepages.fbmev.de/bm957542/Robotics/index.html)
Step by step instructions to build a nice and compact Killough's platform.

Ackerman (http://users.ids.net/~bdfelice/ackerman.html)
Here you can find the history of Mr. Ackerman and the explanation of his steering system.

Chapter 9 Expanding Your Options with Kits and Creative Solutions

Technica (http://w3.one.net/~hughesj/technica/technica.html)
Jim Hughes's site features both a brief TECHNIC history and the very useful Element Register, a pictorial and annotated list of most TECHNIC parts.

LEGO Set Inventories (http://peeron.com/inv/)
Jennifer & Dan Boger maintain this very useful site where you can search for which sets contain a specific part.

MINDSTORMS Add-Ons (www-control.eng.cam.ac.uk/sc10003/addon.html)
Stuart Crawshaw's attempt to describe all the possible add-on sets for LEGO MINDSTORMS.

LEGO Shop At Home (http://shop.lego.com/)
The official LEGO online shop.

Brickbay (www.brickbay.com/)
The Unofficial LEGO Shopping Mall. The most important independent resource to buy individual LEGO parts not supplied by the official LEGO online shop. Counts almost 300 shops that you can search with a powerful engine.

PITSCO-DACTA (www.pitsco-legodacta.com/)
A source of LEGO DACTA sets and service packs in the USA.

Spectrum Educational Supplies (www.spectrumed.com/)
A source of LEGO DACTA sets and service packs in Canada.

All-LEGO Stepper Motor (http://home.earthlink.net/~mrob/pub/lego/stepper.html)
The original Robert Munafo page for his stepper motor.

HiTechnic (www.hitechnicstuff.com/)
John Barnes' company manufactures a range of sensors, controllers, and mechanical accessories compatible with the LEGO MINDSTORMS products.

Techno-Stuff Robotics (www.techno-stuff.com/)
Pete Sevcik produces and sells a broad range of MINDSTORMS compatible sensors, but on his site he also shares some general construction tips.

Robotics Projects (www.verinet.com/~dlc/projects/botproj.htm)
Dennis Clark describes many projects for custom sensors, including compass interfaces and proximity detection circuits.

LEGO Switch Multiplexer (http://baserv.uci.kun.nl/~smientki/Lego_Knex/Lego_electronica/Switch_multiplexer.htm)
Stef Mientki explains his design for a very sophisticated multiplexer.

JCX Home Page (http://jcx.systronix.com/)
JCX is a substitute for the RCX and represents the most radical step toward the expansion of your system. Based on the JStamp Real-time Native Java Module, the JCX design includes eight input ports, four output ports and much, much more.

Mindsensors (www.geocities.com/mindsensors/)
Nitin and Aparna are two robotics fans that sell some of their custom electronic devices, multiplexers in particular.

Using R/C Servos with the RCX (www.hempeldesigngroup.com/lego/servos/index.html)
Ralph Hempel's page about interfacing R/C Servos to the RCX, with schematics and detailed instructions.

InchLab Page (www.inchlab.com/index_noframes.htm)
In this site by Andreas Peter, you can find many hardware projects, including an interface to R/C Servos, an electromagnetic actuator, a laser brick, and more.

TFM's Home Page (www.akasa.bc.ca/tfm/lego_ms2.html)
This site details Dean Husby's custom sensors and multiplexers. His Motor/Sensor Expander can drive up to six full-featured outputs or connect six input sensors, allowing some mixed configurations.

Chapter 10 Getting Pumped: Pneumatics

C.S. SOH'S LEGO Pneumatics Page (www.geocities.com/cssoh1/)
C. S. Soh's site is subtitled "…where air is power." This is the most important reference for LEGO pneumatics on the Web.

Technic Double-Acting Compressor (www.hempeldesigngroup.com/lego/compressor/index.html)
Home page of Ralph Hempel's famous double-acting compressor. The same site also contains his Pressure Switch (www.hempeldesigngroup .com/lego/pressureswitch/index.html).

Sergei Egorov's LEGO Pneumatics Page (www.malgil.com/esl/lego/pneumatics.html)
Nice page with detailed plans for a double-acting compressor and pneumatic switch.

LEGO Construction Site – Ideas (www.telepresence.strath.ac.uk/jen/lego/ideas.htm)
It's difficult to find a place in this appendix for Jennifer Clark's wonderful site, because it covers so many aspects of robotics. Her page of ideas contains many useful suggestions about pneumatics, but don't miss the other tips, and her models as well!

LEGO Compatible Pneumatic Solenoid Valve (www.jps.net/henrik/danweb/solenoid/solenoid3.html)
Daniel Delcollo developed a custom pneumatic valve that can be controlled directly from the RCX.

Chapter 11 Finding and Grabbing Objects

LEGO Robotics (http://british.nerp.net/lego/robot/)
An interesting study for an anthropomorphic LEGO android. Sketches for hands, arms, shoulders, legs, hips, and so on.

Mark's LEGO Grabber Arm (www.mastincrosbie.com/mark/lego/grabber.html)
This site has instructions on how to build a grabber arm that operates from a fixed base point; similar but more sophisticated than that shown in the MINDSTORMS site.

LEGO FetchBot (http://unite.com.au/~u11235a/lego/fetchbot/)
Ben Williamson explains how his FetchBot works: a robot that can find an object, pick it up, and drop it somewhere else.

Chapter 12 Doing the Math

Numerical Methods (http://tonic.physics.sunysb.edu/docs/num_meth.html)
A Web site that covers all aspects of Numerical Analysis, though finding what you're looking for may require some time.

Numerical Analysis (www.math.niu.edu/~rusin/known-math/index/65-XX.html)
The Mathematical Atlas contains this introduction to the topic and links to many other resources.

Introduction to Time Series Analysis (www.itl.nist.gov/div898/handbook/pmc/section4/pmc4.htm)
An index page from the NIST/SEMATECH Engineering Statistics Internet Handbook about the methods used to analyze time series. It includes moving averages and exponential smoothing.

What's Hysteresis? (www.lassp.cornell.edu/sethna/hysteresis/WhatIsHysteresis.html)
Jim Sethna explains hysteresis in laymen's terms and provides some examples.

Chapter 13 Knowing Where You Are

Where Am I (www-personal.engin.umich.edu/~johannb/position.htm)
The site where you can download the not-to-be-missed "Where am I? — Systems and Methods for Mobile Robot Positioning" by J. Borenstein, H. R. Everett, and L. Feng.

Using PID-Based Technique for Competitive Odometry and Dead Reckoning (www.seattlerobotics.org/encoder/200108/using_a_pid.html)
An excellent article, written by G.W. Lucas, about using the Proportional, Integral, and Derivative (PID) approach in odometry.

Robot Navigation (www.doc.ic.ac.uk/~nd/surprise_97/journal/vol1/oh/)
Oliver Henlich's "Where am I going and how do I get there," an overview of local/personal robot navigation techniques.

JP Brown's Serious LEGO (http://jpbrown.i8.com/)
Here, Jonathan Brown describes the Laser Target we mentioned in Chapter 13. Don't miss his wonderful creations; especially his world-famous Rubik's Cube solver.

Robotics Introduction (www.restena.lu/convict/Jeunes/RoboticsIntro.htm)
Boulette's Robotics Page is one of those sites difficult to classify since it contains useful tips and interesting projects in many different areas. We chose to place it here for its discussion on positioning and for its description of highly specialized sensors used for the task: laser emitters and decoders, compasses, and infrared-ultrasonic beacons.

Chapter 14 Classic Projects

Ben Jackson's MINDSTORMS Creations (www.ben.com/LEGO/rcx/)
Ben's site describes his robots and includes the description of his search for a fast line follower.

Doug Wilcox's LEGO MINDSTORMS Site (www.wordsmithdigital.com/mindstorms/)
In the Projects section of his site, Doug describes the story of his rack-and-pinion steering designs (designed with Carl Jagt).

Huw (www.brickset.com/huwhomepage/)
Huw Millington's home page contains the link to his four-wheel car, a rack and pinion vehicle equipped for both obstacle detection and line following. Don't miss Huw's other creations: the Brick Sorter and the Pneumatic Arm.

Rack and Pinion (http://occs.cs.oberlin.edu/~cmaron/ LEGO/journal7.html)
Chad Maron shows a very compact rack-and-pinion design.

Chapter 15 Building Robots That Walk

Technic Puppy Journal (www.geocities.com/technicpuppy/)
Miguel Agullo's site contains detailed instructions for his Hammerhead ankle-bending walker. Don't miss the Lego Biped Links page, the best organized collection of links to MINDSTORMS bipeds.

Joe's MINDSTORMS Gallery (http://member.nifty.ne.jp/ mindstorms/)
Tons of wonderful robots pack Joe's gallery. Walkers and much more…

S6 Details (www.geocities.com/mario.ferrari/s6/s6.html)
Our first biped COG-shifting robot, S6.

Chapter 16 Unconventional Vehicles

SHRIMP (http://dmtwww.epfl.ch/isr/asl/systems/shrimp.html)
The original SHRIMP by the Autonomous Systems Lab of Lausanne, Switzerland.

SHRIMP Details (www.geocities.com/mario.ferrari/shrimp/ shrimp.html)
Our first version of SHRIMP.

MINDSTORMS Projects Info (www.borg.com/~pinkmice/)
John Barnes' Sewer Rat is able to run through 8" pipes!

Rob Stehlik's Home Page (www.ecf.utoronto.ca/~stehlik/ index.html)
Among Rob's wonderful robotic creatures, you'll find his amazing Window Walker which actually climbs windows!

Duna Rossa Details (www.geocities.com/mario.ferrari/ dunarossa/dunarossa.html)
Duna Rossa, our own robotic sailing tricycle.

The LEGO Train Depot (www.ngltc.org/train_depot/)
This site dedicated to LEGO Trains has a Hints & Tips section that includes MINDSTORMS controlled trains.

Zhengrong Zang's RCX Controlled LEGO Train Projects (http://legochina.virtualave.net/)
Zhengrong dedicated his entire site to solutions and schemes about controlling trains with the RCX.

Pacific NW LEGO Train Club (www.pnltc.org/)
PNLTC Web site has articles about using LEGO MINDSTORMS to control trains.

TFM's Home Page (www.akasa.bc.ca/tfm/index.html)
This site covers controlling LEGO locomotives and crossings with the RCX.

Where No Man Has Gone Before (www.hq.nasa.gov/office/pao/History/SP-4214/ch13-3.html)
A description (no images) of the Lunar Rover used in the Apollo missions.

Exploring The Planets – Rovers (www.nasm.edu/ceps/etp/tools/tools_rover.html)
A gallery of rovers used in planet exploration.

Chapter 17 Robotic Animals

MINDSTORMS Info Center (www.mi-ra-i.com/JinSato/MindStorms/index-e.html)
The English index of Jin Sato's site, home of the famous MIBO robotic dog and many other amazing creations.

Creating a Spider Robot Using LEGO MINDSTORMS (http://schalburg.homepage.dk/Spider/Spider.html)
René Schalburg describes, with many interesting details, the process of building a robotic spider.

S17 Details (www.geocities.com/mario.ferrari/s14/s14.html)
Our leash-driven Cyberdog.

Chapter 18 Replicating Renowned Droids

R2-D2 Builders Club (www.robotbuilders.net/r2/)
The R2 Builders Club is a forum about building a personal version of

the renowned droid. It's not about LEGO, but you can find applicable tips and inspiring images.

Johnny-Five.com (www.johnny-five.com)
A fan-made Web site devoted to Johnny Five.

Clint Rutkas' Skunk Works (http://members.nbci.com/_XMCM/rutkas/index.html)
Clint Rutkas's site contains, among many other interesting things, two large MINDSTORMS robots emulating R2-D2 and Johnny Five.

Otto Details (www.geocities.com/mario.ferrari/otto/otto.html)
Our large octagonal R2-D2 clone.

Cinque Details (www.geocities.com/mario.ferrari/cinque/cinque.html)
Our large scale replica of Johnny Five.

Chapter 19 Solving a Maze

MINDSTORMS MazeWalker (www.hempeldesigngroup.com/lego/mazewalker/index.html)
Another link to Ralph Hempel's site—this time to point to his maze solver, an application demonstrated during the 1999 Mindfest at MIT.

Maze Solving Algorithm (www.lboro.ac.uk/departments/el/robotics/Maze_Solver.html)
A description of the Bellman flooding algorithm.

Micromouse: Maze Solving (www.cannock.ac.uk/~peteh/micromouse/maze_solving.htm)
This site is dedicated to Micromouse Maze solving competitions. The page we mention is specifically about Maze Solving algorithms.

Chapter 20 Board Games

TTT: A LEGO MINDSTORMS Tic-Tac-Toe Player (www.geocities.com/mario.ferrari/ttt.html)
The page of the TTT robots we showed at the Mindfest. You can also find Antonio Ianiero's compact YATTT NQC source code there.

Andy's LEGO MINDSTORMS Ideas (www.artilect.co.uk/lego/)
Andy created a robot that plays Four-in-a-row and started a Chess project based on Francesco Ferrara's QC programming system.

Chapter 21 Playing Musical Instruments

S15 & S16 Details (www.geocities.com/mario.ferrari/s15/s15.html)
Our piano player and conductor team.

Chapter 22 Electronic Games

Rolighed's LEGO MINDSTORMS Site (http://home14.inet.tele.dk/rolighed/)
Soren Rolighed made a working LEGO slot machine! Don't miss his MINDSTORMS Typewriter, too.

Chapter 23 Drawing and Writing

LOGO Turtle (www.ecf.utoronto.ca/~stehlik/turtpics.html)
Rob Stehlik shows you step by step how to replicate his Logo Turtle.

Logo Foundation (http://el.www.media.mit.edu/groups/logo-foundation/index.html)
The Logo Foundation Web site: A place to find information and resources useful in learning and teaching Logo.

ECG Sensor (http://baserv.uci.kun.nl/~smientki/Lego_Knex/Lego_electronica/BioSensors/ECG_sensor.htm)
This custom MINDSTORMS ECG sensor can turn your Tape Writer into an ECG machine!

Iñaki, Xabin eta Koldoren orria (www.euskalnet.net/kolaskoaga/)
A sophisticated plotter, able to change pens, too (this site is in Spanish).

Haiku Program (http://severed.tentacle.net/rpeake/archives/programming/haiku.html)
C source for an automatic Haiku writer.

Web-Ku: Haiku for the WWW (www.obs-us.com/people/sunny/haiku/web_ku.htm)
A list of links to random haiku generators.

Chapter 24 Simulating Flight

LEGO MINDSTORMS Inventions (http://mindstorms.lego.com/inventions/default.asp)
The official MINDSTORMS site includes some flight simulators. Even if all of them simulate just the attitude of the plane and not its effects, Wouter Kooijman's site provided us with interesting starting points.

FlightGear Flight Simulator (www.menet.umn.edu/~curt/fgfs/)
For those who want to explore the details of a much more complete simulation, FlightGear is a free, open source, multi-platform cooperative flight simulator.

Chapter 25 Building Useful Stuff

Mike's LEGO MINDSTORMS Page in Wien (http://insel.heim.at/mainau/330001/lego.htm)
Michael Brandl shows his gallery of robots, which include "Adam der Gärtner" the robot that inspired our own "Plant Sprinkler." Don't miss his other imaginative and technically clever robots: a free climber, a robotic fish, and much more.

Chapter 26 Racing Against Time

LUGMAP (www.lugnet.com/map/)
An interactive map of LEGO user groups from around the world.

FIRST LEGO League (www.firstlegoleague.org/)
The official site of the FIRST LEGO League, a partnership between the LEGO Group and an organization called For Inspiration and Recognition of Science and Technology (FIRST). FLL organizes LEGO Robotics Competitions for 9 to 14 year old children.

rtlToronto (http://peach.mie.utoronto.ca/events/lego/)
A very active LEGO users group with a strong focus on MINDSTORMS competitions. The site contains pages about the various events and their rules.

BrickBots (www.brickbots.com/)
Richard Sutherland's repository of building contests and best solutions. A nice site where you can attend "remote" challenges.

Robot Arena (www.azimuthmedia.com/RobotArena/mainframe.html)
A site dedicated to autonomous robotic combat using LEGO MINDSTORMS.

ItLUG (www.itlug.org)
The Italian LEGO Users Group is very active in organising robotic contests. Follow the link to the Events page.

Chapter 27 Hand-to-Hand Combat

Atlanta Hobby Robot Club Web Site (www.botlanta.org/index.html)
AHRC runs robotic sumo contests (including but not restricted to LEGO robots). Their site includes a page with detailed rules.

LEGO MINDSTORMS (www.geocities.com/mario.ferrari/lego_mindstorm.html)
Our LEGO robotics pages contain many of our Sumo and Mini-Sumo contenders.

Chapter 28 Searching for Precision

Maxwell's Demons – Official rules (http://news.lugnet.com/org/us/smart/?n=22)
David Schilling's original post about the rules concerning his Maxwell's Demons competition.

RoboCup Junior 1999 (www.daimi.au.dk/~hhl/RoboCupJr/RoboCupJr_report.html)
Henrik Hautop Lund's page about the first RoboCup Junior event—featuring MINDSTORMS Soccer.

Mindfest (www.daimi.au.dk/~hhl/MindFest/)
Coverage of the LEGO MINDSTORMS Robot Soccer for Children at Mindfest.

LEGO Robots: Challenge (www.cs.uu.nl/~markov/lego/challenge/index.html)
The account of a soda can retrieval challenge at the Department of Computer Science at Utrecht University (in the Netherlands).

Appendix B

Matching Distances

Legend:

- Each cell of the table contains three data: the distance in LEGO units (studs), the quality of the matching, and the resulting angle in degrees.
- Distances are measured *excluding* the starting point. (For example, if one peg is in the first hole of a beam and another is in the tenth, the distance is nine units.)
- The quality of the matching is expressed with a symbol that reflects the difference between the actual distance and the closest perfect match, expressed in LEGO units, according to the following scheme:

Symbol	Meaning	Maximum Tolerance
P	Perfect match	0.00 studs
V	Very good	0.02 studs
G	Good	0.04 studs
N	Not so good	0.06 studs
B	Bad	0.08 studs

Height in Bricks and Plates	**Base in Studs**																				
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1										9 B 8°	10 B 7°	11 B 6°	12 N 6°	13 N 5°	14 N 5°	15 N 5°	16 N 4°	17 N 4°	18 G 4°	19 G 4°	20 G 3°
1 1/3																	16 B 6°	17 B 5°	18 B 5°	19 B 5°	20 B 5°
1 2/3	2 P 90°																				
2																					
2 1/3		3 G 70°																			
2 2/3						6 B 33°															
3							7 V 31°														
3 1/3	4 P 90°			5 P 53°				8 B 30°	9 N 27°												
3 2/3					6 N 48°					10 V 26°	11 B 24°										
4						7 B 44°						12 V 24°	13 B 22°								
4 1/3				6 V 60°			8 B 41°						13 B 23°	14 V 22°	15 B 20°						
4 2/3			6 N 70°					9 G 39°							15 B 22°	16 V 20°	17 N 19°				
5	6 P 90°								10 P 37°									18 G 19°	19 G 18°	20 B 18°	
5 1/3				7 B 65°						11 N 35°										20 N 19°	21 V 18°
5 2/3							9 B 49°					13 B 32°									
6								10 N 46°					14 V 31°								
6 1/3									11 G 44°					15 N 30°	16 B 28°						
6 2/3	8 P 90°	8 B 83°			9 N 63°		10 P 53°			12 N 42°						17 P 28°					
7								11 B 50°			13 N 40°						18 B 28°	19 G 26°			
7 1/3			9 G 77°																20 G 26°	21 B 25°	

Continued

Height in Bricks and Plates	Base in Studs																				
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
7 2/3					10 G 67°		11 V 57°							16 B 35°							22 V 25°
8				10 N 73°											17 G 34°						
8 1/3	10 P 90°	10 N 84°														18 G 34°					
8 2/3							12 V 60°											20 B 31°			
9			11 V 80°							14 N 50°									21 V 31°		
9 1/3											15 V 48°				18 B 39°					22 N 31°	23 B 29°
9 2/3				12 V 75°			13 N 63°					16 V 47°				19 G 38°					
10	12 P 90°	12 N 85°				13 P 67°				15 P 53°			17 G 45°				20 P 37°				
10 1/3					13 G 72°						16 B 51°			18 G 44°				21 N 36°			
10 2/3			13 N 81°												19 G 42°						
11								15 N 62°		16 G 56°						20 V 41°					24 G 33°
11 1/3				14 B 78°													21 V 40°				
11 2/3	14 P 90°	14 G 86°																22 G 39°			
12					15 N 74°			16 V 64°		17 V 58°									23 N 39°		
12 1/3			15 B 82°				16 G 68°						19 N 51°			21 B 45°					
12 2/3						16 V 72°								20 V 49°			22 B 44°				
13										18 V 60°					21 G 48°			23 B 43°			
13 1/3	16 P 90°	16 G 86°											20 P 53°			22 B 47°					
13 2/3														21 B 52°							
14				17 B 80°						19 N 62°											

Continued

Height in Bricks and Plates	Base in Studs																				
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
14 1/3									19 G 65°				21 G 55°								
14 2/3					18 N 77°			19 N 68°													
15	18 P 90°	18 G 87°					19 G 72°														
15 1/3						19 B 75°			20 B 67°		21 N 61°		22 G 57°					25 N 47°			
15 2/3				19 G 81°				20 B 70°								24 N 51°			26 G 46°		
16																	25 V 50°			27 V 45°	
16 1/3					20 V 78°						22 V 63°		23 V 59°					26 N 49°			28 V 44°
16 2/3	20 P 90°	20 G 87°								22 B 66°						25 P 53°					
17						21 V 76°											26 B 52°				
17 1/3				21 V 82°				22 N 71°			23 B 64°		24 V 60°		25 B 56°						
17 2/3							22 G 74°			23 G 67°						26 G 55°					
18					22 G 80°				23 G 70°												
18 1/3	22 P 90°	22 G 87°											25 N 61°		26 B 58°					29 B 49°	
18 2/3						23 N 77°						25 N 64°				27 N 56°					30 G 48°
19				23 V 83°															29 N 52°		
19 1/3							24 G 75°													30 V 51°	
19 2/3					24 B 80°							26 G 65°		27 N 61°		28 G 58°					31 B 50°
20	24 P 90°	24 G 88°						25 P 74°			26 P 67°								30 P 53°		

Appendix C

Note Frequencies

The following table contains note frequencies rounded to the nearest whole number.

Octave	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
1	33	35	37	39	41	44	46	49	52	55	58	62
2	65	69	73	78	82	87	92	98	104	110	117	123
3	131	139	147	156	165	175	185	196	208	220	233	247
4	262	277	294	311	330	349	370	392	415	440	466	494
5	523	554	587	622	659	698	740	784	831	880	932	988
6	1047	1109	1175	1245	1319	1397	1480	1568	1661	1760	1865	1976
7	2093	2218	2349	2489	2637	2794	2960	3136	3322	3520	3729	3951
8	4186	4435	4699	4978	5274	5588	5920	6272	6645	7040	7459	7902

Appendix D

Math Cheat Sheet

Sensors

Raw values to percentage (light sensor):

percentage = 146 – raw value / 7

Raw values to temperatures, in C° (temperature sensor):

C° = (785 – raw value) / 8

Conversion of Celsius to Fahrenheit degrees:

F° = C° x 9 / 5 + 32

Averages

Simple average:

$A = (V_1 + V_2 + \ldots + V_n) / n$

Weighted average:

$A = (V_1 \times W_1 + V_2 \times W_2 + \ldots + V_n \times W_n) \; / \; (W_1 + W_2 + \ldots + W_n)$

Exponential smoothing:

$A_n = (V_n \times W_1 + A_{n-1} \times W_2) \; / \; (W_1 + W_2)$

Interpolation

Linear interpolation: Find the value of the dependent variable Y for a given value of the independent variable X, knowing that for X equal to X_a, Y is Y_a, and for X equal to X_b, Y is Y_b.

$(Y - Y_a) / (Y_b - Y_a) = (X - X_a) / (X_b - X_a)$
$Y = (X - X_a) \times (Y_b - Y_a) / (X_b - X_a) + Y_a$

Equation of the straight line which connects the points (X_a, Y_a) and (X_b, Y_b):

$m = (Y_b - Y_a) / (X_b - X_a)$
$b = Y_a - m \times X_a$
$Y = m \times X + b$

Gears, Wheels, and Navigation

Output angular velocity of the body of a differential gear Oav, given the input angular velocity of the two axles Iav_1 and Iav_2:

$Oav = (Iav_1 + Iav_2) / 2$

Distance, Time, Speed:

distance = speed x time

speed = distance / time

Circumference C of a wheel, given the diameter D:

$C = D \times \pi$

$\pi = 3.1415926\ldots$

Increment in rotation sensor count I that corresponds to a turn of the wheel, given R the resolution of the sensor and G the gear ratio between the wheel and the sensor:

$I = G \times R$

$R = 16$ (for Lego rotation sensors)

Conversion factor F which measures the traveled distance of a wheel for any single increment in the count of a rotation sensor:

$F = C / I = (D \times \pi) / (G \times R)$

Actual traveled distance, given F and the count of the sensor:

$T = \text{Count} \times F$

Traveled distance T_C of a differential drive robot's centerpoint, given the traveled distances T_L and T_R of its left and right drive wheels:

$T_C = (T_R + T_L) / 2$

Change of orientation ΔO_R, in radians, of a differential drive robot, given the traveled distances T_L and T_R of its left and right drive wheels, and the distance B between the wheels:

$\Delta O_R = (T_R - T_L) / B$

New orientation O_i of a robot after a change in orientation ΔO from the previous orientation O_{i-1}:

$$O_i = O_{i-1} + \Delta O$$

New position of a robot (x_i, y_i) of a robot after having covered a distance T_C in direction O_i from position (x_{i-1}, y_{i-1}):

$$x_i = x_{i-1} + T_C \text{ x } \cos O_i$$
$$y_i = y_{i-1} + T_C \text{ x } \sin O_i$$

Conversion of radians to degrees:

$$\text{Degrees} = \text{Radians x } 180 / \pi$$

Required increment in rotation sensor count for a given change of orientation ΔO_R in radians or ΔO_D in degrees:

$$\text{Count} = T / F = (\Delta O_R \text{ x } B / 2) / F = \Delta O_R \text{ x } B / 2F$$
$$\text{Count} = \Delta O_D \text{ x } \pi \text{ x } B / (360 \text{ x } F)$$

Index

% symbol. *See* Not Quite C
3-4-5 triad, usage, 12, 15

A

Absolute errors, 216
Absolute positioning, 234, 236–243, 374
AC. *See* Alternating current
Acceleration, 482. *See also* Negative acceleration
  favoring, 517
  influence, 518
  output, 472
Accessory sets, 161
Accuracy, evaluation, 519
Ackerman
  correction, 272
  steering, 142
    usage. *See* Smooth turns
  Web site, 557
Acrobot, 90
Action, bounded range, 25
Active sensors, 74, 170, 211
Actual traveled distance, 579
ADA, Web site, 554
Adam der Gärtner robot, 503
Adder-subtractor, 551
Agullo, Miguel, 280, 305, 308, 562
AHRC. *See* Atlanta Hobby Robot Club
AI. *See* Artificial Intelligence
Ailerons, 468
Air compressors, construction, 190–194
Air power, 530
Air pressure, 183
Air supply, 188, 193
Air tanks. *See* TECHNIC air tanks
Airflow
  control, 187–190
  detachment, 484
Air-powered models, 181
Alarm clock, 508
Algorithms. *See* Bellman Algorithm; Navigation
  usage. *See* Software
All-LEGO board, 397
All-LEGO Stepper Motor, Web site, 558
Alphanumeric capabilities, 463
Alternating current (AC), 42
Altitude, 471, 481
  display, 472
  losing, 488
Ambient IR radiation, 167
Ambient light, 164, 394
  change, 70
  interference, minimization, 298
  reading, 65
Ambient radiation, 168
Ambient temperature, 229
Ampere, 42–43
Amplitude, 230
AND configuration, 79
Android, study, 559
Andy's LEGO MINDSTORMS Ideas, Web site, 565
Angle connectors, usage, 340, 341
Angles
  radians, 450
  tracking systems, 443
Angular velocity, 19, 29. *See also* Input; Output ports; Rotating wheel
Animals. *See* Four-legged animals; Robotic animals
  creation, 345–346
Ankle-bending model, 308
Ankles
  bending, 306–309
  movements, 308
Antidote robot, design, 536
Aparna, 558
Applied power, 482
Arachnids, 286
Armadillos, 345
Arrays
  elements, 222
  usage, 220
Articulated drive, 150
Artificial Intelligence (AI), 392, 396

frontiers, 410
Web site, 550
Artificial vision, 234
Asymmetrical caster, 335
Asymmetrical system, 23
Atlanta Hobby Robot Club (AHRC), Web site, 567
Atmospheric pressure, 503
Attack, 534–535
strategies. *See* Robotic sumo
Automatic Haiku Writer, 462
Automation devices, 229
Autonomous Systems Lab, 312, 562
Average value. *See* Dark/bright
Averages, 214, 578. *See also* Simple averages; Weighted averages
Averaging, 218. *See also* Data
Axis. *See* Lateral axis; Longitudinal axis; Pivoting axis; Vertical axis
Axle-pegs, 85
Axles, 18, 155, 579. *See also* Driving axles; Front wheels; Long joined axle; Output ports; Perpendicular axle; Pivoting axle; Steer axle; Traverse axle; Wheels
angle, increase, 271
assembly, 479
connecting. *See* Distant axles
connection, 136
force, 88
holding, 32
hole, 297, 397, 479
motion transfer, 19
studs, inclusion, 157
support. *See* Load-bearing axles
turning, 19, 23
turns, 71
usage, 6–7, 36

B

Baby entertainer, 508–509
back (command), 449, 454
back (subroutine), 450
Backlash, 29, 264
Backward motion, 175
Badham, John, 350
Baker, Steve, 73, 551, 552
Balloon tires, 139, 161
Balls, 428
Bank, 468, 471
angle, 481
change, 475
control, 473–476
coordination, 472
effect, 485
measurement, 476
movement, 475
values, 472
variable, computation, 482
Bar code, 236
Barnes, John, 166–168, 558, 562
Base beam, 415
Baseplate, 383, 534
Basic Input/Output System (BIOS), 99
Bass drum, 413
Batteries, 88–89, 528. *See also* NiMH batteries; Rechargeable batteries
AA batteries, 517
box, 51, 193, 518. *See also* External battery box
combination, 517
draining, 45, 505
making. *See* Fake batteries
supply, 506
types, 516
Battery-powered pen laser pointers, 241
Baum, David, 102, 548, 552–553
Beacons, 241. *See also* Infrared-ultrasonic beacons
following. *See* Radio beacons
Beams, 6, 160–161. *See also* Flashlight beam; Half-beam; Horizontal beams; Infrared; Layered beams; Longitudinal beam; One-hole beam; Parallel beams; Perpendicular beam; Stacked beams; Vertical beams
connecting, 10
deforming, 22
dimension, 312

hole, 570
importance. *See* Locking beams
mounting. *See* Inner beams
placement, 155
plates, combination, 9
support. *See* Inner beams; Outer beams
usage. *See* Stack construction; Standard beams; Traverse beam
vertical position, 415
BEAT value, 121
Behavior, 515. *See also* Complex behavior; Elastic behavior; Plane; Predefined behavior; Random behavior; Robot; Violent behavior
Bellman Algorithm, 388
Belts, 49
coupling, 51
elasticity, 33
pulleys, friction, 33
slippage, 49, 202
usage, 31–35
Ben Jackson's MINDSTORMS Creations, Web site, 561
Bend connector, 340
Berti, Marco, 237, 392, 517, 544
Beri, Marco, 555
Bevel gears, 29, 148, 342. *See also* Central bevel gear; Double bevel gear
12t, 30
pair, 402
rotation, 38
Bibliography, 548–549
Bi-directional outputs, 176
BIOS. *See* Basic Input/Output System
Bipeds, 286, 299–309. *See also* Center of gravity
robots, 284
turning ability, 309
Bi-stable mechanism, 203
Bi-stable system, 203
Black pegs, usage, 85, 176, 203
Black pins, usage, 85
Bliss, Steve, 549
Blocks, 12. *See also* Code; Multicolor movable block
Board games, 391
introduction, 392
playing, 409–410. *See also* Chess; Tic-Tac-Toe
summary, 410
Boats, 329–330
Body structure, 343
Boger, Jennifer/Dan, 557
Bogie, 312. *See also* Left bogie; Right bogie
control, 320
Borderline, 68–69, 262
Borenstein, J., 560
Bot-Kit, Web site, 553
Boulette's Robotics, Web site, 556, 561
Bower, Andy, 550
Bracing. *See* Diagonal bracing; Vertical bracing
hinges, usage, 14–15
Brake effect, 54
Braking. *See* Effective braking
limit, 518
power, 518
Brandl, Michael, 503, 566
Brick Command, 105
Web site, 554
Brick Programmer. *See* Gordon's Brick Programmer
Brick Sorter, 561
Brickbay, Web site, 557
BrickBots, Web site, 567
Bricks, 4, 60, 101, 368. *See also* Light brick; Programmable bricks; TECHNIC brick; Transparent bricks
colors, 66
damage, 12
delicacy, 60
dimensions, 46, 131, 203, 256, 428. *See also* TECHNIC brick
side. *See* Weighed brick side
orientation, 458
usage, 362, 432
Brickshelf, Web site, 550
Bricx Command, Web site, 554
BricxCC, 554
Brightness, 238

Broad Blue, 401
  arms, 408
  connecting/programming, 406–408
  robotic arm, 397
  stability, 405
  wirings/resource allocations, 406
Brok, Eric, 10, 551
Brown, Johnathan, 242, 561
Bühn, Oliver, 555
Bumper cars, 145
  monitoring, 269
Bumpers, 59–60, 210, 252. *See also* Front bumpers; Left side bumper; Rear bumpers
  addition, 359
  closing, 79
  design. *See* Contact bumpers; Smooth bumpers
  operation, 269
  placement, 329
  testing, 110
  wheels, 381
Bushings. *See* Half bushings
Bytecode, 99
  format, 100
  interpretation, firmware (usage), 100
  transferring, 100

C

C code, 554
Cable railways, 329
Cadmium sulfide (CdS) photo-resistor, 164
Calibrate (routine), 111
Calibration
  process, 109
  routine, 108
Calibration (routine), 119
Cars. *See* Front-wheel drive; Remote-controlled car; Robotic car; Supercar
  modeling, 265–277
  racing, 522
  steering assembly, 354
Cartesian coordinates, 245, 464
Cartesian path, 228
Cartesian plane, 383
Cartesian scheme, 397
Cartesian system, 383, 401, 455
Casters, 380. *See also* Asymmetrical caster
  usage. *See* Straight path
C/C++ code, 104
CdS. *See* Cadmium sulfide
Cells. *See* Closed cells; Open cells
Cellular phones, 123
Celsius degrees, 74–75
  conversion, 578
Center of gravity (COG), 280, 443, 474. *See also* Real COG; Static COG; Virtual COG
  COG-shifting biped, 302
  movement, 281, 284, 286, 302–303
  placement, 360, 544
  position, 283, 289, 308, 322
  shifting, 299, 302–306. *See also* Walker
Center_steer (subroutine), 269, 270
Central bevel gear, 38
Centrifugal force, 471
Chain links, 35, 157
Chains, usage, 31–32, 35–36
Challenges, 514, 516
Changeover catch, 132. *See also* Transmission
  pushing, 274
Chassis. *See* Jointed chassis; Swinging chassis; Wheeled chassis
  construction, 90
Chassis/modularity/load, combination, 90–94
Check_Bumper (subroutine), 108, 110, 112
Checkers, 392, 409
check_sequence (subroutine), 434, 435, 437
Chess, 392
  construction, variations, 408–409
  interface, 397
  mechanical interface, construction, 400–408
  playing, 396–409
  project, 565
  software, interfacing, 396
  visual interface, construction, 397–400

Cinque, 205
Cinque Details, Web site, 564
Circuits. *See* Closed circuit; Short circuit
  bypassing. *See* Current-limiting circuit
  cycling. *See* Control circuit
Circular pointer, 221, 222
Circumference, 33, 216, 579. *See also* Wheels
Clark, Dennis, 558
Clark, Jennifer, 559
Clicks, counting, 62–64
C-like syntax, 102
Clint Rutkas' Skunk Works, Web site, 564
Closed cells, 387
Closed circuit, 53
Clutch. *See* Gears
Clutch gear, 187, 193
  incorporation, 26
  usage, 43. *See also* Strength
Code. *See* Bytecode; Not Quite C; Programming; Read-only memory; Robotic Command eXplorer
  blocks, 101
    sequence, 102
  count, 453
  counting, 63
  design, benefits. *See* Modular code design
  execution, 521
  instructions. *See* Machine code instructions
  organization, 106–113
  routines, 422
  structure, 112
  usage. *See* LEGO RCX code
  writing, 380, 518
COG. *See* Center of gravity
Collection abilities, 540–543
  variations, 543
Collisions, 59
  number, monitoring, 253
Colored markers, 394
Colors, 160. *See also* Bricks; Reading colors; Sectors; Walls
  choice. *See* Parts
  differentiation. *See* Light sensors
  recognition, 65. *See also* Cubes
  stripes, 238
Combat. *See* Hand-to-hand combat
Command-line tool, 103
Communication protocol, 407, 502
Compact bumpers, design, 532
Compasses, 154
  sensors, 167
Competing robots, 520
Competitive odometry, Web site. *See* PID-Based Technique for Competitive Odometry and Dead Reckoning
Compiler. *See* Not Quite C
  management. *See* Cross-compilers
Complex behavior, 350
Components, 210, 366. *See also* Non-original components; Specialized components
  creation. *See* Custom components
Compressed air, source, 191
Compressors, 192–194, 424. *See also* Double acting compressors; Double-acting compressor
  construction. *See* Air compressors
  usage, 505
Configuration. *See* AND configuration; OR configuration
Connection. *See* Diagonal connections; Touch sensors
  double-checking, 55
Connectors, 155, 161, 339. *See also* Bend connector; Pin-axle connectors; TECHNIC connectors
  usage. *See* Angle connectors
Console mode, 66
Constants, 120
  table, 121
Constant-velocity (CV) joint, 271
Construction
  equipment, 150
  kits, 165
Construction strategies, 83
  introduction, 84
  summary, 94–95
Constructopedia, 134, 363

Contact sensors, usage, 532
Contests, 511
 attending, 515
 hosting, 514–516
 introduction, 514
 participation, 514–516
 rules
  defining, 514–515
  studying, 515
 strategy, 515
 summary, 523
Control Center unit, 51
Control circuit, cycling, 74
Control Lab Learning Environment. *See* DACTA
Control panel, 437
Control system
 need, 409
Conversion factor, 244
Coordinate system, 397
Co-ordination, 342
Copying, 464
Corrections, number, 256
Corrective action, 521
Counter-rotating wheels, 533, 534
Coupling. *See* Belts; Mechanical couplings; Motors
Crabs, 334, 346
Crankshaft, 305
Crawshaw, Stuart, 557
Creating a Spider Robot. *See* Spiders
Cross-compilers, management, 104
Crossed hole, 267, 323
Crown gear, 30
C.S. Soh's LEGO Pneumatics, Web site, 559
Cubes, 5, 206, 210
 color, recognition, 540
 competition. *See* Stealing the Cube competition
 discovery, 540
 stealing, 541–542
Cul-de-sac, 253
Current, 42
 consumption, 45
Current-limiting circuit, bypassing, 51
Current-limiting device, 51, 55, 528. *See also* Inner current-limiting device
Custom components, creation, 163–172
Custom sensors, construction, 164–170
CV. *See* Constant-velocity
Cyberdog, 563
Cybermaster, 100, 103, 366
 limitations, 101
 set, 80
Cycles. *See* Low duty cycle
Cylinders, 181–187. *See also* Pneumatic cylinder
Cymbal, 413

D

DACTA
 boxes, 162
 catalog, 163
 Control Lab Learning Environment, 454
 eLAB, Web site, 552
 service, 162
 Web site. *See* PITSCO-DACTA
Dark gray 16t gear, 444
Dark Side Developer Kit, 161
Dark/bright, average value, 256
Data
 averaging, 218–226
 logging, 103
Datalog, 119
DC. *See* Direct current
Dead reckoning, 235, 242, 377, 449
 concepts, 255
 rotation sensors, usage, 498
 techniques, 542
 Web site. *See* PID-Based Technique for Competitive Odometry and Dead Reckoning
Decorative parts, 302
Decorative tail, 300
Decouplers, 329
Deduced reckoning, 235
Deep Blue, 401
Defense subsystems, 535

Defense system. *See* Robotic sumo
#define statements, grouping, 436
Degree of freedom (DOF), 455, 464, 474
  joint, 478
  mechanism, 208
  understanding, 200, 205–209
Delay values, 422
Delcollo, Daniel, 559
Delimited space, 209
Demons. *See* Maxwell's Demons
Dependent variable, 227, 578
Destination
  inputting, 399
  square, 399, 407, 409
Detection
  field, 166
  techniques, 522
Detectors. *See* Hall-effect detector
Diagnostics, 103
Diagonal beams, usage, 10, 12
Diagonal bracing, 10–12
Diagonal connections, 10
Diagonal spreadsheet (length), Web site, 551
Diagonals. *See* Working diagonal
Differential drive, 111, 379, 443, 473–474
  architecture, 128, 250
  construction, 128–135. *See also* Dual differential drive
  edge detection, 265
  favoring, 519
  line-following program, 262
  motors, 480
  observation, 258
  platform, 489
  power, 140
  robot, 128, 579
  straight path, maintenance, 130–135
  usage, 174, 256, 539. *See also* Dual differential drive; LOGO turtle
  variation, 138
Differential gear, 36, 189
  body, 579
    observation, 38
  connection, 531
  lock motor, 133
  motor, driving, 335
  usage, 172–174, 454
Differential light sensor, 164
Differentials, 136, 274. *See also* Limited slip differential
  driving, 260
  rotation, 137
  usage, 36–39, 189
Dinosaur-like appearance, 302
Dinosaurs, 345
Direct current (DC), 42
  motors, 49
Direction, 481. *See also* Perpendicular direction; Reverse direction; Rotation direction; Side-to-side direction
  change, 258, 309. *See also* Robot
  combination. *See* Speed
  reversing. *See* Motors
Disconnection. *See* Robotic Control eXplorer
Discovery abilities, 540–543
Discovery set. *See* Robotics
Distances, 10, 20, 579. *See also* Actual traveled distance; Relative distance; Traveled distance; Working distances
  alteration, 452
  covering, 37, 374, 580
  highlighting, 47
  increasing, 9
  knowledge, 521
  matching, 569–574
  measuring, 6
  ratio, 238
  spacing, 13
  tracking systems, 443
Distant axles, 49
  connecting, 35
Dives, 488
Divide et impera (motto), 106
Dividend, 217
Division, 215–218, 228
Divisor, 217
DOF. *See* Degree of freedom
Dog. *See* MIBO robotic dog

Dog trainer, 509
Dolphin Smalltalk, 553
Done (return value), 407
Dorst, Leo, 201
Double acting compressors, 192
Double bevel gear, 30
Double-acting compressor, 405, 504, 559
Doug Wilcox's LEGO MINDSTORMS, Web site, 561
Doug's LEGO Robotics, Web site, 556
DOWN constant, 448
Drag, 470, 482
  output, 472
Drag racing, 516–518
Draw bridges, 329
Drawing, 441. *See also* Mazes
  introduction, 442
  suggestions, 464–465
  summary, 465–466
Drives. *See* Articulated drive; Four-wheel drives; Front-wheel drive; Pivot drive; Single-motor steering drive; Tri-star wheel drive
  configurations, 150–151
  construction. *See* Differential drive; Dual differential drive; Skid-steer drive; Steering; Synchro drive; Tricycle drive
  gearing, 259
  shafts, 271
  wheels, powering, 140
Driving axles, 90
  blocking, 146
Driving light source, 356
Driving ring. *See* Transmission
Droid Developer Kit, 161
Droids, construction. *See* Johnny five-style droid; R2-D2-style droid
Droids, replication, 349
  introduction, 350
  summary, 368–369
Drummer
  construction, 413–416
  creation, 412–417
  programming, 416–417
  variations, 417
Dual differential drive
  construction, 136–138
  usage, 174, 245
Dual IRPD, 166
Dual RCX robot, 161, 367
Dual-RCX projects, 406
Duna Rossa, 330
  Details, Web site, 562
Duty cycle. *See* Low duty cycle
Dynamic balance, 286
  usage, 287
Dynamic forward motion, 143
Dynamic stability, 287
Dynamics, 286

E

Eaton, Doug, 555
ECG Sensor, Web site, 565
Echo, 166
Edges
  detection, 253–255. *See also* Differential drive
  sensors, 254. *See also* Left-edge sensors; Right-edge sensors
Editors, 103. *See also* Text editor
Effective braking, 269
Efficiency, 22, 256
Egorov, Sergei, 551, 559
Eight-cylinder motor, 272
eLAB. *See* DACTA
Elastic behavior, 180
Elasticity, 181, 200
Electric brake, 53, 132
Electric coils, usage, 409
Electric devices, 77
Electric motor, 42, 200
  power, control, 52
Electric Piston, 201
Electric plates, 169
Electric system, 50
Electric valve, 187–188
Electrical energy, 42
Electrical signals, RCX conversion, 66
Electricity

conduction, 330
flow, 58
sources. *See* Non-LEGO electricity sources
Electrocardiograms, 463
Electroencephalograms, 463
Electromagnetic coil, 409
Electromagnets, 438
Electronic components, 330
Electronic games, 425
introduction, 426
suggestions, 438–439
summary, 439
Electronic interface, 172
Element packs, 162
Element Register, 557
Elevators, 328
Elevators (airflight), 468
trim, 470
Ellipse, description, 288
Emitting ball. *See* Infrared
Emulators. *See* Terminal emulator; Unix emulators
EmulegOS, Web site, 555
Encapsulation. *See* Not Quite C
Encoder, 243
resolution, 244
Enemy, discovery, 530–532
Energy. *See* Electrical energy; Mechanical energy; Potential energy; Thermal energy
conversion, 42
density, 181
storage, 186
Engineering
Statistics Internet Handbook, 560
Web site, 552
Engines, construction. *See* Pneumatic engine
Environment, analysis, 372
Eolo, 530
Epistemology and Learning Group, 442
Equilateral triangle, 452
Eriksson, Mikael, 554
Errors. *See* Absolute errors; Percentage errors; Relative errors
Erwin, Benjamin, 549
Esterel, 555
Event-driven programs, 114
Event-driven scheme, 114
Everett, H.R., 560
Expansion sets, 161
Exploration Mars Set, 161
Exploring the Planets, Web site, 563
Exponent, 217
Exponential smoothing, 223–226, 578
Expression, last step, 217
External battery box, 50
External guidance, choice, 234–235
External light sources, 77
External power source, supply. *See* Robotic Command eXplorer
External reference points, usage, 374
External sources, effect, 70
Extra-large robot, 400
Extreme Creatures Set, 161

F

Face-down light sensor, 107, 518
Fahrenheit degrees, 74–75
conversion, 578
Fake batteries, making, 506
Falco, Mark, 555
Fast painting, 522
Feeding mechanism, 458
Feet. *See* U-shaped feet
size, 303
Felt-tip pen, 522
Feng, L., 560
Ferrara, Francesco, 554, 565
Ferrari, Mario, 555
FetchBot, Web site. *See* LEGO
Fiber Optic System unit, 161
Fiber, position, 429
Fiber-Optic System (FOS)
control, 398
device, 78, 396, 398
increments, 399

lights, 399, 431
unit, 397, 427–429
Field boundaries, 527
Fingers, 205, 206
activation, 422
control, 419
groups, 420
Firmware, 73, 99–103, 215
allowances, 223
installation. *See* Robotic Command eXplorer
location, 101
NQC, support. *See* RCX2 firmware
opportunities, 103
options, 521
replacements, 105
upgrading, possibility, 101
usage, 218, 228, 264. *See also* Bytecode
variables, limitation, 243
FIRST. *See* For Inspiration and Recognition of Science and Technology
Flaps, 470
Flashlight beam, 356
Flex system, 159, 464
Flex tubing, 458
Flight
attitude, 485
physical equations, 480
status variable, 482
visual reference, 473
Flight simulation, 467
introduction, 468
summary, 492
Flight simulator
physical structure, 488
static version, 489
Web site. *See* FlightGear Flight Simulator
FlightGear Flight Simulator, Web site, 566
Float mode, 54, 134, 501
switching, 416
Float state, 132
Floating point math, 451
Floating point support, 105
Floating-point numbers, 217–218
Floating-point variables, 217
Floor, 111
positioning, control, 328
Floor sweeper
construction, 494–498
improvements, 497–498
programming, 497
Flow detection, RCX usage, 58
Foam rubber, sheet, 496
Follow_Line (routine), 110, 113
for (control statement), 222
For Inspiration and Recognition of Science and Technology (FIRST) LEGO League, Web site, 566
Force, 20, 280, 482, 485. *See also* Centrifugal force; Gravitational force; Push/pull force
absorption. *See* Rubber bands
application, 186
FORTH, 104. *See also* Programmable brick FORTH
forward (command), 449, 454
Forward motion, 276
FOS. *See* Fiber-Optic System
Four-in-a-row, 565
Four-legged animals, 286
Four-legged robot, 289–292
Four-legged walker, 341
Four-wheel drives, 270–274
Four-wheel steering systems, 272
Francia, Roberto, 532–533
Frequency, 118, 486. *See also* Note frequencies
Friction, 22–23, 86–88, 138, 173. *See also* Belts; Gears; Limited friction; Tracks
addition, 517
disappearance, 256
increase, 78, 85
minimization, 34
reduction, 327
usage, 504
Front assembly, 261
Front bumpers, 291
Front collisions, detection, 377
Front sensor, 380

Front wheels, 315–318, 380
  assembly, 382
  axles, 267
Front-wheel drive, 270–274
  cars, 271
    construction, 305
  problem, solving, 274
Fulcrum, 288, 295
Funnels, 209
Fusion, Mike, 551

G

Galloping, 43–46, 287
Gasperi, Michael, 81, 164, 169, 170, 548, 552
Gear wheels, 296, 551
  40t, 55
  pulleys, advantages, 34–35
Gearboxes, 156, 161, 312, 320
Geared motor, 43, 45, 46
Gearing down, Web site, 552
Gearing ratio, choice, 22
Gearing system. *See* Pulley-belt worm-24t gearing system
Gearing up/down, 20
Gearings, 172, 189, 203, 300. *See also* Drives
  24t, 499
  idling, 193
Gears, 18, 579–580. *See also* Bevel gears; Crown gear; Differential gear; Double bevel gear; Idler gears; Input; Unrolled gear; Worm gears
  8t, 21, 26–28, 46
    connection. *See* Motors
    engaging, 275
    mounting, 263
    powering, 148
  16t, 29, 35, 38, 263, 274. *See also* Dark gray 16t gear; Upper 16t gear
    with clutch, 132
    connection, 176
    engaging, 275
    matching. *See* Worm gears
    usage, 46, 131
  24t, 23–28, 38, 46, 53, 295, 476
    connection, 456
    engaging, 275
    matching. *See* Worm gear
    meshing, 138
    placement, 55, 266
    rotation, 172
  40t, 21, 26–28, 35, 323
    attachment. *See* Single-stage geartrain
    power, 189
    rotation, 137
    usage, 148, 489
  fitting, 26–31
  friction, 200
  holes, 287
  introduction, 18
  number, increase/decrease, 19–21
  placement, 26–31
  ratios, 517, 518
    choice, 250
  slack, 408
  spacings, 551
  summary, 39–40
  switch. *See* Synchronized gear switch
    usage, 276–277
  switching, 274–277
  turning, 23
  usage, 17, 517. *See also* Straight path; Strength limitation
Geartrains, 21–22, 32. *See also* Internal multistage reduction geartrain; Single-stage geartrain; Turntable
  purpose, 131
  Web site, 551
General interest sites, 549–568
General Paranoyaxc RCX Tools, Web site, 555
General Purpose Analog Interface, 169
Generators, motors (usage), 55
Geometry, 551. *See also* Legs; Robot
  introduction, 4
  summary, 15
  understanding, 3
Global functionality, 175

Global Positioning System (GPS), 235, 239
Golia II, 533
Gondolas, 329
Gordon's Brick Programmer, 105
  Web site, 553
Go_Straight (subroutine), 107, 111
GPS. *See* Global Positioning System
Grabbers
  arms, Web site. *See* Mark's LEGO Grabber Arm
  operating, 200–205
Grabbing mechanism. *See* Lifting/grabbing mechanism
Gradient pad, construction, 238
Grafcet, 555
Granularity, 395
Graphical interface, 102
Graphing system, 462
Gravitational force, 471
Gravity, 280. *See also* Center of gravity
  effect, 486
GRAY constant, 231
Gray pegs, design, 85
Grid. *See* Standard grid
Grippers, 200
Groove, 32
Ground clearance, 379
Guidance, choice. *See* External guidance; Internal guidance
Guiding IR light, 367–368
Guy's LEGO page, Web site, 556

H

H8300 (Hitachi), 99, 104
Haiku, 462–463. *See also* Automatic Haiku Writer
  generators, 56
Haiku Program, Web site, 565
Half bushings, 476
Half-beam, 85
Hall-effect detector, 169
Hands. *See* Three-finger pneumatic hand
  operating, 200–205
Hand-to-hand combat, 525
  introduction, 526
  summary, 536
Handwriting, emulation, 464
Hansen, John, 554
Hardware, construction. *See* Tic-Tac-Toe
Harvesting systems, 542
Head mechanism, 364
Heading, output, 472
Hedgehogs, 346
Height, 4
Hemispherical head, 350
Hempel, Ralph, 104, 172, 192–193, 548, 552–553, 558–559, 564
Henlich, Oliver, 561
High-speed applications, 198
High-speed race tracks, 471
High-torque applications, 156
Hinges, usage. *See* Bracing
Hitachi. *See* H8300
HiTechnic, Web site, 558
Holes, 6, 18, 46, 131. *See also* Axles; Beams; Crossed hole; Gears; Output
  beams. *See* One-hole beam
  counting, 10
  interleaving, 13
Holonomy, 151
Horizontal beams, 6–7, 10, 84
Horizontal ladder, 522
Horizontal sizes, expressing, 12–14
Horizontal stabilizer, 468
Horizontal units, expressing, 12–14
Horizontal wheel, 377
Hubs, 36
Hughes, Jim, 557
Human assistant, 396
Humidity, 80
Husby, Dean, 559
Huw, Web site, 561
Hydraulics, 181
  metaphor, 42
Hysteresis, 214, 215, 262
  advantage, 521
  understanding, 229–231
  Web site, 560

I

Ianiero, Antonio, 395, 530, 564
IBM, 401
Idea Books, 284
Ideal Gas Law, 180
Idler gears, 28, 444
  usage, 39
Idler wheel, 533
Impact force, 60. *See also* Sensors
Iñaki Zabin eta Koldoren orria, Web site, 565
InchLab, Web site, 558
Independent beam, 533
Independent rear wheels, 143
Independent tasks, running, 113–114
Independent variable, 227, 578
Indirect control, 51
Inertia, 518
  concept, 89
  definition, 87
  effect, 352
Infrared (IR)
  beam, 210, 211
  communication, definition, 445
  devices, 367
  emissions, 541
  emitting ball, 237
  interface, 99–100, 489
  LED, 70, 543, 544
  light, 58, 210, 367, 445. *See also* Guiding IR light
    emission, 538
    proximity measurement system, 165
  link, usage, 100
  messages, 423, 491
  radiation. *See* Ambient IR radiation
  rays, 444
  serial communication interface, 98
  tower, 65, 99, 100
Infrared proximity detector (IRPD), 154, 165, 170. *See also* Dual IRPD
  sensors, 166, 359
Infrared-ultrasonic beacons, 561
Initialization, 108
Ink-jet printer, 208, 455
Inner beams
  mounting, 384
Inner beams, support, 91
Inner current-limiting device, 518
Inner loop, 437
In-place steering, 139
Input, 136
  angular velocity, 39, 579
  devices, 417
  gears, 176
Input ports, 98, 101, 107, 161, 231. *See also* Robotic Control eXplorer
  dedicating. *See* Touch sensors
  multiplexing, 171
  number, 170
  status, 105
  touch sensors, addition, 432
Input/output device, 396
Insects, 286, 345
Insulating materials, 506
Insulators, 330
Integer numbers, 215
Integers. *See* Whole integers
  usage, 451
Integrated development environments, 103
Intelligence, form, 497
Interest sites. *See* General interest sites
Internal counter, implementation, 398
Internal guidance, choice, 234–235
Internal map, 388
Internal multistage reduction geartrain, 45
Internal resistors, 80
Internals, Web site. *See* Robotic Command eXplorer
International System of Units (SI), 20
Internet resources, 463
Interpolation, 214, 578. *See also* Linear interpolation
  usage, 226–229
INTERVAL constant, 481
In-the-field tuning, 449
Inverted round tiles, 135
IR. *See* Infrared

IRPD. *See* Infrared proximity detector
Italian LEGO Users Group (ItLUG), 514, 527–528, 541
  Web site, 567
ItLUG. *See* Italian LEGO Users Group

J

Jackson, Ben, 561
Jammed micromotors, releasing, 45–46
Jansson, Gustav, 551
Java
  Module. *See* JStamp Real-time Native Java Module
  systems, 105
JCX, Web site, 558
Joe's MINDSTORMS Gallery, Web site, 562
Johnny Five, Web site, 564
Johnny five-style droid
  construction, 350, 361–368
    variations, 366–367
  programming, 364
Jointed chassis, 475
Joints. *See* Constant-velocity; Universal joints
JP Brown's Serious LEGO, Web site, 561
JStamp Real-time Native Java Module, 558

K

Kangaroos, 334, 345
Kasparov, Garry, 401
Killough platform, 151, 556
Killough, Stephen, 151
Knudsen, Jonathan B., 548, 556
Koalas, 346
Kooijman, Wouter, 566
Kotay, Keith, 150

L

Label-writing machine, 462, 463
Labyrinth, 382. *See also* Makeshift labyrinth
  plan, 388
  solution, 374
  underlying cell, 383
Lachmann, Michael, 553
Lancillotto (Lancelot), 532–533
Landmarks, 236–243. *See also* Natural landmarks
Languages. *See* NQC
  usage. *See* Programming
Large-scale cleaning, 461
Laser light, 241
Laser pointers. *See* Battery-powered pen laser pointers
Lateral axis, 468, 474
  pivoting points, 475
Layered beams, 6
Layered structure, 160
Layers, locking, 84–87
LDraw, Web site, 553
Learning by example, 465
LED, 65. *See also* Infrared; Sensors
  illumination, 66
  turning on, 397
  usage, 438
Left bogie, 315
left (command), 454
Left drive wheels, 579
Left side bumper, 377
Left side rule tactic, 382
Left-edge sensors, 254
Left-side strategy, usage, 374–376
Legato effect, 120
Legged robots, 128
LEGO
  board. *See* All-LEGO board
  Compatible Pneumatic Solenoid Valve, Web site, 559
  Construction Site-Ideas, Web site, 559
  devices. *See* Non-LEGO devices
  Direct, 162
  engineers, 8
  FetchBot, Web site, 560
  inventory, 320
  Online Shop, 323
  originals, 163
  parts. *See* Non-LEGO parts
  Parts Reference. *See* LUGNET

Web site, 549
Pneumatics, Web site, 559
Robots, Web site, 568
Set Inventories, Web site, 557
Shop At Home, Web site, 557
supplies, 243
Switch Multiplexer, Web site, 558
Train Depot, Web site, 562
tree, 346
Users Group, 514
Lego Biped Links, 562
LEGO Users Group Network (LUGNET), 315, 515
LEGO Parts Reference, 85
Web site, 549
legOS, 52, 123, 218, 243, 463
emulator, 555
switching, 264
usage, 104, 264, 465, 480
Web site, 553
LegoSim, Web site, 555
Legs, 281, 305
assemblies, 196
attachment, 339
connection, 291
construction, 287–289. *See also* U-shaped legs
geometries, 289
geometry, 295, 301. *See also* Turtle
interlacing, 299–302
movements, 308
opening/closing, 323
synchronization, 293
terminal point, 288
leJOS, 218, 243
switching, 264
usage, 105, 465, 480
Length, 4
LEO & LEGO, Web site, 556
Level crossings, 329
Lift, 470, 483–484
output, 472
Liftarms, 85, 155, 160, 476
usage, 189, 203
Lifting arm, operation, 499
Lifting mechanism, 402, 408, 464
Lifting/grabbing mechanism, 409
Light
emitting, 65
follower, 64
measuring. *See* Reflected light
reading. *See* Ambient light
sources. *See* Driving light source; External light sources
value, 109, 262
variation, 344
Light brick, 431, 432
Light sensors, 64–70, 75, 170, 504. *See also* Differential light sensor; Face-down light sensor
addition, 417
allowance, 531
color, differentiation, 236
detection ability, 543
mounting, 256, 354
placement, 242
position, 264
raw values, 217, 228
readings, 383, 541
stacking, 256
trick, Web site, 552
usage, 241, 254, 431, 464
Light/dark areas, readings, 256
Light-driven robot, 64
Limit Switch, 171
Limited friction, 135
Limited slip differential, 144
Line following, 67–69, 236, 256–265, 519–520
competitions, 258
interruptions, 519
optimization, 264–265
program. *see* Differential drive
Line variable, 109
Linear interpolation, 226, 578
Linear motion, 180
Links. *See* Chain links; Track links; Tread links

usage. *See* Infrared
Load, combination. *See* Chassis/modularity/load
Load-bearing axles, 91
support, 89
Lobsters, 346
Lock motor. *See* Differential gear
Locking beams, importance, 86
Locking layers, 84
Locking scheme, 15
Locomotives, 328, 563
Logging. *See* Data
Logo Foundations, Web site, 565
LOGO systems, 443
LOGO turtle, 462, 539
construction, 443–448
creation, 442–455
differential drive, usage, 455
language, choice, 453–454
programming, 448–453
variations, 454–455
Web site, 565
Long joined axle, 402
Longitudinal axis, 468, 474
Longitudinal beam, 323. *See also* Parallel longitudinal beams
Loopholes, 514
Looping. *See* Program
Loops. *See* Inner loop; Mazes
usage, 434
Loran system, 239
Lorenzetti, Sergio, 532
Low duty cycle, 52
Low-level subroutines, 107
Lucas, George, 350
Lucas, G.W., 561
LUGMAP, Web site, 566
LUGNET. *See* LEGO Users Group Network
Luis Vitton Cup 2000, 330
Luna Rossa, 330
Lunar Rover, 563
Lund, Henrik Hautop, 543, 567
Lustre, 555

M

Machine code instructions, 100
Machine Learning, Web site, 550
Machinery, 18
Macros, 416
Macs Robotics, Web site, 557
Magnetic switches, 154
Magnetism, 229
Magnets, 169
Mainsail, 330
Makeshift labyrinth, 381
Makino, Shigeru, 555
Mantissa, 217
Marbles, 206
Mark's LEGO Grabber Arm, Web site, 560
Maron, Chad, 562
Martin, Fred, 550, 551
Masetti, Paolo, 104, 264
Mass, 87
Massachusetts Institute of Technology (MIT), 119, 237, 392, 442, 543
Programmable Bricks, 550
Master-slave configuration, 406
Masthead, 330
Matching. *See* Distances
quality, 570
Mathematical Atlas, 560
Mathematical functions, 015
Mathematics, 213
introduction, 214–215
summary, 232
Mathematics, cheat sheet, 577
MAX constant, 435
maxlight_left (variable), 357
maxlight_right (variable), 357
Maxwell, James Clerk, 540
Maxwell's Demons, 540–542
Web site, 567
Maze Runner, 372
construction, 377–382
limitations, 382
programming, 380
variations, 381–382
Maze solutions, 371, 520

introduction, 372
summary, 389–390
Maze Solver, 372, 393
construction, 382–389
programming, 387–389
Maze Solving Algorithm, Web site, 564
Mazes
creation, 381
drawing, 387
exiting, 373–377
strategies, 376–377. *See also* Left-side strategy; Right-side strategy
exploration, 388
loops, 375
representation, memory usage, 387
scanning, 388
MazeWalker, Web site. *See* MINDSTORMS
MDOF. *See* Multi-Degree-of-Freedom
Mechanical adding/subtracting device, 38
Mechanical configuration. *See* Robot
Mechanical couplings, 489
Mechanical device, 208
Mechanical energy, 42
Mechanical interface, construction, 408. *See also* Chess
Mechanical resistance, 134
Mechanics, 154
Mechatronics, 98
Media Lab facility, 392
Melody, construction, 422
Memory
array, 394
limitations, 388
requirement, 377
usage. *See* Mazes
Messaging system. *See* Two-byte messaging system
Metric system, 72, 482
MIBO robotic dog, 345, 563
Micromotors, 43, 46, 464
releasing. *See* Jammed micromotors
Micromouse Maze solving competitions, 564
MicroScout, 161–162
Microscout (programming), Web site, 555
Micro-switches, 165
Microwave, 502
MIDI. *See* Musical Instruments Digital Interface
Mientki, Stef, 555, 558
Mike's LEGO MINDSTORMS Page in Wien, Web site, 566
Milk Guard, 494
construction, 498–502
improvements, 502
programming, 501
usage, 501
warnings, 501–502
Millington, Huw, 561
Milton Bradley, 432
Mindfest
(1999), 119, 237, 392, 543, 564
TTT, 396
Web site, 567
MindScope, Web site, 555
Mindsensors, Web site, 558
MINDSTORMS
Add-Ons, Web site, 557
box, 13, 23, 302, 372, 426, 499
contents, 58
Test Panel, 66
CD-ROM, 99, 101
creations, 392
equipment, 140, 377
expansion sets, 155
Info Center, Web site, 563
Inventions, Web site, 566
MazeWalker, Web site, 564
programmers, 106
programming, 105
set, 383, 527, 539. *See also* Star Wars MINDSTORMS sets
software, 100
system, 234, 287, 426
Web site, 549, 567
MINDSTORMS kit, 13, 30–33, 36, 158
contents, 45, 47, 50
documentation, 98

limitations, 154
parts, 90
inclusion, 291, 312, 362, 494
noninclusion, 69, 71, 74, 77, 337, 499
recommendation, 162
tracks
availability, 361
usage, 366
trick, 266
usage, 206, 234, 250, 392–393
Mini Sumo, 527
rules, 533
tournament, 531
MIT. *See* Massachusetts Institute of Technology
MLCad, Web site, 553
Mobile platform, 160
Mobile robots, 86, 130
Mobility, 127
introduction, 128
summary, 151
Modular code design, benefits, 112
Modular construction, 87
Modular robots, design, 84
Modularity
combination. *See* Chassis/modularity/load
maximization, 87–88
Modulo operator, 222
Moles, 334, 346
Momentum, 532
Monkey bars, 522
Monophonic device, 120
Monophonic sound, 123
Motion. *See* Backward motion; Linear motion; Straight motion
control. *See* Synchro drive
random pattern, 497
receiving, 156
resistance, 54
reversal, 175
transferring, 29, 33, 35, 146
transmitting, 30
Motor Mis-Match, Web site, 551
Motor-driven valve switches, 200
Motors, 22, 49, 188, 206, 320. *See also* Direct current; Electric motor; Geared motor; Micromotors; Servo motors; Steering; Stepper motor; TECHNIC motor; Ungeared motor
addition, 413
attachment. *See* Robotic Control eXplorer
braking, 53–54
complexity, reduction, 315
connection. *See* Turning
control, 41
introduction, 42–43
summary, 56
coupling, 55–56
decoupling, 45
direction, reversing, 531
expense, 48
location, 305
mounting, 46–49
operation, 423
power
control, 52–55
supply, 140
releasing. *See* Jammed micromotors
removal, 47
reversing, 518
saving, 320
shaft, 8t gear connection, 379
short-circuiting, 50
size, 361
stalling, 43
support, 420
testing, 50–51
usage, 315, 351. *See also* Generators
wiring, 49–52
Motor/Sensor Expander, 559
Mounting brackets, 46
Mounts. *See* Vertical mounts
Mouse
construction, improvement, 338–339
creation, 334–339
Movable plate, 158
Movable shaft, 275
Mozilla Public License, 104

Mueller, Frank, 555
Multicolor movable block, 69
Multi-Degree-of-Freedom (MDOF) vehicles, 150
Multilegged walkers, 345
Multiplatforms, 103
Multiple sensors, connecting, 78–80
Multiplexer, 438
Multiplexing, 170. *See also* Input ports; Output ports
Multiplication, 215–218, 228, 265
Multiprogram downloading, 105
Multistage reduction
  geartrain. *See* Internal multistage reduction geartrain
  system, 21
Multitasking, 113
Munafo, Robert, 177, 558
Music
  introduction, 118
  playing, 117, 119–122
  summary, 124–125
Musical Instruments Digital Interface (MIDI)
  files, conversion, 122–124
  MIDI2RCX, 556
    conversion utility, 122, 124
  Web site, 556
Musical instruments, playing, 411
  introduction, 412
  suggestions, 424
  summary, 424
Music-Robots, Web site, 556

N

Nagata, Joe, 280, 549
Native Java Module. *See* JStamp Real-time Native Java Module
Natural landmarks, 498
Navigation, 535, 579–580. *See also* Playing field; Room; Tiles
  abilities, 166
  algorithm, 545
  allowance, 543
  capabilities, 236
  control, 175
  problems, 209
    solving, 167
  system, 374
  Web site. *See* Robot
Ncm. *See* Newton-centimeter
Negative acceleration, 483, 484
Nelson, Russell, 550
Newton-centimeter (Ncm), 20, 25, 26
Newton-meter (Nm), 20
NiMH batteries, 517
NIST/SEMATECH, 560
Nitin, 558
Nm. *See* Newton-meter
No-contact switches, 169
Noga, Markus, 104
Non-LEGO custom devices, 154
Non-LEGO devices, 164
Non-LEGO electricity sources, 51
Non-LEGO parts, 163, 323, 446
Non-LEGO sensors, 81, 164
Nonlinear systems, 226
Nonoriginal components, 367
Nonoriginal parts, addition, 80
Nonpivoting wheels, 150
Nonsteering SHRIMP rover, 320
Nose-ups, 488
Not Quite C (NQC), 98, 215, 243
  % symbol, 222
  access, 114
  API Programmer's Guide, Web site, 554
  code, 105, 122, 124, 358
    fragment, 269
    solution, adoption, 483
    usage, 220–221, 231, 258, 434
  compiler, 103
  documents/tutorials, 103
  encapsulation, 103
  features, 106
  implementation, 245, 452
  installation, 103
  instructions, 118
  program, 465

programming, 263
request, 108
subroutine, 398
support. *See* Robotic Command eXplorer
usage, 102–103, 106, 395
version, 104, 480
Web site, 553
Note frequencies, 575–576
Note Names, Web site, 556
Notepad, 103
Note-taking, 373–374, 389
NQC. *See* Not Quite C
NQCEdit, Web site, 554
Numbers. *See* Floating-point numbers; Integer numbers; Whole numbers
Numerical analysis, 215
Numerical Analysis, Web site, 560
Numerical Methods, Web site, 560

O

Objects, finding, 199, 209–212
introduction, 200
summary, 212
Objects, grabbing, 199
introduction, 200
summary, 212
Obstacles, 59, 70. *See also* Static obstacle
detection, 73, 148, 166, 250, 265, 372
interference, 254
variations, 255–256
hitting, 112
overcoming, 520
removal, 520
usage, 432
Odometry, 243
Web site. *See* PID-Based Technique for Competitive Odometry and Dead Reckoning
OHP. *See* Omnidirectional Holonomic Platform
Omnidirectional Holonomic Platform (OHP), 151
Omni-directional sensor
connection, 149
usage, 148
On state, 78, 79
One-hole beam, 69
on_left variable, 488
On/off switches, 315
on_right variable, 488
Open cells, 387
Open Source project, 105, 555
Operating hands, 200–205
Operations, number, 220
Opposing rubber bands, 205
Optic fibers, 367, 397
Options, expansion, 153
introduction, 154
summary, 177–178
OR configuration, 78
Organizing committee, 515, 516
Orientation, 375
change, 23, 443, 450, 579–580
losing, 235
Orthogonal direction, 30, 474
Oscillations, 230
Ostriches, 334, 345
Otto, 207, 208
Otto Details, Web site, 564
Outer beams, support, 91
Output. *See* Bi-directional outputs
angular velocity, 39, 579
axles, 172, 174
conversion, 167
direction, 28
holes, 398
Output ports, 98, 106, 175, 409. *See also* Robotic Command eXplorer
exiting, 182
multiplexing, 171
number, 170
touch sensors, connection, 432
Output torque, 26
Overflows, 217, 218
Overloads, protection, 52
Overmars, Mark, 554
Oysters, 345

P

Pacific NW LEGO Train Club (PNLTC), Web site, 563
Pacing, 43–46
Pagliarini, Luigi, 543
Painting. *See* Fast painting
Paper feeding motor, 464
Paper transport system, 456
Papert, Dr. Seymour, 442
Paradigm shift, 519
Parallel beams, 301
Parallel longitudinal beams, 420
Parallel wheels, 37
Parallelogram, 316
Parts, 22, 47, 143, 271, 293, 299. *See also* Non-LEGO parts
  acquisition, 154–163
  addition. *See* Non-original parts
  colors, choice, 160–161
  purchasing, 161–163
  Reference. *See* LEGO Users Group Network
    Web site. *See* LEGO
  ruining, 25
  usage, 146, 151, 350, 372
  washing, 461
Passive sensors, 74, 101, 161
Passive stations, usage, 242
Patil, Nitin, 170, 171
pbForth. *See* Programmable brick FORTH
Peak values, 218
Pegs, 169, 267. *See also* Axle-pegs
  design. *See* Gray pegs
  number, 393
  rows, 384
  technique, 408
  usage, 387. *See also* Black pegs
Pen carriage, removal, 458
Pen motor, 448
Pen movements. *See* Up/down pen movements
pendown (command), 448, 454
PEN_TIME constant, 449
penup (command), 448, 454
Percentage errors, 215
Perimeter, 375
Perpendicular axle, 262
Perpendicular beam, 6, 7
Perpendicular direction, 455, 474
Pet feeder, 509
Peter, Andreas, 558
Phillips, Chris, 169
Photoelectric cell, 69
Pianist
  changes/improvements, 423–424
  construction, 417–422
  creation, 417–424
  programming, 422–423
  scores, suggestions, 423
  strengthening, 419
Piano keys, resistance, 417
PID. *See* Proportional Integral and Derivative
PID-Based Technique for Competitive Odometry and Dead Reckoning, Web site, 561
Pin, Francois, 151
Pin-axle connectors, 323
Pinball machine
  construction, 426–431
    improvements, 432
  creation, 426–432
  programming, 431
Ping-Poing, 438
Pins, usage. *See* Black pins
Pipe junction, 507
Pipe network, 508
Pistons. *See* Electric Piston
Pitch (angle), 468, 471, 482–483
  adjustment, 475
  compensation, 485
  control, 473, 474, 476
  increase, 484
  measurement, 476
  movement, 475
  values, 472
Pitch (sound), 118
  decoding, 423

sensor, 166
PITSCO, 163
PITSCO-DACTA, Web site, 557
Pivot drive, 150
Pivoting axis, 143
Pivoting axle, 335
Pivoting point. *See* Lateral axis; Robot
Pivoting wheel, 91
Plane. *See* Vertical plane
behavior, 470
Planetary gear, 551
Plant sprinkler
construction, 502–508
improvements, 505–508
programming, 504–505
usage, 504–505
Plates, 5, 8, 47, 85. *See also* Baseplate; Electric plates; Movable plate; TECHNIC plates; Transparent plates
combination. *See* Beams
counting, 7
dimension, 302, 394, 496
layer, 84
placement, 155
usage, 161, 496
Platforms. *See* Differential drive; Killough platform
motion, 487
Playing field, navigation, 540
play_sequence (subroutine), 434–436
PlaySound (command), 118, 124
PlayTone (command), 119, 124
Pliers, 203
Pliers-down command, 407
Pliers-up command, 407
Pneumatic arm, 561
Pneumatic cylinder, 182, 202, 206, 503
operation, 190
Pneumatic devices, 159
Pneumatic engine, construction, 194–198
Pneumatic hand. *See* Three-finger pneumatic hand
Pneumatic parts, 200
Pneumatic pumps, 503
Pneumatic system, 180, 186, 503
Pneumatic valve, 507
Pneumatics, 179, 508
interfaces. *See* RCX-to-pneumatics interfaces
introduction, 180
summary, 198
usage, 464
Web site. *See* LEGO
PNLTC. *See* Pacific NW LEGO Train Club
Pointer. *See* Circular pointer
Polarity, 49, 438
reversing, 42
switch, 51, 187, 312, 320, 480
requirement, 315
rubber bands, usage, 316
usage, 78, 193–194
Porcupines, 346
Ports. *See* Input ports; Output
conservation, 190
increase, solutions. *See* RCX
limitations, solutions, 170–172
number, limitation, 172
Position, 375. *See also* Resting position
control
allowing, 385
system, 377
marking, 393
Positioning, 24, 233, 245. *See also* Absolute positioning; Relative positioning
accuracy, 26
introduction, 234
summary, 246
Postfix notation, 104
Potential energy, 24
Power, 43, 53, 531. *See also* Air power; Applied power; Braking
control. *See* Electric motor; Motors
level, 53
trimming, 130
portion, 483
removal, 54
source, 26
supply. *See* Robotic Command eXplorer

supplies, 509
transferring, problem, 49
Power activated sensors, 397
Precise positioning, 538–539
Precision, 537
combination. *See* Speed
introduction, 538
summary, 545–546
Predefined behavior, 101
Pre-match test time, 528
Pressure, 180, 181, 505. *See also* Air pressure; Atmospheric pressure
exertion. *See* Touch sensors
requirement, 505
switch, 405, 559
Primary arm, 405
Program
looping, 59
slots. *See* Robotic Command eXplorer
statements. *See* Robotic Control eXplorer
Programmability, limitation, 161
Programmable brick FORTH (pbForth), 462, 465
switching, 264
usage, 104–105, 480
Web site, 553
Programmable Bricks, 554. *See also* Massachusetts Institute of Technology
Programmable bricks, 100–101, 103, 442
Programming. *See* LOGO turtle; Robot
code, 214
environment, 480
languages, 62
usage, 103–105
tools/environments, usage, 105
Projects, 249, 364
introduction, 250
summary, 277–278
Web site. *See* Robotics
Proportional Integral and Derivative (PID) approach, 561
Prototype. *See* Robot
construction, 538
Prototyping, 26
Proudfoot, Kekoa, 103, 553
Proximity detection, 70, 211, 359
exclusion, 541
Proximity detectors. *See* Infrared proximity detector; Ultrasonic proximity detectors
Proximity sensing, 522
Pseudo-code, 62, 93, 106
Pulley-belt setup, 202
Pulley-belt systems, 32, 262, 364
Pulley-belt worm-24t gearing system, 354
Pulleys, 49, 51
advantages. *See* Gear wheels
combination, 33
friction. *See* Belts
sizes, 264
slippage, 143
usage, 31–35
Pulling cables, 329
Pulse width modulation (PWM), 52
Pumping technique, 508
Pumps, 181–187, 191. *See also* Pneumatic pumps
Push buttons, 98, 211
Push/pull force, 197
PWM. *See* Pulse width modulation
Pyroelectric sensor, 167
casing, 168
Pythagoras, 10
theorem, 11, 14, 15

Q
QC programming system, 565
QC, Web site, 554

R
R2-D2 Builders Club, Web site, 563–564
R2-D2 clone, 564
R2-D2-style droid, 364, 380
construction, 350–360
variations, 359–360
programming, 356–360
Races, 32–33, 522
Racing. *See* Cars; Drag racing

Rack and pinion assembly. *See* Upside-down rack and pinion assembly
  usage, 402, 458
Rack and pinion scheme, 273
Rack and pinion steering
  assembly, 157
  mechanism, 142
Rack and Pinion, Web site, 562
Racks, 158. *See also* Switching rack
Radians
  conversion, 451, 580
  usage, 579
Radii, sum, 27
Radio beacons, following, 239–243
Radio frequency, 101
Radio-controlled models, 171–172
Radius turns, 142
Rails, 47, 91
Rajcok, Mark, 554
RAM. *See* Random Access Memory
Random Access Memory (RAM), 99–101, 103
Random behavior, 344
Random turns, execution, 376
Random values. *See* Turns
Raster graphics, 124
Ratchets, 172
  beams, 189
Raw values, 227. *See also* Light sensors; Sensors; Temperature
  understanding, 66–67
  usage, 170
R/C Servos. *See* Using R/C Servos with the RCX
RCX. *See* Robotic Command eXplorer
RCX-to-pneumatics interfaces, 200
Reactive Languages, Web site, 555
Readability, 112
Reading colors, 67
Reading head, 383
Readings, calibration, 60, 68
Read-only memory (ROM), 101
  code, 99
Real COG, 478
Real-life robots, 312
Real-world application, 34
Real-world objects, 412
Rear bumpers, 291
Rear wheels, 145, 315–318. *See also* Independent rear wheels
Rechargeable batteries, 544
Rectangular sizes, 5
Recycling, 165
Reduction
  geartrain, 404. *See also* Internal multistage reduction geartrain
  ratio, 24, 33, 45, 156, 474–476
  stage, 21
  system, 34. *See* Multistage reduction
Reed bulb, 169
Reed switch, 169
Reference points, usage. *See* External reference points
Reflected light, measuring, 66–67
Reflection, 65, 541
Reflector, 243
Regulated adapter, 507
Relations, highlighting, 47
Relative distance, 65
Relative errors, 215
Relative positioning, 234, 243–246, 374
Remote control, 50, 93, 478
Remote-controlled car, 472
Remove_Obstacle (subroutine), 110, 113
Resistance, 53. *See also* Mechanical resistance
Resources, 547
  introduction, 548
Resting position, 395
Resultant, 280
Retractable tail, 344
Reusability, 112
Reverse direction, 291
Reverse Polish notation (RPN), 104, 454
Richard, Martin, 55
Right bogie, 315
right (command), 454
Right drive wheels, 579

right (routine), 452
Right-angled triangle, 10
Right-edge sensors, 254
Right-side strategy, usage, 374–376
RIS 1.5 equipment, 250
RIS parts, 289
Röblitz, Thomas, 555
RoboCup Junior 1999, Web site, 567
Robo-Rats Locomotion, Web site, 556
RoboSports Set, 161
Robot. *See* Bipeds; Competing robots; Differential drive; Dual RCX robot; Four-legged robot; Legged robots; Light-driven robot; Mobile robots; Six-legged robots; Six-legged steering robots; Two-legged robots; Wheeled robots
  architecture, 243
  assembly, 48
  behavior, 107
  blueprint, 98
  body, 60
  centerpoint, 244, 579
  construction, 87, 260, 287, 515. *See also* Useful robots; Walking robots
  contact. *See* Walls
  course correction, 68
  design, 106, 508–509, 516. *See also* Antidote robot; Modular robots; Robust robots
  direction, change, 148
  dismantling, 47
  fixed-head version, 351
  generality, 256
  geometry, 322
  head, 419
  home base, 243
  manual triggering, 211
  mechanical configuration, 521
  motion, 140, 209
  Navigation, Web site, 561
  orientation, 579, 580
  performance, 168, 264
  physical architecture, 111
  physical properties, 449
  physical structure, 112, 113
  pivoting point, 443
  position, 580
  programming, 327, 359, 384
  prototype, 22
  rotation, 356
  scale, 312
  software. *See* Tic-Tac-Toe
  speed, 264
  stability, 94
  turning, 295, 315
  usage, 106
  warnings, 501–502
Robot Arena, Web site, 567
Robot Pages, Web site, 554
Robotic animals, 333
  introduction, 334
  summary, 346
Robotic architecture, 520, 556
Robotic arm, 24. *See also* Broad Blue
Robotic car, 275
Robotic Command eXplorer (RCX), 52, 87, 164
  addition, 346
  calculation, performing, 407
  Command Center, 554
  computational process, 388
  console, 50
  control, 50
  conversion. *See* Electrical signals
  covering, 344
  damage, 45, 328
  decoding process, 170
  definition, 98–101
  detachability, 88
  disconnection, 54
  display, 502
    updating, 481
  external power source, supply, 506–507
  firmware, installation, 521
  free port, 320
  function, understanding, 99
  hosting, 252
  implementation, 54

input ports, 62, 78
installation, 303
instructions, 231
interception, 429
interface, 241
internals, Web site, 553
interpretation ability, 167
motors, attachment, 55
movement, 94
ordering, 60
output port, 51, 397
placement, 90–91, 93, 242, 443–445, 492
polling. *See* Sensors
port, 51, 206, 381
increase, solutions, 172–177
position, 89, 444
power, 102
processing power, 468
processor, 100
program
slots, 101
statements, 124
programming, 97, 453, 504
introduction, 98
summary, 114–115
projects. *See* Dual-RCX projects
protocol, 407
RCX2 firmware, NQC support, 103
removal, 289, 337
robot. *See* Dual RCX robot
score, display, 427
size, 361
sliding, 305
sound system, 412
tools, Web site. *See* General Paranoyaxc RCX Tools
usage, 171, 186, 328, 541. *See also* Flow detection; Valves
weight, 88
whole numbers, usage, 67
Robotic Command eXplorer (RCX) Code, 99–100, 215, 264
drawbacks, 102
hidden power, 102
suitability, 120
usage, 101–102, 108
Robotic cranes, 397
Robotic dog. *See* MIBO robotic dog
Robotic hand, opening/closing, 159
Robotic Invention System, 91
Robotic parking, 328
Robotic sumo. *See* Mini Sumo
attack strategies, 530–534
construction, 526–530
defense system, 534–535
rules, setting, 527–528
speed, usage, 532–533
strength/traction, maximization, 528–530
testing, 535–536
transmission, usage, 533
tricks, 533–534
Robotics, 43, 87, 114, 118, 540
benefit, 214
Discovery Set, 161, 334
experience, 172
Introduction, Web site, 561
Projects, Web site, 558
uses, 426
Web site, 552
Wiki, Web site, 550
Robots, Web site. *See* LEGO
Robust robots, design, 84
Rolighed, Soren, 565
Rolighed's LEGO MINDSTORMS, Web site, 565
Roll, 468
ROM. *See* Read-only memory
ROM routines, 554
Room
exploration, 250–256, 372
navigation, 494
Rossa. *See* Duna Rossa; Luna Rossa
Rotating bumper, 557
Rotating devices, 432
Rotating support, 288, 489
Rotating wheel, angular velocity, 38
Rotation. *See* Bevel gear; Worm gear
counting, 72

degree. *See* Turntable
direction, 35
change, 42
Rotation sensors, 34, 58, 71–73, 243–244. *See also* Throttle
addition, 463
connection, 330, 427, 450
count, 451
increment, 579, 580
emulation, 75–78, 396, 539
function, explanation, 71
incorporation, 326
involvement, 276
measurements, 476
owning, 255
performance, 131
positions, 407
required increment, 580
resolution, 72, 449
ticking, 481
usage, 130, 161, 170, 406, 450. *See also* Dead reckoning
Rotational sensors, Web site, 552
ROTATION_TIME constant, 357, 358
Round tiles. *See* Inverted round tiles
Round trips, 519
Route, tracking, 376
Routines, 416
RPN. *See* Reverse Polish notation
rtlToronto. *See* Toronto Users Group
Web site, 566
Rubber bands, 33, 149, 193. *See also* Opposing rubber bands
force, 256
absorption, 60
linkage, 458
need, 420
placement, 446
usage, 177, 203. *See also* Polarity; Steering
Rubik's Cube solver, 561
Rudder, 468, 471
control, 472
pedals, 468
Rutkas, Clint, 564

S

S6 Details, Web site, 562
S15 Details, Web site, 565
S16 Details, Web site, 565
S17 Details, Web site, 563
S18 Details, Web site, 557
Sailing tricycle, 330
Sali, Fabio, 272
Sato, Jin, 280, 345, 549, 563
Scanning. *See* Mazes; Tic-Tac-Toe
technique, 409
Schalburg, René, 563
Schilling, David, 540, 567
Schumm, Tom, 79, 552
Science, refresher, 180–181
Scout, 103, 161. *See also* MicroScout
programmable unit, 504
programming, Web site. *See* Microscout
Search engine, 515
Secondary arm, 405
rotation, 404
Sectors, colors, 76
Self-calibration process, 256
Self-centering, 143
Self-powered flying machine, 468
Self-protection mechanism, 498
Semaphores, 329
SENSOR_BANK, 481
SENSOR_PITCH, 482
Sensors. *See* Active sensors; Differential light sensor; Face-down light sensor; Front sensor; Light sensors; Non-LEGO sensors; Passive sensors; Rotation sensors; Temperature; Touch sensors
addition, 268
additional types, 80–81
connecting. *See* Multiple sensors
construction. *See* Custom sensors; Slope sensor
count, 579
emulation, 58. *See also* Rotation sensors; Touch sensors
exposure. *See* Temperature
impact force, 60

initialization, 380
insertion, 426
introduction, 58
LED, 7
number, reduction, 25
raw values, 578
RCX polling, 71
reading, 57
conversion, 73
resolution, 579
summary, 81
tips/tricks, 75–80
usage. *See* Omni-directional sensor; Straight path
SENSOR_THROTTLE, 482
Serial communication interface. *See* Infrared
Service packs, 161, 162
Servo motors, 171
power, 172
Set Inventories, Web site. *See* LEGO
Sethna, Jim, 560
SetSensor statements, 108
Sevcik, Pete, 165, 171, 558
Sewer Rat, 562
Shafer, Joel, 540
Shapes, 26–27
Shock absorbers, 158
Shoot the Tank, 439
Short circuit, 53
Short Circuit, 350
SHRIMP Details, Web site, 562
SHRIMP rover. *See* Non-steering SHRIMP rover
abilities, 313
construction, 320–321
creation, 312–321
power, 321
turning, 315
SHRIMP Web site, 562
SI. *See* International System of Units
Side lengths, 12
Side sensor, 380
Side-by-side start, 530
Side-to-side direction, 455
Simon-Says game
construction, 432–434
creation, 432–438
programming, 434–438
variations, 438
Simple averages, 218–223, 578
Single-motor steering drive, 338
Single-motor synchro drive, 175
Single-motor tricycle drives, 175
Single-stage geartrain, 40t gear attachment, 379
Sink the Ship, 438
Six-legged robots, 284
Six-legged steering robots, 293–299
SIZE (constant), 221–223
Size constraints, 527
Sizes, 26–27
expressing, 4–5. *See also* Horizontal sizes
Skibot, 321
testing process, 327
Skidding, avoidance, 140
Skid-steer drive, 360
construction, 138–140
Skier, creation, 321–327
Skis. *See* V-shaped skis
design, 325
poles, 322, 326
Skogsrud, Simen Svale, 552
Skunk Works. *See* Clint Rutkas' Skunk Works
Slave configuration. *See* Master-slave configuration
Slip differential. *See* Limited slip differential
Slippage, 35, 315. *See also* Belts; Pulleys
problem, 34
Slope sensor, construction, 277
Slow speed/high torque application, 46
Smalltalk, 553. *See also* Dolphin Smalltalk
Smooth bumpers, design, 532
Smooth surfaces, 140
Smooth turns, 91
Ackerman steering, usage, 142
Smoothing. *See* Exponential smoothing
Soccer, playing, 543–545

Software, 50, 61, 98. *See also* MINDSTORMS; Tic-Tac-Toe
  algorithms, usage, 521
  generality, 256
  reactivity, 264
  simplicity, 416
  usage, 75
Soh, C.S., 191, 192, 197, 559
Solorzano, Jose, 105
Sound effects, 431
Sound frequency decoders, 154
Sounds
  introduction, 118
  playing, 117
  summary, 124–125
Source files, editing, 104
Space. *See* Three-dimensional space
Sparring partner, 535
Specialized components, 155–161
Spectrum Educational, 163
Spectrum Educational Supplies, Web site, 557
Speed, 19, 35, 470, 481–483, 579. *See also* Robot
  control, 53
  difference, 39
  direction, combinations, 129
  doubling, 137
  gaining, 264
  high torque application. *See* Slow speed/high torque application
  optimization, 516–518
  precision, combination, 518–522
  reduction, 24, 56
  regulator. *See* Train speed regulator
  usage. *See* Robotic sumo
Spiders, 345
  Robot creation, Web site, 563
Spoke wheels, 529
Squaring, 6–10
Squirrels, 334, 345
Stability. *See* Dynamic stability
Stabilizer. *See* Horizontal stabilizer
Stack construction, beam usage, 7
Stacked beams, 84
Stall situations, 52, 472
Standard beams, usage, 13
Standard grid, 10
Star Wars, 350
Star Wars MINDSTORMS sets, 161
Static COG, 286
Static obstacle, 428
Static simulator, 489
Statics, 280, 287
Station, expected heading, 242
Stationary wheel, 111
Status variables, 485
Stealing the Cube competition, 542
Steer axle, 316, 318
Steering. *See* Ackerman; In-place steering
  angle, 145
  arms, 157
    replacement, 143
  assembly, 261, 274. *See also* Cars
    rubber bands, usage, 382
  centering, 269
  drive, 145, 150, 381. *See also* Single-motor steering drive
    architecture, 258
    construction, 140–144
    variation, 335
  mechanism, 140, 264. *See also* Rack and pinion steering
    space, 318
  motor, 157, 262
    connection, 274
    switching, 269
  position, 265
  robots. *See* Six-legged steering robots
  systems, 316. *See also* Four-wheel steering systems
  vehicles. *See* True-life steering vehicles
  wheel
    control, 262, 381
    drive, transferring, 270
Steering-driven wheels, 175
Steer_left (subroutine), 269
Steer_right (subroutine), 269, 270

Stehlik, Rob, 562, 565
Step-by-step flow, 114
Stepper motor, 177
Stones, 409
Stormont, Brian, 80, 552
Straight and Narrow, Web site, 556
Straight line, equation, 578
Straight motion, 269, 276, 454
  resuming, 298
Straight path
  casters, usage, 134–135
  gears, usage, 131–134
  maintenance. *See* Differential drive
  sensors, usage, 130–131
Strength, 514
  limitation, clutch gear usage, 24–26
  maximization. *See* Robotic sumo
Strijbosch, Ronald, 554
String instruments, 424
Structures, loading, 88–90
Studs, 4, 6, 362, 451
  dimensions, 46, 527
  inclusion, 31. *See also* Axles
  measurement, 30
  position, 426
  radius, 27
  size limit, 530
Studs-down, 534
  construction, 385
  usage, 146, 325, 384
Studs-front, 143
Subroutines, 107. *See also* Low-level subroutines
  usage, 462
  writing, 417
sum (variable), 222
Sumo. *See* Mini Sumo
  construction. *See* Robotic sumo
  contests, 528
Supercar, 272
Supporting base, 283, 352
Supporting point, 281, 342
Suspension, 272
Sutherland, Richard, 540
Swing bridges, 329
Swinging chassis, 297
Switches. *See* Limit Switch; Magnetic switches; Micro-switches; Motor-driven valve switches; No-contact switches; On/off switches; Polarity; Pressure; Reed switch
Switching points, 329
Switching rack, 176
Symmetrical assembly, 476
Synchro drive, 245, 268, 270–271. *See also* Single-motor synchro drive
  construction, 146–149, 305
  motion control, 174
Synchro Drive, Web site, 556
Synchronization, 305, 341. *See also* Legs
Synchronized gear switch, 276
System operations, 100

T

Tanks, 185–186
Tape writer, 455–463, 565
  construction, 455–461
  content, choice, 462–463
  programming, 462
  variations, 463
Target language, 123
Tasks, running. *See* Independent tasks
TCL-RCX, Web site, 555
TECHNIC air tanks, 329
TECHNIC brick, 6
  dimensions, 352
TECHNIC connectors, 157
Technic Double-Acting Compressor, Web site, 559
TECHNIC models, 8, 155, 272
TECHNIC motor, 45
TECHNIC parts, 91, 155
TECHNIC plates, 143, 157
Technic Puppy Journal, Web site, 562
TECHNIC sets, 154, 156, 158, 162
TECHNIC system, 19
Technica, Web site, 557
Teeth, 264

counting, 18–19
Temperature, 463. *See also* Ambient temperature
programmed value, 500, 502
raw values, 578
sensors, 74–75, 227, 494
exposure, 502
Temporization, 454
Terminal emulator, 104
Test Panel. *See* MINDSTORMS
Testability, 112
Text editor, 103
TFM, Web site, 559
TFM's Home Page, Web site, 563
Themed sets, 161
Thermal energy, 43
Thermodynamics, 180
Thermostat, 229
Three-dimensional space, 239
Three-finger pneumatic hand, 206
Throttle, 470–471, 478–479, 482
applying, 488
maximum, 483
rotation sensor, 491
THRSH constant, 398
Thrust, 470
output, 472
Ticks, number, 244
Tic-Tac-Toe (TTT), 392, 408
board, scanning, 394
game, improvement, 396
hardware, construction, 393–394
playing, 392–396
program, writing, 394–396
robot, 392. *See also* Mindfest
software, 395
scheme, tracing, 393
Web site, 564
Tiles, 158, 383, 387. *See also* Inverted round tiles
navigation, 498
usage, 458
Tilting, 10–12
Time, 579
Time Series Analysis, Web site, 560
TinyVM Project, 105
Tissue paper, 496
T-junction, 185
Toggle(OUT_B) command, 270
Tolerance, 215, 358
Tom-tom, 413
Tones, communication, 118–119
Toronto Users Group (rtlToronto), 522
Torque, 21, 29. *See also* Output torque
application. *See* Slow speed/high torque application
definition, 20
increasing, 24
maximum, 26
measurement unit, 25
transferring, 35
Torque-limiting component, 193
Torque-limiting mechanism, 34
Touch sensors, 58–64, 161, 402. *See also* Y-axis touch sensor
activation, 255
addition. *See* Input ports
attachment. *See* Wheels
closing, 268
connection, 107, 427. *See also* Output ports
emulation, 75
input port, dedicating, 68
opening, 256
pressing, couting, 63
pressure, exertion, 542
pushing, 76
replacement, 521
technique, 408
usage, 171, 190, 254, 316, 328
Tow Truck, 529
Track links, 157
Tracks. *See* MINDSTORMS kit
friction, 140
gripping ability, 140
Traction, maximization. *See* Robotic sumo
Trains, 328–329
speed regulator, 50, 329

Transformers, 155
Transmission, 157
  changeover catch, 132, 274
  driving ring, 132, 274
  usage. *See* Robotic sumo
Transparent bricks, 438
Transparent material, 240
Transparent plates, 438
Traveled distance, 579. *See also* Actual traveled distance
Traverse axle, 59, 316
Traverse beam, usage, 7
Tread links, 35
Tree. *See* LEGO
Triads, usage. *See* 3-4-5 triad
Triangles, 14, 476. *See also* Right-angled triangle
Tricycle drive, 258. *See also* Single-motor tricycle drives
  construction, 144–145
Tricycles. *See* Sailing tricycle
Trigonometric functions, 245, 377
Trigonometry, 243, 245
  usage, 407
Tri-star wheel drive, 150
Trotting, 43–46
True-life steering vehicles, 142
Trufelli, Guido, 122, 124, 531, 542, 556
Truncations, 218
TTT. *See* Tic-Tac-Toe
Tubes, 503
Tubing, 196
Turning
  ability, 499
  angle, 451
  capabilities, 465
  motor, connection, 452
Turns
  Ackerman steering, usage. *See* Smooth turns
  control, 252
  random values, 253
Turntable, 146, 147, 489
  geartrain, 405
  rotation, degree, 408
Turtle. *See* LOGO turtle
  construction, improvements, 344–345
  creation, 339–345
  leg geometry, 334
Turtle Graphics, 442
Twelve-legged walker, 298
Two-byte messaging system, 454
Two-legged robots, 284
Two-word commands, 168

U

Ultimate Accessory Set, 50, 71, 161
Ultrasonic proximity detectors, 154
Ultrasonic sensors, 166
Ultrasonic signal, 166
Umeda, Kazuhiro, 280
Unconventional vehicles, 311
  introduction, 312
  summary, 331
Underflows, 217
Ungeared motor, 43, 45
Units, 32, 570. *See also* Vertical unit
  expressing, 4–5. *See also* Horizontal units
  ratio, 4
Universal joints, 271, 312, 320
  location, 272
Unix emulators, 104
Unix-based simulator, 555
Unix-like machine, 104, 105
Unrolled gear, 142
UP constant, 448
Up/down pen movements, 455
Upper 16t gear, 499
Upside-down rack and pinion assembly, 305
Useful robots, construction, 493
  introduction, 494
  summary, 509
User interface, 103
Users Group. *See* LEGO
U-shaped feet, 299
U-shaped legs, construction, 284
Using R/C Servos with the RCX, Web site, 558

Utrecht University, Department of
Computer Science, 568
U-Turn, 539

V

Valves, 185. *See also* Electric valve; Pneumatic valve
- movement, 194
- opening/closing, RCX usage, 187
- state, 190
- switches, 530. *See also* Motor-driven valve switches
  - opening, 409

Van Dam, Bert, 197
van Dam, Bert, 550
van Loo, Reinard, 552
Variables, 215, 243, 491. *See also* Dependent variable; Floating-point variables; Independent variable; Line variable; Status variables
- bits, addressing, 388
- determination, 280
- error, 435
- limitation. *See* Firmware
- reading, 108
- square, assigning, 395

Vehicles. *See* Multi-Degree-of-Freedom vehicles; True-life steering vehicles; Unconventional vehicles
- creation, 328–330
- rear view, 266

Velocity, 34. *See also* Angular velocity
- reduction, 26
- situations, 35

Vertical axis, 468
Vertical beams, 7–8, 84, 343
Vertical bracing, 6–10
Vertical mounts, 156
Vertical movement, 400
Vertical plane, 12
Vertical unit, 7
VHF Omni-directional Range (VOR) system, 239
Villa, Luis, 104, 548
Violent behavior, 527
Virtual animal, 442
Virtual COG, 474
Vision. *See* Artificial vision
Vision Command, 162
Visual interface, construction. *See* Chess
VisualNQC, Web site, 554
Voice Recognition unit, 154, 168
Voltage, 42–43, 517
- increase, 43

VOR. *See* VHF Omni-directional Range
V-shaped beams, 443
V-shaped skis, 321

W

Wait statement, 120
Wait(200) statement, 113
WaitBumperPress (subroutine), 109
Walker. *See* Four-legged walker; Multilegged walkers; Twelve-legged walker
- COG, shifting, 306

Walking robots, construction, 279
- introduction, 280
- summary, 310

Walking theory, 280–287
Walls
- climing, 522
- color, 520
- following, 389, 520–521
- height, 520
- material, 520
- recognition, 374–376
- robot contact, 520

Water
- network, 507
- outlet, 503
- tank, 507

Watts, 43
WAV files, conversion, 124
WAV2RCX, 556
Weather conditions, surveying, 80
Web-Ku, Web site, 566
Weighed brick side, 402
Weight constraints, 527

Weighted averages, 223–226, 578
Wheeled chassis, 499
Wheeled mobility configurations, 128
Wheeled robots, 128
Wheels, 579–580. *See also* Bumpers; Front wheels; Gear wheels; Horizontal wheel; Non-pivoting wheels; Parallel wheels; Pivoting wheel; Rear wheels; Spoke wheels; Stationary wheel; Steering-driven wheels
  angular velocity. *See* Rotating wheel
  assembly, 479
  axles, 313
  circumference, 72, 579
  locking, 132
  skidding, 134
  substitution, 517
  touch sensors, attachment, 429
Where Am I, Web site, 560
Where No Man Has Gone Before, Web site, 563
while(true) statement, 108
White-black difference, 230
Whole integers, 486
Whole numbers, 215
  usage. *See* Robotic Control eXplorer
Width, 4
Wiki, Web site. *See* Robotics
Wilcox, Doug, 561
Williamson, Ben, 560
WinCE platforms, 555
Window Walker, 562
WinVLL, Web site, 555
Wiring. *See* Motors
Workaround, 505
Working diagonal, 11
Working distances, 551
Worm gears, 23–24, 156, 173, 308
  16t, matching, 340, 343
  24t, matching, 320
  driving, 200
  gear, meeting, 323
  rotation, 201
  system, 264
  usage, 499
Writer. *See* Tape writer
Writing, 441
  head, 455
  introduction, 442
  suggestions, 464–465
  summary, 465–466

Y

YATTT. *See* Yet Another Tic-Tac-Toe
Yaw, 468
  coordination, 472
Y-axis touch sensor, 384–385
Yet Another Tic-Tac-Toe (YATTT), 395, 564
Y-motor, 385
Yoke, 471, 478
  pushing, 488

Z

Zang, Zhengrong, 563
Zhengrong Zang's RCX Controlled LEGO Train Projects, Web site, 563