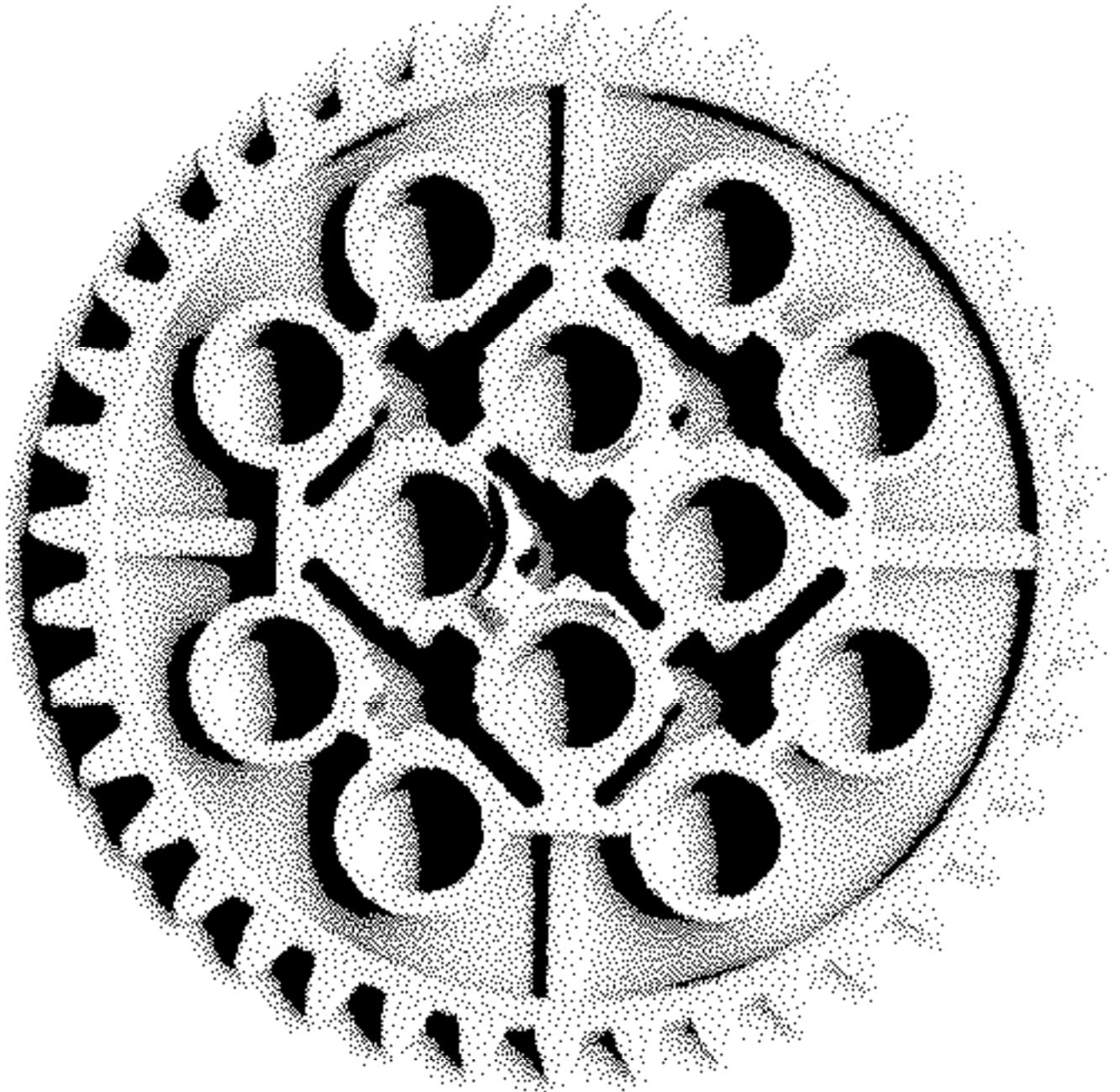


# The 6.270 Robot Builder's Guide



for the 1992 M.I.T. LEGO  
Robot Design Competition

Project Conceived and Developed by: Fred Martin, Pankaj Oberoi, and Randy Sargent

This Document Created by: Fred Martin

# The 6.270 Robot Builder's Guide

Copyright © 1992 by Fred G. Martin

Second edition, December 1992.

The author hereby grants to M.I.T. permission to reproduce and to distribute copies of this document in whole or in part.

This document may also be distributed freely in verbatim form (in whole or in part) provided that no fee is collected for its distribution (other than reasonable reproduction costs) and this copyright notice is included.

Other than verbatim copies with copyright notice intact, no part of this document may be reproduced in any form without written permission of the author. For example, the author does *not* grant the right to make derivative works based on this document without written consent.

To obtain additional copies of this document, write to:

The Epistemology and Learning Group  
The Media Laboratory  
The Massachusetts Institute of Technology  
20 Ames Street Room E15-309  
Cambridge, MA 02139

This document may also be obtained in electronic form (in PostScript (R) files) via anonymous FTP to [cherupakha.media.mit.edu](ftp://cherupakha.media.mit.edu) (Internet address 18.85.0.47).

Section 1.1, “Electronic Assembly Technique,” is based on a previous version written by Pankaj Oberoi.

Chapter 4, “LEGO Design,” was co-authored with Randy Sargent.

This document was typeset by the L<sup>A</sup>T<sub>E</sub>X text formatting system running on a DECstation 5000 computer. The `psfig` macro (written by Trevor Darrell) was used for including PostScript (R) figures in L<sup>A</sup>T<sub>E</sub>X documents. `dvips` version 5.478 was used to create final PostScript output.

Many diagrams were created with the `idraw` X-windows-based drawing program, distributed by the MIT Athena computer network.

Electrical circuit schematics were drawn using the DesignWorks (R) software package, distributed by Douglas Electronics, Inc., running on a Macintosh IIfx computer.

Printed circuit board artwork was created using the Professional Layout (R) software package, distributed by Douglas Electronics, Inc., running on a Macintosh IIfx computer.

Cover art was scanned by Wanda M. Gleason on an HP Scanjet IIc.

# Foreword

These notes were first written for the students of an experimental class at the Massachusetts Institute of Technology, the *LEGO Robot Design Competition*. They were compiled into the current book form and distributed to the students of the 1992 class. This second version of the *Robot Builder's Guide* contains some minor corrections and reorganization from the first edition, intended to make the work more suitable for perusal by both robotic enthusiasts and teachers interested in robot design as an educational activity.

Appendix A of this edition explains the organization of the month-long course and also contains the 1992 contest specification. If the reader is interested in either of these two topics, it may be worthwhile to scan through this section first.

Chapter One presents assembly directions for all of the hardware in our robot building kit. Interspersed throughout the assembly information are brief introductions to each of the components. It's probably the case that most readers won't have the physical hardware in front of them when they read this section. Nevertheless it seemed the most appropriate chapter to introduce the materials. The reader may wish to skip over Section 1.1, which just discusses standard electronic components. Of course, if the reader has obtained a kit of materials, reading all of this chapter to learn how to assemble the kit seems like a good place to start!

The work documented in this manual is the result of contributions from many persons and in particular from a collaboration with two other individuals: Randy Sargent and Pankaj Oberoi. Most of the hardware described herein was co-designed with Randy, and he is the main author of the "Interactive C" environment described in Chapter 7. Pankaj has contributed to the project in many ways, including having played a steadfast organizational and motivational role. The enthusiastic participation of past students has also contributed to the development of both the technology and our ideas about how to best use it in an educational setting.

During the past year, the M.I.T. Department of Electrical Engineering and Computer Science and the M.I.T. Media Laboratory, who sponsored the development of the class technology, agreed to unrestricted and free distribution of the robotics technology described herein, including the printed circuit board artwork and software programming environment. While the material hasn't been placed in the public

domain (it's still copyrighted), this means that any individual or organization can use the works for whatever purposes they desire. Our hope is that people will use this work in the spirit of GNU software, which is distributed freely and is supported largely by a community of interested users.

The 1992 class used version 2.1 of the robot controller board; this manual is based on a newer version 2.11, which fixes several minor wiring bugs that were discovered in the 2.1 design. The two designs are identical in terms of functionality; version 2.11 is a replacement for the older design. Information on obtaining plans and/or kits for the board set may be obtained from `cherupakha.media.mit.edu` (Internet address 18.85.0.47), an anonymous FTP file server maintained at the M.I.T. Media Laboratory that is used to distribute this work, among other things. This information may also be obtained by writing to the author at the address provided on the copyright page.

The LEGO Robot Design course is an on-going educational experiment. The 1993 class is being organized mainly by undergraduates who were students of prior classes. They are taking on the tasks of upgrading the hardware designs and re-writing much of the course notes to suit their own sense of what's best for the next generation of students. To me this represents an important step in the evolution of the class. In attracting such dedicated leaders, the class is keeping alive the creative excitement that marked the years in which I was involved.

Fred Martin  
December 1, 1992

# Preface

Michael B. Parker, an undergraduate in MIT’s Electrical Engineering and Computer Science (Course Six) department, had just taken MIT’s famous “Introduction to Design” class (course number 2.70) in the Mechanical Engineering department. In 2.70, undergraduates use scrap parts—metal, plastic, and wood—to build machines that go on to compete in a head-to-head contest at the end of the course.

Mike liked the course so much that he was jealous: why should there be a course like this for Mechanical Engineering students, but not for the students in his department?

So in 1987, Mike organized the first 6.270 contest, as “Course Six’s answer” to the 2.70 course. The contest was a programming competition in which students wrote programs to control computer-simulated robots. In the first two years of the contest, the goal was to design a simulated robot that tried to find and destroy other robots.

A couple years later, Mike saw a project at MIT’s Media Laboratory (where I was working) called “LEGO/Logo,” in which children build robots and other mechanized devices out of LEGO bricks, motors, and electronic sensors, and then write programs to control them using a special version of the Logo programming language. Mike wanted to provide the 6.270 students with similar technology, so they could build *real* robots, not just the computer-simulated robots that been done in the past.

Along with me, Mike recruited Randy Sargent, a fellow student of his, to be the technical consultants to the upcoming 6.270 contest (which was starting in just a few weeks). It sounded like a fun way to spend IAP.<sup>1</sup>

Randy and I spent most of our holiday break designing an interface board that connected to a PC or Macintosh computer, controlling motors and providing input from a few simple sensors. The budget was tight and time was short as we scurried about the local Radio Shack stores, buying electronic parts for the twenty teams of students who had preregistered for the course.

Everything went wrong that month: the LEGO parts showed up late; we were unable to obtain approval to use campus Athena workstations; mix-ups in room scheduling forced us, a group of students wielding soldering irons, to wander about

---

<sup>1</sup>MIT’s “Independent Activities Period” is a one-month break between the fall and spring semesters.

campus looking for available classrooms. It was only the students' excitement that carried us through the month and into a competition at the end of IAP. The contest lasted about four hours—it was a long, drawn-out affair—but the students enjoyed it.

In October of the next year, Pankaj (“P.K.”) Oberoi, who had been a student in the class, called a meeting of interested past participants. P.K. thought the class had great potential and wanted to help organize it for the upcoming year.

P.K. had already worked on key administrative aspects, like recruiting corporate sponsors. Microsoft had donated some money to the course previously and was willing to up the ante for another year. P.K. also got the support of Motorola, which agreed to donate valuable semiconductor parts.

So Randy and I again teamed up to develop technology for the course. We laid out artwork for a custom printed circuit board that used a new Motorola microprocessor with lots of control features. P.K. and I wrote handouts for the students, telling them how to build sensors, a battery charger, and other robotics components.

Eighty students took the course that year (organized into thirty teams). Even before the contest that year, it was evident that the course was a success. As students carried their robot kits around campus, interest and excitement spread. The contest itself was a hit.

By the end of that year, I was interested in the course not only from a technical perspective, but from a special educational perspective. Students at MIT were choosing to pull all-nighters building robots rather than taking ski trips. In doing so, they were learning about engineering design and robotic technologies from first-hand, experiential involvement in a project-based course. The course seemed to fill a gap in the students' education, providing them with a complement to the theoretical orientation of many MIT classes.

The course's approach to learning also fits closely with the educational ideas of the research group I am part of at the Media Laboratory. Seymour Papert, Director of the Epistemology and Learning Research group, has developed an educational philosophy called “constructionism.” According to constructionism, learning and the acquisition of knowledge are active processes engaged in by the learner; i.e., knowledge is constructed by the learner. Also, this process can be greatly facilitated when the learner is building something real in the world, in addition to building knowledge inside his or her own head.

By providing the tools and materials for students to work with complex electronic, mechanical, and software ideas, 6.270 gives the students a place to explore and learn about key ideas in technology, engineering, and design. Teamwork, learning-by-doing, and learning from one's peers is primary. 6.270 provides a constructionist-style learning experience for MIT students.

The 1990 class was a big success but we wanted to make the class even better. The

class was hampered by a controller board that had to be programmed in assembly language and only had a small amount of memory. Afterward, we began work on a robotic technology that would be more powerful and more useful to 6.270 students, allowing them to get even deeper into robot design and other technological issues.

By the time of the 1991 class, we had developed a robot-building kit with the high degree the power and flexibility we had wanted. Students were able to develop software for their robot using the C programming language. The new embedded controller board we developed had numerous features, including a small display screen that could be used to print debugging messages. Students were able to use powerful Unix workstations (the MIT Athena network), located all across campus, to develop programs for their robots.

We put a lot of thinking into how to organize the class to maximize the students' learning potential. We scheduled the class activities so that students would have all the pieces of a robot functioning as early as possible into the course. At this point, students would be able to take charge of their projects, and enjoy themselves as they saw their robots become more and more functional.

This document, the course notes for the 1992 6.270 class, is the result of several years of work. They have evolved from a couple of pages of wiring charts in the first year, to a collection of diagrams and other assembly notes handed out in a piecemeal fashion, into this document.

I was motivated to transform the notes based on my observations from last year's class: it seemed that many of the students wanted more in-depth discussion of the class's technology and more theory than lectures alone could provide. I hope that these course notes satisfy this intellectual curiosity.

Fred Martin  
January 3, 1992



# Sponsors

Many companies both large and small have contributed generously to the 6.270 project. We are grateful to both the corporations who have supported us, and for the encouragement given to us by many individuals who have worked for them.

MICROSOFT CORP.

MOTOROLA SEMICONDUCTOR INC.

POLAROID INC.

LEGO SYSTEMS INC.

METHODE INC.

GATES ENERGY PRODUCTS INC.

3M INC.

ABRAMS-GENTILE ENTERTAINMENT INC.



# Introduction to this Guide

These course notes contain a lot of information that should be valuable in your robot-building endeavor. Here's a brief synopsis of the organization of these course notes:

- Chapter 1 is the assembly manual for nearly all of the components used to construct a 6.270 robot—including the printed circuit boards, electronic sensors, motors, and battery packs.
- Chapter 2 introduces robot motor technology.
- Chapter 3 discusses battery technology and battery charger operation.
- Chapter 4 discusses secret construction methods using the LEGO Technics (R) building materials.
- Chapter 5 delves into robotic sensors, explaining the principles of operation and applications of various sensors in the 6.270 kit.
- Chapter 6 investigates how to program a mobile robot to face up to the uncertainties and challenges of practical operation.
- Chapter 7 is a reference manual for the C language software that has been developed for the 6.270 contest.

Four appendix sections provide additional material:

- Appendix A is the student's introduction to the 1992 class, covering administrative information, the month's schedule, and the contest rules.
- Appendix B explains the workings of the 6.270 hardware, including the microprocessor board, the expansion board, and the infrared circuitry. This section is written with the assumption of some prior background in electronics.
- Appendix C provides reproductions of the printed circuit board layouts, handy for reference when debugging.
- Appendix D contains an introduction to ordering electronic parts from both the retail and surplus markets, and a listing of more than twenty top suppliers.



# Contents

<b>1</b>	<b>Assembly Manual</b>	<b>1</b>
1.1	Electronic Assembly Technique . . . . .	1
1.1.1	Soldering Technique . . . . .	2
1.1.2	Desoldering Technique . . . . .	4
1.1.3	Component Types and Polarity . . . . .	4
1.1.4	Component Mounting . . . . .	9
1.1.5	Component Value Markings . . . . .	10
1.2	The Microprocessor Board . . . . .	11
1.2.1	Assembling the Microprocessor Board . . . . .	12
1.2.2	Testing the Microprocessor Board . . . . .	20
1.2.3	After Board Checkout . . . . .	22
1.3	The Expansion Board . . . . .	22
1.3.1	Assembling the Expansion Board . . . . .	22
1.3.2	Testing the Expansion Board . . . . .	26
1.3.3	After Board Testing . . . . .	28
1.4	The LCD Display . . . . .	28
1.5	The Battery System . . . . .	28
1.5.1	The Battery Charger . . . . .	29
1.5.2	Battery Pack Construction . . . . .	31
1.6	The Infrared Transmitter . . . . .	33
1.6.1	Assembly Instructions . . . . .	34
1.7	The Motor Switch Board . . . . .	36
1.7.1	Assembly Instructions . . . . .	36
1.8	Cable and Connector Wiring . . . . .	39
1.9	Sensor Wiring . . . . .	39
1.9.1	Touch Sensors . . . . .	43
1.9.2	Photocell Light Sensor . . . . .	44
1.9.3	Infrared Reflectance Sensor . . . . .	45
1.9.4	Single Bend Sensor . . . . .	46
1.9.5	Dual Bend Sensor . . . . .	47
1.9.6	Sharp Infrared Sensor . . . . .	48

1.9.7	Potentiometer . . . . .	48
1.9.8	Slotted Optical Switch . . . . .	49
1.9.9	Hall Effect Sensor . . . . .	50
1.10	Motor Wiring . . . . .	51
1.10.1	The Polaroid Motor . . . . .	51
1.10.2	Servo Motor . . . . .	54
1.11	Computer Cable Wiring . . . . .	56
<b>2</b>	<b>Motors</b>	<b>59</b>
2.1	DC Motors . . . . .	59
2.1.1	Ratings and Specifications . . . . .	59
2.1.2	Measuring Motor Torque . . . . .	61
2.1.3	Speed, Torque, and Gear Reduction . . . . .	62
2.1.4	Pulse Width Modulation . . . . .	63
2.2	Stepper Motors . . . . .	64
2.3	Servo Motors . . . . .	64
2.3.1	Control . . . . .	64
2.3.2	Application . . . . .	65
<b>3</b>	<b>Batteries</b>	<b>67</b>
3.1	Cell Characteristics . . . . .	67
3.1.1	Voltage . . . . .	67
3.1.2	Capacity . . . . .	67
3.1.3	Power Density . . . . .	68
3.1.4	Discharge Curve . . . . .	68
3.1.5	Internal Resistance . . . . .	68
3.1.6	Rechargeability . . . . .	69
3.1.7	Cost . . . . .	70
3.2	Battery Packs . . . . .	70
3.3	6.270 Battery Charger . . . . .	71
<b>4</b>	<b>LEGO Design</b>	<b>73</b>
4.1	Fundamental LEGO Lengths . . . . .	73
4.2	LEGO Gearing . . . . .	76
4.2.1	Gear Reduction . . . . .	78
4.2.2	Chain Drives . . . . .	79
4.2.3	Testing a Geartrain . . . . .	79
4.2.4	Low-Force Geartrains . . . . .	79

<b>5</b>	<b>Sensor Design</b>	<b>81</b>
5.1	Sensors as Transducers . . . . .	81
5.1.1	Analog and Digital Sensors . . . . .	81
5.2	Sensor Inputs on the 6.270 Board . . . . .	82
5.2.1	Analog Inputs . . . . .	82
5.2.2	Digital Inputs . . . . .	82
5.2.3	Connector Plug Standard . . . . .	83
5.2.4	Sensor Input Wiring . . . . .	84
5.2.5	The Voltage Divider Circuit . . . . .	84
5.3	Tactile Sensors . . . . .	85
5.3.1	Touch Switch . . . . .	86
5.3.2	Bend Sensor . . . . .	88
5.3.3	Potentiometer . . . . .	89
5.4	Light Sensors . . . . .	89
5.4.1	Photocell . . . . .	91
5.4.2	Infrared Reflectance Sensor . . . . .	91
5.4.3	Infrared Slotted Optical Switch . . . . .	93
5.4.4	Modulated Infrared Light Detector . . . . .	94
5.5	Other Sensors . . . . .	94
5.5.1	Magnetism Sensor . . . . .	94
5.5.2	Motor Current Sensor . . . . .	95
<b>6</b>	<b>Robot Control</b>	<b>97</b>
6.1	Basic Control Methods . . . . .	98
6.1.1	Feedback Control . . . . .	98
6.1.2	Open-Loop Control . . . . .	100
6.1.3	Feed-Forward Control . . . . .	101
6.1.4	Summary . . . . .	102
6.2	Sensor Calibration . . . . .	102
6.2.1	Manual Sensor Calibration . . . . .	102
6.2.2	Dealing with Changing Environmental Conditions . . . . .	103
6.2.3	Using Persistent Global Variables . . . . .	104
6.3	Robot Control . . . . .	104
6.3.1	Sensor Unreliability . . . . .	105
6.3.2	Task-Oriented Control . . . . .	106
6.3.3	Coordination of Tasks . . . . .	110
<b>7</b>	<b>IC Manual</b>	<b>115</b>
7.1	Getting Started . . . . .	116
7.2	Using IC . . . . .	117
7.2.1	IC Commands . . . . .	117

7.2.2	Line Editing . . . . .	118
7.2.3	The <code>main()</code> Function . . . . .	119
7.3	A Quick C Tutorial . . . . .	119
7.4	Data Types, Operations, and Expressions . . . . .	121
7.4.1	Variable Names . . . . .	121
7.4.2	Data Types . . . . .	121
7.4.3	Local and Global Variables . . . . .	122
7.4.4	Constants . . . . .	123
7.4.5	Operators . . . . .	124
7.4.6	Assignment Operators and Expressions . . . . .	125
7.4.7	Increment and Decrement Operators . . . . .	126
7.4.8	Precedence and Order of Evaluation . . . . .	126
7.5	Control Flow . . . . .	127
7.5.1	Statements and Blocks . . . . .	127
7.5.2	If-Else . . . . .	127
7.5.3	While . . . . .	127
7.5.4	For . . . . .	128
7.5.5	Break . . . . .	128
7.6	LCD Screen Printing . . . . .	128
7.6.1	Printing Examples . . . . .	129
7.6.2	Formatting Command Summary . . . . .	129
7.6.3	Special Notes . . . . .	130
7.7	Arrays and Pointers . . . . .	130
7.7.1	Declaring and Initializing Arrays . . . . .	130
7.7.2	Passing Arrays as Arguments . . . . .	131
7.7.3	Declaring Pointer Variables . . . . .	131
7.7.4	Passing Pointers as Arguments . . . . .	132
7.8	The IC Library File . . . . .	133
7.8.1	Output Control . . . . .	133
7.8.2	Sensor Input . . . . .	135
7.8.3	Time Commands . . . . .	140
7.8.4	Tone Functions . . . . .	141
7.9	Multi-Tasking . . . . .	141
7.9.1	Overview . . . . .	141
7.9.2	Creating New Processes . . . . .	142
7.9.3	Destroying Processes . . . . .	143
7.9.4	Process Management Commands . . . . .	144
7.9.5	Process Management Library Functions . . . . .	144
7.10	Floating Point Functions . . . . .	144
7.11	Memory Access Functions . . . . .	145
7.12	Error Handling . . . . .	146

7.12.1	Compile-Time Errors . . . . .	146
7.12.2	Run-Time Errors . . . . .	146
7.13	Binary Programs . . . . .	147
7.13.1	The Binary Source File . . . . .	147
7.13.2	Interrupt-Driven Binary Programs . . . . .	150
7.13.3	The Binary Object File . . . . .	154
7.13.4	Loading an <code>icb</code> File . . . . .	154
7.13.5	Passing Array Pointers to a Binary Program . . . . .	154
7.14	IC File Formats and Management . . . . .	155
7.14.1	C Programs . . . . .	155
7.14.2	List Files . . . . .	155
7.14.3	File and Function Management . . . . .	156
7.15	Configuring IC . . . . .	156
<b>A</b>	<b>Introduction to 6.270</b>	<b>157</b>
A.1	Registration Policy . . . . .	158
A.2	Kit Fee and Toolkit Fee . . . . .	158
A.3	Team Organization . . . . .	159
A.4	6.270 Alumni . . . . .	160
A.5	Credit Guidelines . . . . .	160
A.6	Schedule . . . . .	162
A.6.1	Important Dates . . . . .	163
A.6.2	Detailed Schedule of the Month's Activities. . . . .	165
A.7	Robo-Cup Contest Rules . . . . .	169
A.7.1	Object . . . . .	169
A.7.2	Balls . . . . .	169
A.7.3	The Goal . . . . .	172
A.7.4	Period of Play . . . . .	172
A.7.5	Control . . . . .	173
A.7.6	Infrared Beacon . . . . .	173
A.7.7	The Contest Playing Table . . . . .	174
A.7.8	Structure . . . . .	175
A.7.9	The \$10 Electronics Rule . . . . .	176
A.7.10	Scoring . . . . .	177
A.7.11	Organizers . . . . .	177
A.8	Parts List . . . . .	177
<b>B</b>	<b>6.270 Hardware</b>	<b>183</b>
B.1	The Microprocessor and Memory . . . . .	183
B.1.1	Multiplexing Data and Address Signals . . . . .	184
B.2	Memory Mapping . . . . .	185

B.2.1	Memory-Mapping the RAM . . . . .	187
B.2.2	Memory-Mapping with the 74HC138 Chip . . . . .	188
B.2.3	System Memory Map . . . . .	191
B.2.4	Digital Inputs . . . . .	191
B.2.5	Digital Outputs . . . . .	191
B.2.6	6811 and Memory Schematic . . . . .	193
B.3	The Motor Drivers . . . . .	193
B.3.1	The H-Bridge Circuit . . . . .	195
B.3.2	The H-Bridge with Enable Circuitry . . . . .	195
B.3.3	The SGS-Thomson Motor Driver Chip . . . . .	196
B.3.4	Power Considerations . . . . .	199
B.3.5	Expansion Board Motor and LED Circuitry . . . . .	201
B.4	Analog Inputs . . . . .	201
B.4.1	Motor Current Monitoring Circuit . . . . .	202
B.4.2	Analog Input Multiplexing on the Expansion Board . . . . .	203
B.5	The Serial Line Circuit . . . . .	204
B.5.1	Serial Output . . . . .	204
B.5.2	Serial Input . . . . .	206
B.6	Battery-Backing the Static RAM . . . . .	207
B.6.1	Powering the Memory Chip . . . . .	207
B.6.2	The Power-Off Interrupt . . . . .	207
B.6.3	The Power-Up Delays . . . . .	208
B.7	The Infrared Transmission Circuit . . . . .	209
B.7.1	The IR Beacon . . . . .	211
B.8	The LCD Display . . . . .	212
B.9	The Low-Battery Indicator . . . . .	214
<b>C</b>	<b>Printed Circuit Layouts</b> . . . . .	<b>215</b>
C.1	Microprocessor Board . . . . .	216
C.2	Expansion Board . . . . .	217
C.3	Battery Charger Board . . . . .	218
C.4	Motor Switch Board . . . . .	218
C.5	Infrared Beacon Board . . . . .	219
<b>D</b>	<b>Electronic Suppliers</b> . . . . .	<b>221</b>
D.1	Introduction . . . . .	221
D.1.1	Retail Suppliers . . . . .	221
D.1.2	Surplus Suppliers . . . . .	222
D.1.3	Ordering Catalogs . . . . .	222
D.2	The Surplus Companies . . . . .	223
D.2.1	Surplus Company Listing . . . . .	223

*CONTENTS*

xix

D.3 The Retail Companies . . . . . 228



# List of Figures

1.1	Proper Soldering Technique . . . . .	2
1.2	Improper Soldering Technique . . . . .	3
1.3	Resistor Pack Internal Wiring . . . . .	5
1.4	Typical Diode Package . . . . .	6
1.5	Identifying LED Leads . . . . .	6
1.6	Top View of 14-pin DIP . . . . .	7
1.7	Top View of 52-pin PLCC . . . . .	7
1.8	Summary of Polarization Effects . . . . .	9
1.9	Flat Component Mounting . . . . .	9
1.10	Upright Component Mounting . . . . .	10
1.11	Resistor Color Code Table . . . . .	11
1.12	6.270 Microprocessor Board Component Placement . . . . .	13
1.13	6.270 Microprocessor Board Header Placement . . . . .	16
1.14	Motor Chip Stacking Technique . . . . .	19
1.15	Expansion Board Component Placement . . . . .	23
1.16	Expansion Board Female Header Mounting . . . . .	25
1.17	Mounting Method for Male Header Pins . . . . .	26
1.18	Expansion Board Male Header Pin Placement . . . . .	27
1.19	LCD Connector Mounting . . . . .	28
1.20	Battery Charger Component Placement . . . . .	30
1.21	Two Battery Pack Configurations . . . . .	32
1.22	Battery Plug and Cable Wiring Diagram . . . . .	32
1.23	Battery Pack Wiring Diagram . . . . .	34
1.24	Infrared Transmitter Component Placement . . . . .	35
1.25	Motor Switch Board Component Placement . . . . .	37
1.26	Standard Connector Plug Configurations . . . . .	38
1.27	Step One of Connector Wiring . . . . .	40
1.28	Step Two of Connector Wiring . . . . .	40
1.29	Step Three of Connector Wiring . . . . .	41
1.30	Step Four of Connector Wiring . . . . .	41
1.31	Motor Housing with Tubing . . . . .	52

1.32	LEGO Jig for Mounting Polaroid Motor . . . . .	53
1.33	Servo Motor and Integral Connector Plug . . . . .	55
1.34	Servo Motor Cable Wiring Diagram . . . . .	55
1.35	VAXstation 2000 Computer Cable Wiring Diagram . . . . .	57
2.1	Polaroid Motor Specifications . . . . .	61
2.2	Experiment to Measure Motor Torque . . . . .	61
2.3	Example of Several Pulse Width Modulation Waveforms . . . . .	63
3.1	Table of Cell Characteristics . . . . .	70
3.2	Battery Charger Schematic Diagram . . . . .	71
4.1	The Unit LEGO Brick . . . . .	74
4.2	Perfect 2-Unit Vertical LEGO Spacing . . . . .	74
4.3	Clamping Two Beams at Perfect Vertical Spacing . . . . .	75
4.4	Meshing of an 8-Tooth Gear and a 24-Tooth Gear . . . . .	77
4.5	Diagonal Meshing of an 8-Tooth Gear and a 16-Tooth Gear . . . . .	77
4.6	Diagonal Meshing of a 16-Tooth Gear and a 24-Tooth Gear . . . . .	77
4.7	LEGO Gearbox Example . . . . .	80
5.1	Wiring a Generic Sensor . . . . .	83
5.2	Sensor Input Wiring, One Sensor . . . . .	84
5.3	Voltage Divider Schematic . . . . .	85
5.4	A Typical Microswitch . . . . .	86
5.5	Robotic Platform Employing Bumper Coupled to Touch Sensor . . . . .	87
5.6	Touch Switch Circuit . . . . .	87
5.7	Potentiometer Circuit . . . . .	90
5.8	Two Optical Shaft Encoder Wheels . . . . .	92
5.9	Phototransistor and Infrared Emitter Circuit . . . . .	93
6.1	Driving along a Wall Edge . . . . .	98
6.2	Using a Bend Sensor to Measure Distance to a Wall . . . . .	99
6.3	Wall-Following Function . . . . .	99
6.4	Negotiating a Corner . . . . .	100
6.5	Negotiating a Corner with Touch Sensor Feedback . . . . .	101
6.6	Wall Following Function with Exit Condition . . . . .	107
6.7	Wall Following Function with Exit and Timeout Conditions . . . . .	109
6.8	Task Analysis of Wall Follow Function with Sensing and Timeout Capability. . . . .	110
6.9	Wall Following Function with Exit, Timeout, and Feedback Monitoring Conditions, Page 1 . . . . .	111

6.10	Wall Following Function with Exit, Timeout, and Feedback Monitoring Conditions, Page 2 . . . . .	112
7.1	IC Command-Line Keystroke Mappings . . . . .	119
7.2	Sample IC Binary Source File: <code>testicb.asm</code> . . . . .	148
7.3	Interrupt Structure Before User Program Installation . . . . .	150
7.4	Interrupt Structure After User Program Installation . . . . .	151
7.5	<code>sysibeep.asm</code> : Binary Program that Installs into System Interrupt . . . . .	153
7.6	Sample IC Binary Object File: <code>testicb.icb</code> . . . . .	154
A.1	Pictorial Schedule of the Month's Activities . . . . .	164
A.2	Robo-Cup '92 Contest Playing Field Specification . . . . .	170
A.3	Robo-Cup '92 Contest Goal Specification . . . . .	171
A.4	6.270 Parts Listing, Page One of Four . . . . .	179
A.5	6.270 Parts Listing, Page Two of Four . . . . .	180
A.6	6.270 Parts Listing, Page Three of Four . . . . .	181
A.7	6.270 Parts Listing, Page Four of Four . . . . .	182
B.1	Block Diagram of Microprocessor and Memory . . . . .	184
B.2	Block Diagram of Microprocessor and Memory with Latch . . . . .	185
B.3	6.270 System Block Diagram . . . . .	186
B.4	Enabling the Memory . . . . .	187
B.5	The 'HC138 Address Decoder . . . . .	189
B.6	Wiring the 'HC138 Address Decoder . . . . .	189
B.7	6811 System Memory Map . . . . .	192
B.8	Digital Input Circuit . . . . .	193
B.9	6811, Memory, Address Decoding and Miscellaneous Circuitry . . . . .	194
B.10	The H-Bridge Circuit . . . . .	195
B.11	The H-Bridge with Left-to-Right Current Flow . . . . .	196
B.12	The H-Bridge with Enable Circuitry . . . . .	197
B.13	The SGS-Thomson L293 Motor Driver IC . . . . .	198
B.14	Motor Driver Circuit . . . . .	200
B.15	Power Filtering and Switching Circuit . . . . .	201
B.16	Expansion Board Motor and LED Circuitry . . . . .	202
B.17	Expansion Board Analog Input Circuitry . . . . .	203
B.18	Host and Board Communications over 3-Wire Serial Link . . . . .	204
B.19	Serial Line Circuit . . . . .	205
B.20	Square Wave Consisting of Bursts of 40 Khz Signals . . . . .	209
B.21	Sharp IR Sensor Decoding IR-Encoded Square Wave . . . . .	210
B.22	Block Diagram of Infrared Circuitry . . . . .	210
B.23	Infrared Transmission Circuit . . . . .	211
B.24	Infrared Beacon Circuit . . . . .	212

B.25	Low Battery Indicator Circuit . . . . .	214
C.1	Microprocessor Board, Component Side . . . . .	216
C.2	Microprocessor Board, Solder Side . . . . .	216
C.3	Expansion Board, Component Side . . . . .	217
C.4	Expansion Board, Solder Side . . . . .	217
C.5	Battery Charger Board, Component and Solder Sides . . . . .	218
C.6	Motor Switch Board, Component and Solder Sides . . . . .	218
C.7	Infrared Beacon Board, Component and Solder Sides . . . . .	219

# Chapter 1

## Assembly Manual

This chapter presents an introduction to electronic assembly followed by step-by-step instructions to assembling the 6.270 hardware. The instructions assume no prior background in electronics.

Instructions are provided for the following boards and devices:

- Microprocessor Board
- Expansion Board
- Battery Charger Board
- Battery Packs
- Infrared Transmitter Board
- Motor Switching Board
- Sensor Assemblies
- Motor Assemblies
- VAX2000 Cable

### 1.1 Electronic Assembly Technique

If there are places in life where “neatness counts,” electronic assembly is one of them. A neatly-built and carefully soldered board will perform well for years; a sloppily- and hastily-assembled board will cause on-going problems and failures on inopportune occasions.

This section will cover the basics of electronic assembly: proper soldering technique, component mounting technique, and component polarities.

By following the instructions and guidelines presented here, you will make your life more enjoyable when debugging time rolls around.

### 1.1.1 Soldering Technique

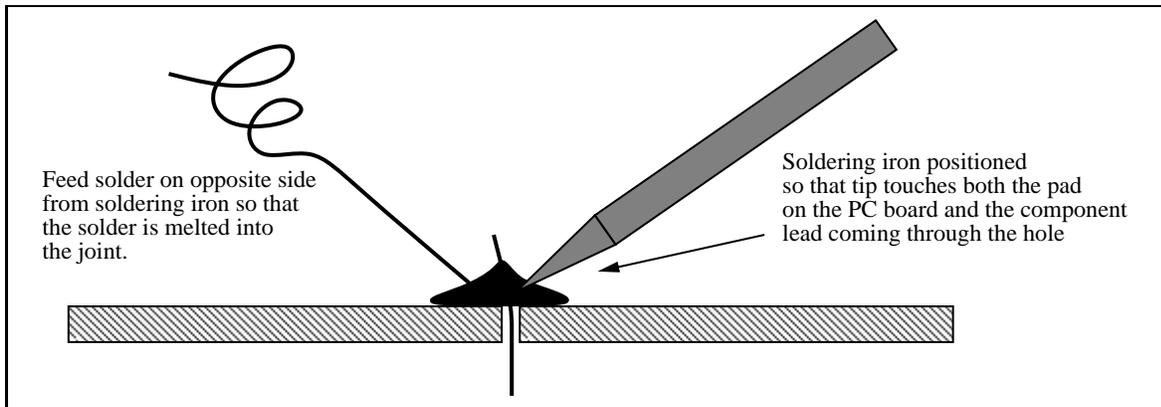


Figure 1.1: Proper Soldering Technique

Figure 1.1 shows proper soldering technique. The diagram shows the tip of the soldering iron being inserted into the joint such that it touches both the lead being soldered *and* the surface of the PC board.

Then, solder is applied into the joint, *not* to the iron directly. This way, the solder is melted *by the joint*, and both metal surfaces of the joint (the lead and the PC pad) are heated to the necessary temperature to bond chemically with the solder.

Figure 1.2 shows the typical result of a bad solder joint. This figure shows what happens if the solder is “painted” onto the joint after being applied to the iron directly. The solder has “balled up,” refusing to bond with the pad (which did not receive enough heat from the iron).

With this technique in mind, please read the following list of pointers about electronic assembly. All of these items are important and will help develop good skills in assembly:

1. Keep the soldering iron tips away from everything except the point to be soldered. The iron is *hot* and can easily damage parts, cause burns, or even start a fire. Keep the soldering iron in its holder when it is not being held.
2. Make sure that there is a damp sponge available used for cleaning off and tinning the tip. Soldering is basically a chemical process and even a small amount of contaminants can prevent a good joint from being made.

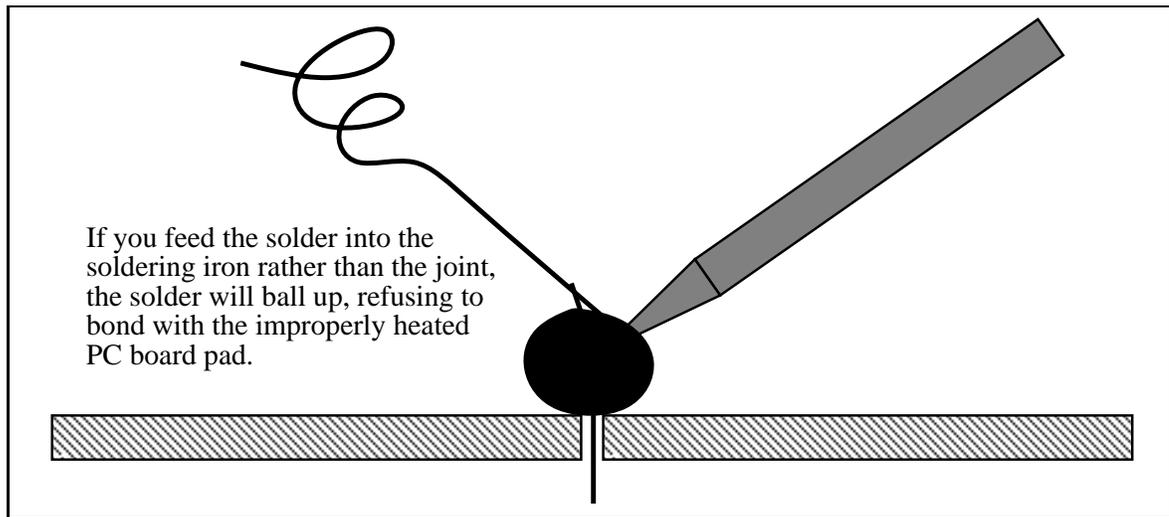


Figure 1.2: Improper Soldering Technique

3. Always make sure that the tip is *tinned* when the iron is on. Tinning protects the tip and improves heat transfer.

To tin the iron, clean the tip and wipe it on a damp sponge and then immediately melt some fresh solder onto the tip. The tip should be shiny and coated with solder.

If the iron has been idle for a while, always clean and then re-tin the tip before continuing.

4. The tips of the irons are nickel-plated, so do not file them or the protective plating on the tips will be removed.
5. A *cold solder joint* is a joint where an air bubble or other impurity has entered the joint during cooling. Cold solder joints can be identified by their dull and mottled finish. The solder does not flow and wrap around the terminal like it should.

Cold joints are brittle and make poor electrical connection. To fix such a joint, apply the tip at the joint until the solder re-melts and flows into the terminal. If a cold solder joint reappears, remove solder with desoldering pump, and re-solder the joint.

6. Do not hold the iron against the joint for an extended period of time (more than 10 seconds), since many electronic components or the printed circuit board itself can be damaged by prolonged, excessive heat. Too much heat can cause the traces on the printed circuit board to burn off.

Some components that are particularly sensitive to heat damage are: diodes, ICs, and transistors.

### 1.1.2 Desoldering Technique

It takes about ten times as long to desolder a component than it did to solder it in the first place. This is a good reason to be careful and take one's time when assembling boards; however, errors will inevitably occur, and it's important to know how to fix them.

The primary reasons for performing desoldering are removing an incorrectly-placed component, removing a burnt-out component, and removing solder from a cold solder joint to try again with fresh solder.

Two main methods of desoldering are most common: desoldering pumps and desoldering wick. The 6.270 toolkit includes a desoldering pump as standard equipment.

To use a desoldering pump, first load the pump by depressing the plunger until it latches. Grasp the pump in one hand and the soldering iron the other, and apply heat to the bad joint. When the solder melts, quickly remove the soldering iron and bring in the pump in one continuous motion. Trigger the pump to suck up the solder while it is still molten.

Adding additional solder to a troublesome joint can be helpful in removing the last traces of solder. This works because the additional solder helps the heat to flow fully into the joint. The additional solder should be applied and de-soldered as quickly as possible. Don't wait for the solder to cool off before attempting to suck it away.

The desoldering pump tip is made of Teflon. While teflon is heat-resistant, it is not invincible, so not jam the teflon tip directly into the soldering iron. Solder will not stick to Teflon, so the desoldering operation should suck the solder into the body of the pump.

### 1.1.3 Component Types and Polarity

There will be a variety of electronic components in use when assembling the boards. This section provides a brief introduction to these components with the goal of teaching you how to properly identify and install these parts when building the boards.

#### Component Polarity

*Polarity* refers to the concept that many electronic components are not symmetric electrically. A polarized device has a right way and a wrong way to be mounted. Polarized components that are mounted backwards will not work, and in some cases will be damaged or may damage other parts of the circuit.

The following components are always polarized:

- diodes (LEDs, regular diodes, other types)
- transistors
- integrated circuits

Capacitors are an interesting case, because some are polarized while others are not. Fortunately, there is a rule: large capacitors (values 1  $\mu\text{F}$  and greater) are generally polarized, while smaller ones are not.

Resistors are a good example of a non-polarized component: they don't care which direction electricity flows through them. However, in the 6.270 board, there are *resistor packages*, and these have non-symmetric internal wiring configurations, making them polarized from a mounting point of view.

Incandescent lamps are another non-polarized component.

The following paragraphs discuss the aforementioned components individually, explaining standardized component markings for identifying a component's polarity.

**Resistors** Resistors are small cylindrical devices with color-coded bands indicating their value (how to read color-coding is explained in a subsequent section).

Most of the resistors in the 6.270 kit are rated for  $\frac{1}{8}$  watts, which is a very low power rating. Hence they are quite tiny devices.

A few resistors are much larger. A 2 watt resistor is a large cylindrical device, while a 5 watt resistor has a large, rectangular package.

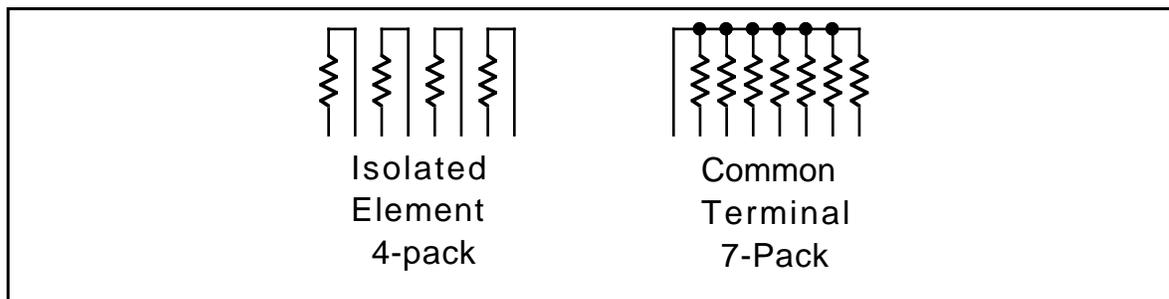


Figure 1.3: Resistor Pack Internal Wiring

**Resistor Packs** Resistor packs are flat, rectangular packages with anywhere from six to ten leads. There are two basic types of resistor pack:

- **Isolated Element.** Discrete resistors; usually three, four, or five per package.
- **Common Terminal.** Resistors with one pin tied together and the other pin free. Any number from three to nine resistors per package.

Figure 1.3 illustrates the internal wiring of an 8-pin resistor pack of each style.



Figure 1.4: Typical Diode Package

**Diodes** Diodes have two leads, called the *anode* and *cathode*. When the anode is connected to positive voltage with respect to the cathode, current can flow through the diode. If polarity is reversed, no current flows through the diode.

A diode package usually provides a marking that is closer to one lead than the other (a band around a cylindrical package, for example). This marked lead is always the *cathode*.

Figure 1.4 shows a typical diode package.

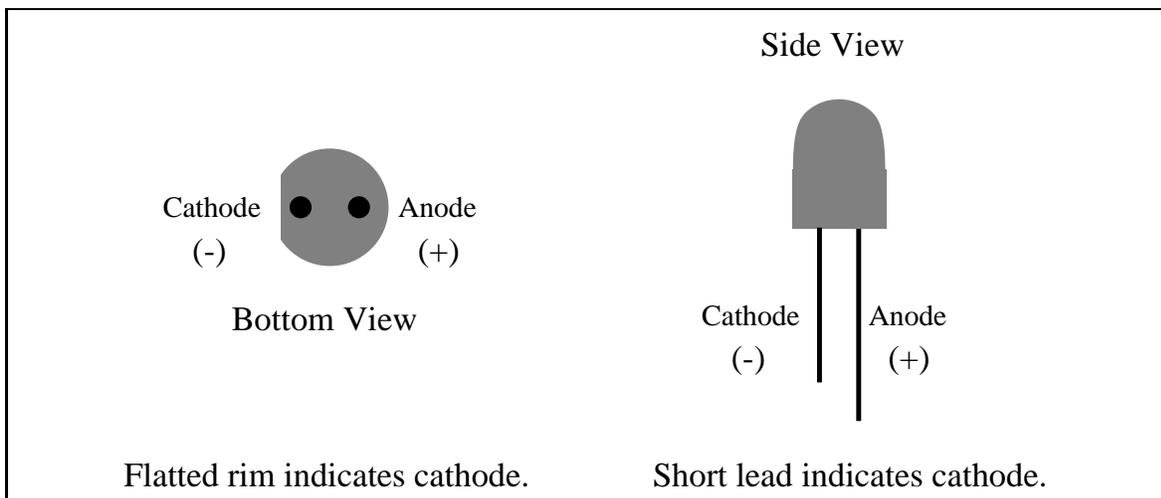


Figure 1.5: Identifying LED Leads

**LEDs** *LED* is an acronym for “light emitting diode,” so it should not come as a surprise that LEDs are diodes too. An LED’s cathode is marked either by a small flat edge along the circumference of the diode casing, or the shorter of two leads.

Figure 1.5 shows a typical LED package.

**Integrated Circuits** Integrated circuits, or ICs, come in a variety of package styles. Two common types, both of which are used in the 6.270 board design, are called the *DIP* (for *dual-inline package*), and the *PLCC* (for *plastic leaded chip carrier*).

In both types, a marking on the component package signifies “pin 1” of the component’s circuit. This marking may be a small dot, notch, or ridge in the package.

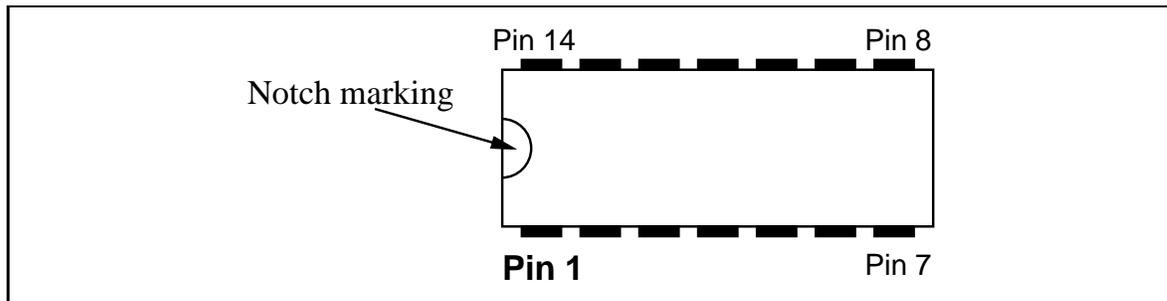


Figure 1.6: Top View of 14-pin DIP

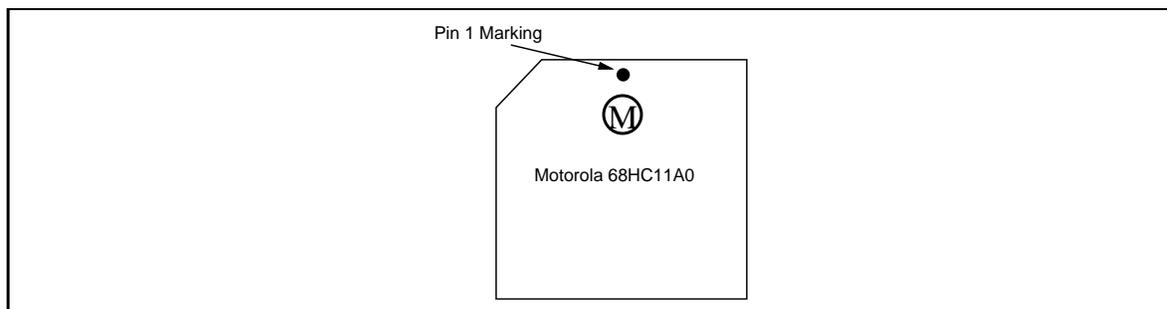


Figure 1.7: Top View of 52-pin PLCC

After pin 1 is identified, pin numbering proceeds sequentially in a counter-clockwise fashion around the chip package.

Figure 1.6 shows the typical marking on a DIP package. Figure 1.7 is a drawing of the PLCC package.

**DIP Sockets** Most of the integrated circuits (ICs) are socketed. This means that they are not permanently soldered to the 6.270 board. Components that are socketed can be easily removed from the board if they are damaged or defective.

Do not place the components into the sockets before you mount the sockets onto the board! Sockets are also used to avoid the need to solder directly to ICs, reducing the likelihood of heat damage.

DIP sockets also have a similar marking to those found on the components they will be holding. DIP sockets are not mechanically polarized, but the marking indicates how the chip should be mounted into the socket after the socket has been soldered into the board.

**PLCC Sockets** PLCC sockets *are* polarized, however: a PLCC chip can only be inserted into its socket the “correct” way. Of course, this way is only correct if the socket is mounted right in the first place.

When assembling the 6.270 board, a marking printed onto the board indicates the correct orientation of the PLCC socket.

**Capacitors** Quite a few different kinds of capacitors are made, each having different properties. There are three different types of capacitors in the 6.270 kit:

- **Monolithic.** These are very small-sized capacitors that are about the size and shape of the head of a match from a matchbook. They are excellent for use when small values are needed ( $0.1 \mu\text{F}$  and less). They are inexpensive and a fairly new capacitor technology. Monolithic capacitors are always non-polarized.
- **Electrolytic.** These capacitors look like miniature tin cans with a plastic wrapper. They are good for large values ( $1.0 \mu\text{F}$  or greater). They become bulky as the values increase, but they are the most inexpensive for large capacitances. Electrolytics can have extremely large values ( $1000 \mu\text{F}$  and up). They are usually polarized except for special cases; all the electrolytics in the 6.270 kit are polarized.
- **Tantalum.** These capacitors are compact, bulb-shaped units. They are excellent for larger values ( $1.0 \mu\text{F}$  or greater), as they are smaller and more reliable than electrolytic. Unfortunately they are decidedly more expensive. Tantalum capacitors are always polarized.

As indicated, some capacitors are non-polarized while other types are polarized. It's important to mount polarized capacitors correctly.

On the 6.270 boards, all polarized capacitor placements are marked with a plus symbol (+) and a minus symbol (-). The capacitors themselves are sometimes obviously marked and sometimes are not. One or both of the positive or negative leads may be marked, using (+) and (-) symbols. In this case, install the lead marked (+) in the hole marked (+).

Some capacitors may not be marked with (+) and (-) symbols. In this case, one lead will be marked with a dot or with a vertical bar. This lead will be the positive (+) lead.

Polarized capacitors that are mounted backwards won't work. In fact, they often overheat and explode. Please take care to mount them correctly.

**Inductors** The inductor used in the 6.270 kit looks like a miniature coil of wire wound about a thin plastic core. It is about the size of a resistor.

Some inductors are coated with epoxy and look quite like resistors. Others are big bulky coils with iron cores.

Inductors are not polarized.

**Transistors** The transistors used in the 6.270 kit are small, three-wire devices. They are distinctive because they are the only three-wire devices used.

Transistors are polarized devices.

The table shown in Figure 1.8 summarizes this discussion with regard to polarity issues.

Device	Polarized?	Effect of Mounting Incorrectly
Resistor	no	
Isolated R-Pack	no	
Common R-Pack	yes	circuit doesn't work
Diode	yes	circuit doesn't work
LED	yes	device doesn't work
Monolithic capacitor	no	
Tantalum capacitor	yes	explodes
Electrolytic capacitor	yes	explodes
DIP socket	yes	user confusion
PLCC socket	yes	52-pin severe frustration
Integrated circuit	yes	overheating; permanent damage
Inductor	no	
Transistor	yes	circuit doesn't work

Figure 1.8: Summary of Polarization Effects

### 1.1.4 Component Mounting

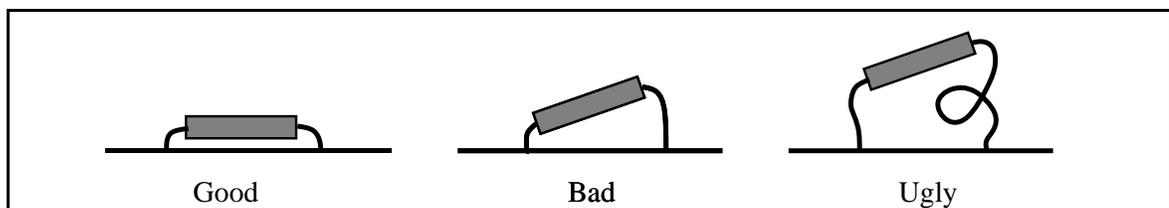


Figure 1.9: Flat Component Mounting

When mounting components, the general rule is to try to mount them as close to the board as possible. The main exception are components that must be folded over before being soldered; some capacitors fall into this category.

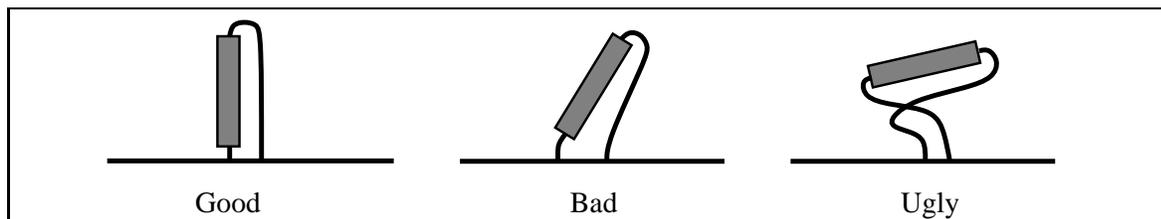


Figure 1.10: Upright Component Mounting

Most resistors and diodes must be mounted upright while others may lay flat. If space has been provided to mount the component flat, then do so, and try to keep it as close to the board as possible. If not, then just bend one lead over parallel to the component, and mount the component tightly.

See Figures 1.9 and 1.10 for clarification.

### 1.1.5 Component Value Markings

Various electronic components have their values marked on them in different ways. For the same type of component, say, a resistor, there could be several different ways that its value would be marked.

This section explains how to read the markings on resistors and capacitors. Other devices, such as transistors and integrated circuits, have their part number printed clearly on the device package.

#### Resistors

The largest resistors—in terms of wattage, not resistive value—simply have their value printed on them. For example, there are two large, rectangular  $7.5\Omega$  resistors in the 6.270 kit that are marked in this fashion.

Other resistors are labelled using a standard *color code*. This color code consists of three value bands plus a tolerance band. The first two of the three value bands form the value mantissa. The final value band is an exponent.

It's easiest to locate the tolerance band first. This is a metallic silver- or gold-colored band. If it is silver, the resistor has a tolerance of 10%; if it is gold, the resistor has a tolerance of 5%. If the tolerance band is missing, the tolerance is 20%.

The more significant mantissa band begins opposite the tolerance band. If there is no tolerance band, the more significant mantissa band is the one nearer to an end of the resistor.

Figure 1.11 shows the meaning of the colors used in reading resistors.

A few examples should make this clear.

- *brown, black, red*:  $1,000\Omega$ , or  $1k\Omega$ .

Color	Mantissa Value	Multiplier Value
Black	0	1
Brown	1	10
Red	2	100
Orange	3	1000
Yellow	4	10,000
Green	5	100,000
Blue	6	1,000,000
Violet	7	
Grey	8	
White	9	

Figure 1.11: Resistor Color Code Table

- *yellow, violet, orange*:  $47,000\Omega$ , or  $47k\Omega$ .
- *brown, black, orange*:  $10,000\Omega$ , or  $10k\Omega$ .

### Capacitors

Reading capacitor values can be confusing because there often are numbers printed on the capacitor that have nothing to do with its value. So the first task is to determine which are the relevant numbers and which are the irrelevant ones.

For large capacitors (values of  $1\mu\text{F}$  and greater), the value is often printed plainly on the package; for example, “ $4.7\mu\text{F}$ ”. Sometime the “ $\mu$ ” symbol acts as a decimal point; e.g., “ $4\mu 7$ ” for a  $4.7\mu\text{F}$  value.

Capacitors smaller than  $1\mu\text{F}$  have their values printed in picofarads (pF). There are 1,000,000 pF in one  $\mu\text{F}$ .

Capacitor values are similar to resistor values in that there are two digits of mantissa followed by one digit of exponent. Hence the value “472” indicates  $47 \times 10^2$  picofarads, which is 4700 picofarads or  $0.0047 \mu\text{F}$ .

## 1.2 The Microprocessor Board

The 6.270 Microprocessor Board is the brains and brawn of the 6.270 Robot Controller system. It uses a Motorola 6811 microprocessor equipped with 32K of non-volatile memory. It has outputs to drive four motors, inputs for a variety of sensors, a serial communications port for downloading programs and user interaction, and a host of other features.

### 1.2.1 Assembling the Microprocessor Board

Before beginning assembly, make sure to have a well-lighted, well-ventilated workspace. Make sure that all of the electronic assembly tools are available.

All of the 6.270 boards have component placements silkscreened directly onto the board. In addition, diagrams in these instructions will provide copies of the diagrams printed on the boards, often at better resolution. Refer to the printed diagrams as often as necessary to be sure that components are being placed correctly.

Figure 1.12 illustrates the component placement on the microprocessor board.

The instruction checklist may be marked off as each step is completed.

The component numbering for parts on the microprocessor board increments in a counter-clockwise fashion around the board for resistors, capacitors, and resistor packs.

- 1-□ **Get the 6.270 Microprocessor Board, and determine which is the “component side.”** The Microprocessor Board is the largest of the 6.270 boards.

The side of the board that has been printed with component markings is the “component side.” This means that components are mounted by inserting them down from the printed side; then they are soldered on the obverse, the unprinted side.

Please make sure that the components are mounted on the proper side of the board! It would be a terrible mistake to mount everything upside down.

- 2-□ **Resistor Packs.**

Begin by installing the resistor packs. Most of the resistor packs are polarized: the common terminal end is marked with a dot or band. On the 6.270 board, find a *square metal pad* at one end of the area that each resistor pack will mount. *Insert the resistor pack such that the marked end mounts in the square hole.* (The square hole is more easily discernable on the unprinted solder side of the board.)

The “caddy-cornering” technique of soldering the two end terminals first is helpful here.

- **RP1**–47k $\Omega$ ×9, 10 pins, polarized, marked “E47K $\Omega$ .”
- **RP2**–47k $\Omega$ ×4, 5 pins, polarized, marked “E47K $\Omega$ .”
- **RP3**–1k $\Omega$ ×3, 6 pins, non-polarized, marked “V1K $\Omega$ .”
- **RP4**–1k $\Omega$ ×5, 6 pins, polarized, marked “E1K $\Omega$ .”
- **RP5**–22k $\Omega$ ×3, 6 pins, non-polarized, marked “B223GA.”

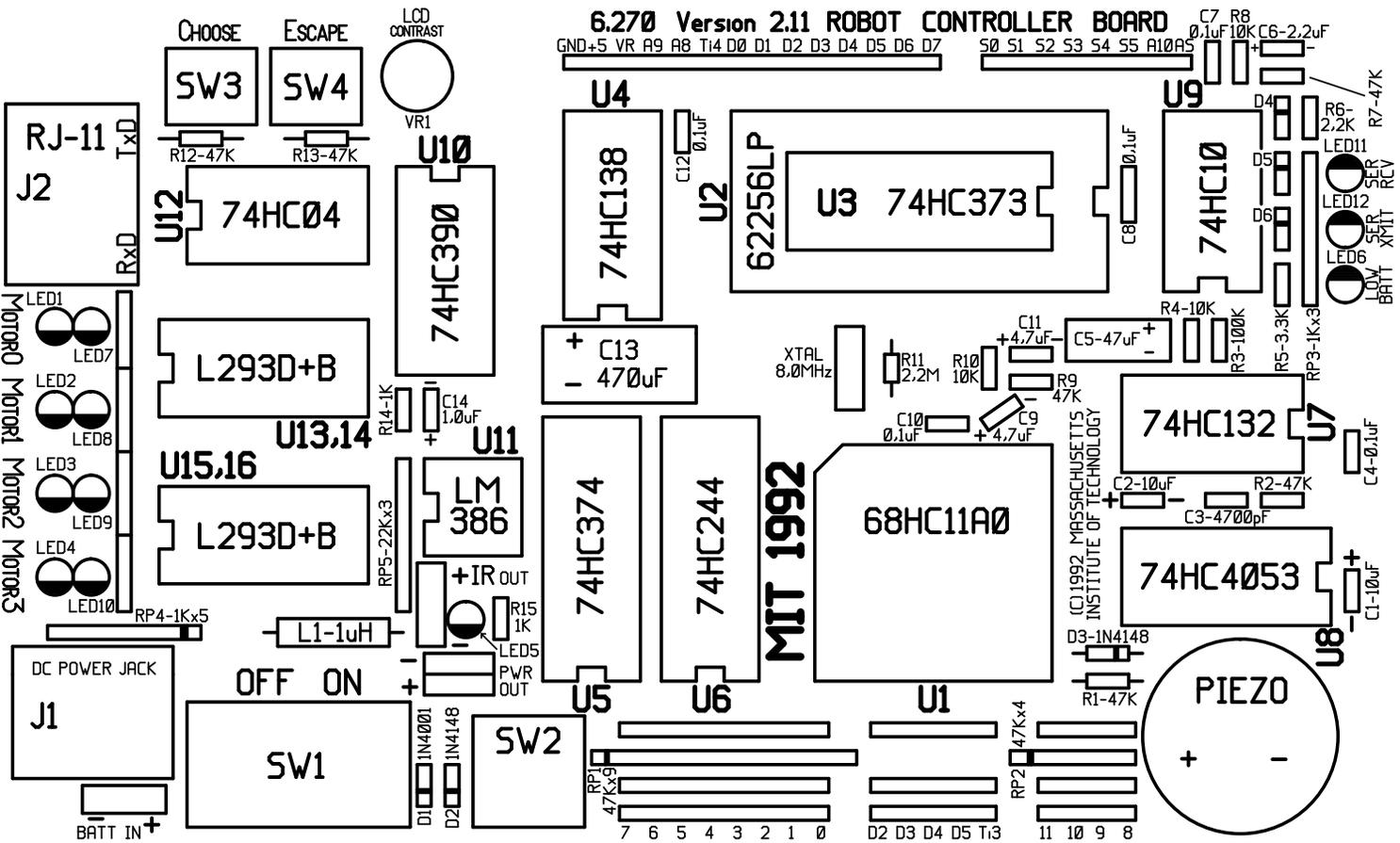


Figure 1.12: 6.270 Microprocessor Board Component Placement

**3-□ Non-polarized Capacitors.**

These capacitors are not polarized. After installing, solder and clip leads close to the board.

- **C3**–4700 pF, marked “472.”
- **C4**–0.1  $\mu$ F, marked “104.”
- **C7**–0.1  $\mu$ F, marked “104.”
- **C8**–0.1  $\mu$ F, marked “104.”
- **C10**–0.1  $\mu$ F, marked “104.”
- **C12**–0.1  $\mu$ F, marked “104.”

**4-□ IC Sockets.**

Mount the DIP sockets such that the notch in the socket lines up with the notch marking in the rectangular outline printed on the PC board. “DIP4” means the DIP socket for integrated circuit U4.

Note that there are two varieties of 16-pin DIP socket. One type is gold-plated and costs about ten times as much as the inexpensive variety. This type is to be used for the motor driver ICs to make better contact and is installed as indicated in the instructions.

The caddy-cornering technique should help here too. After inserting a DIP into the board, solder its two opposite-corner pins first. This will hold the chip in place. Make sure it is pressed down as far as it can go; then solder the other pins.

- **DIP4**–16 pins
- **DIP5**–20 pins
- **DIP6**–20 pins
- **DIP7**–14 pins
- **DIP8**–16 pins
- **DIP9**–14 pins
- **DIP10**–16 pins
- **DIP11**–8 pins
- **DIP12**–14 pins
- **DIP13/14**–16 pins, gold-plated pins
- **DIP15/16**–16 pins, gold-plated pins

**5-□ Direct Mount Chip.**

One chip is soldered directly to the board. Be careful not to apply too much heat to its pins when soldering. The soldering iron should not be in contact with any given pin for more than about eight seconds. It's okay to wait for things to cool down and try again if problems arise.

Mount this chip such that its notch is aligned with the rectangular notch printed on the PC board.

- **U3-74HC373.**

**6-□ Remaining Sockets.**

- With the orange utility knife, cut center support from **DIP2**, a 28-pin socket. Install on top of **U3**, with the notch marking as indicated. Solder.
- Install **PLCC1**, 52-pin square socket for the 6811. The Pin 1 marking is indicated by the numeral "1" and an arrow in the socket; this marking mounts nearest to **U2**, the 32K RAM chip. There should be a beveled notch in the upper-left corner of the chip and the outline printed on the board, with respect to the pin 1 marking. *Be absolutely sure to mount this socket correctly; the socket is polarized and will only let you mount the chip into it one way.* Solder.

**7-□ Female Socket Headers.**

To cut socket headers to length, repeatedly score between two pins using the orange knife. Score on both sides of one division and then snap the strip in two. *Do not try to snap header pieces before they have been sufficiently scored,* or they will break, destroying one or both of the end pieces in question.

When mounting the sockets, pay attention to how well they are lining up vertically. Sometimes reversing the way a strip is mounted will help its connections to line up better with the others. It may be helpful to insert a strip of male header into the socket to hold them at proper horizontal and vertical placement before soldering.

Refer to Figure 1.13 for placement of these parts.

- Cut three 8-long strips, *or* use six 4-long strips. Install the **Digital Input** connector block. Solder.
- Cut three 5-long strips. Install the **Port D I/O** connector block. Solder.
- Use three 4-long strips. Install the **Analog Input** connector block. Solder.
- Cut one 12-long strip. Install the **Motor Output** connectors. Solder.

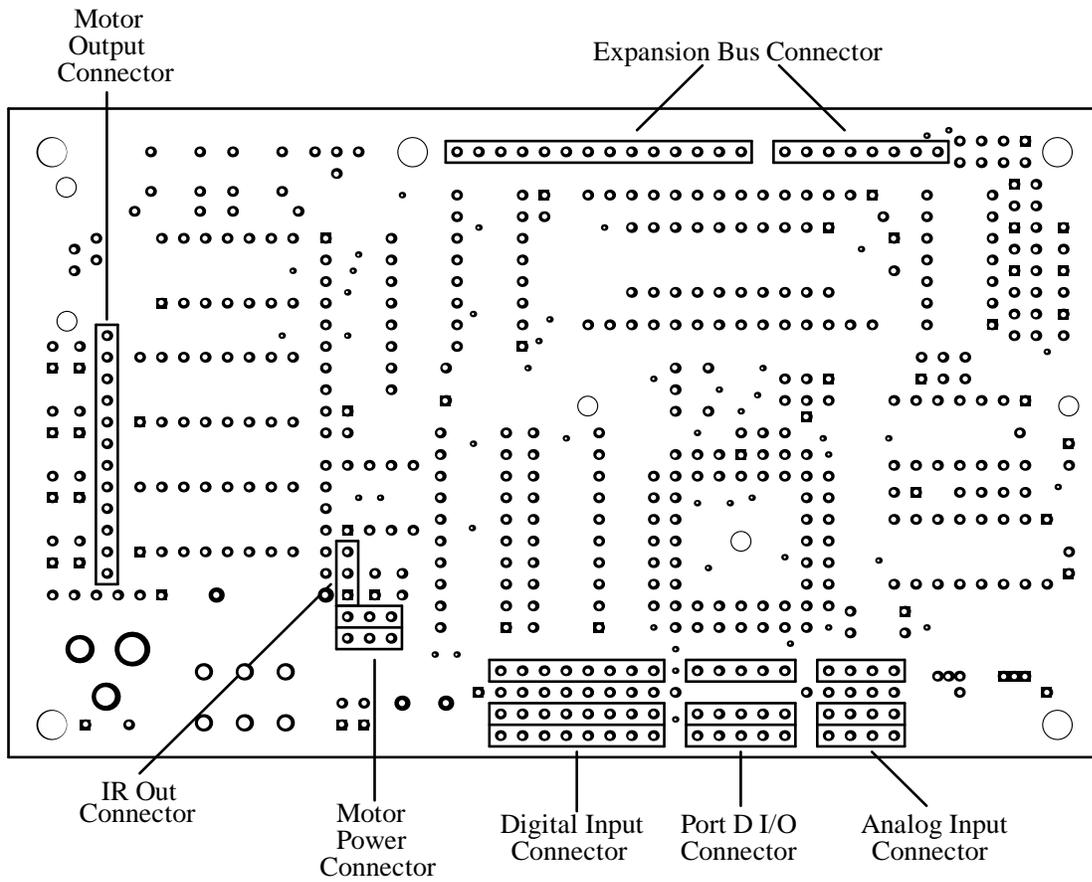


Figure 1.13: 6.270 Microprocessor Board Header Placement

- Cut one 8-long and one 14-long strip. Install the **Expansion Bus** connector. Solder.
- Cut two 3-long strips. Install the **Motor Power** connector. Solder.
- Cut one 3-long strip. Install the **IR Out** connector. Solder.

### 8-□ LEDs.

LEDs must be mounted so that the *short lead* (the cathode) is inserted into the *shaded half* of the LED placement marking.

Be sure to mount LEDs properly as it is very difficult to desolder them if they are mounted backward.

- LED1**—red
- LED2**—red
- LED3**—red
- LED4**—red
- LED5**—red
- LED6**—red
- LED7**—green
- LED8**—green
- LED9**—green
- LED10**—green
- LED11**—green
- LED12**—yellow

### 9-□ Resistors.

Most resistors mount vertically: try to mount them perfectly upright, with one end very close to the board, and the wire lead bent around tightly.

Others mount flat on the board: try to get the body of the resistor very close to the board.

*If you have trouble discerning colors*, you may wish to have your teammates handle this task. It is fairly difficult to read the color bands from  $\frac{1}{8}$  watt resistors, even to the trained eye.

- R1**—47k $\Omega$ , *yellow, violet, orange*
- R2**—47k $\Omega$ , *yellow, violet, orange*

- **R3**–100k $\Omega$ , *brown, black, yellow*
- **R4**–10k $\Omega$ , *brown, black, orange*
- **R5**–3.3k $\Omega$ , *orange, orange, red*
- **R6**–2.2k $\Omega$ , *red, red, red*
- **R7**–47k $\Omega$ , *yellow, violet, orange*
- **R8**–10k $\Omega$ , *brown, black, orange*
- **R9**–47k $\Omega$ , *yellow, violet, orange*
- **R10**–10k $\Omega$ , *brown, black, orange*
- **R11**–2.2M $\Omega$ , *red, red, green*
- **R12**–47k $\Omega$ , *yellow, violet, orange*
- **R13**–47k $\Omega$ , *yellow, violet, orange*
- **R14**–1k $\Omega$ , *brown, black, red*
- **R15**–1k $\Omega$ , *brown, black, red*

#### 10–□ **Large Capacitors.**

All of these capacitors are polarized. Make sure that the lead marked (+) on the capacitor goes into the hole that is marked (+). If the capacitor leads are not marked (+) or (–), the lead marked with a dot or bar is the (+) lead.

- **C1**–10  $\mu\text{F}$
- **C2**–10  $\mu\text{F}$
- **C5**–47  $\mu\text{F}$ . Fold capacitor flat to the board before soldering.
- **C6**–2.2  $\mu\text{F}$
- **C9**–4.7  $\mu\text{F}$
- **C11**–4.7  $\mu\text{F}$
- **C13**–470  $\mu\text{F}$ . Fold capacitor flat to the board before soldering.
- **C14**–1  $\mu\text{F}$

#### 11–□ **Diodes.**

Diodes are polarized. Mount them such that the lead nearer the banded end goes into the square hole on the circuit board.

- **D1**–1N4001. This diode has a black epoxy body and fairly thick leads.
- **D2**–1N4148. This is a glass-body diode.

- **D3**–1N4148
- **D4**–1N4148
- **D5**–1N4148
- **D6**–1N4148

**12**–□ **Ceramic Resonator.**

Install **XTAL1**, 8 Mhz. ceramic resonator.

**13**–□ **Inductor.**

Install **L1**, 1 uH.

**14**–□ **Switches.**

- **SW1**–DPDT slide switch
- **SW2**–large red pushbutton switch
- **SW3**–miniature pushbutton switch
- **SW4**–miniature pushbutton switch

**15**–□ **Trimpot.**

Install **VR1**, 100kΩ.

**16**–□ **Piggy-Backing the L293 Chips.**

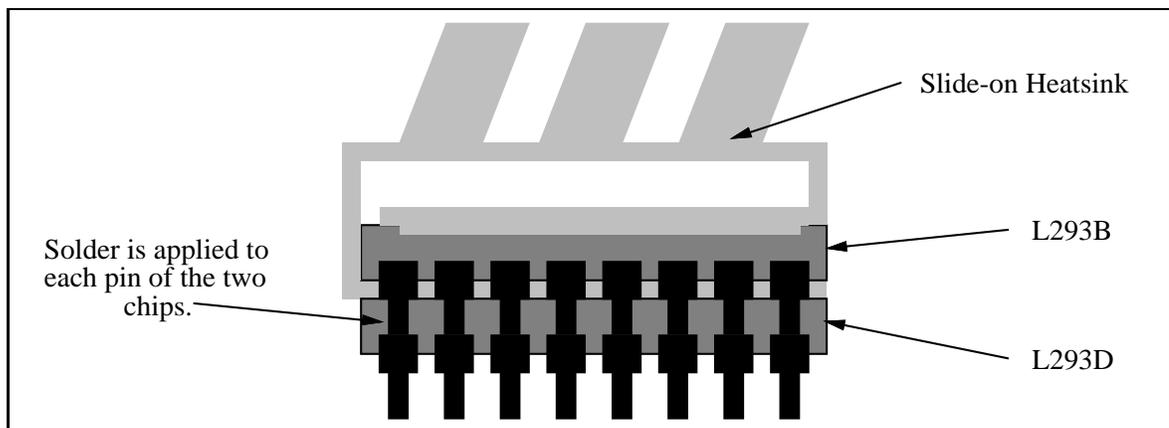


Figure 1.14: Motor Chip Stacking Technique

Motor driver chips **U13/14** (L293D plus L293B) and **U15/16** (L293D plus L293B) will be piggy-backed and soldered together before installing in their socket.

The instructions will be given for one pair and can be repeated for the second pair. *Make sure* that each pair consists of one L293D and one L293B chip!

Begin by sliding the gold-colored heat sink over an L293B chip. Then, press this assembly onto an L293D chip, as indicated in Figure 1.14. Make sure that the two chips have their notches lined up. Also, be sure to remember where which way the notches face, as they may be obscured.

Finish by soldering the two chips together, pin by pin. Try to have them pressed together as is possible, so that both press firmly against the heat sink.

Repeat for the other pair of motor driver chips.

### 17-□ Power Jack.

Install **J1**, DC power jack. When soldering, use ample amounts of solder so that solder completely fills mounting pads.

### 18-□ Phone Jack.

Install **J2**, modular phone jack.

### 19-□ Piezo Beeper.

Mount the piezo beeper so that it is centered on circular outline. Polarity does not matter.

### 20-□ Battery pack.

- Clip connector and about 1" of length off battery pack leads.
- *From bottom of board*, insert leads for battery pack. Note polarization: black lead goes in hole marked (-), red lead in hole marked (+). Solder from top of board and clip leads.

## 1.2.2 Testing the Microprocessor Board

This section explains a few simple tests to be performed *before* installing the ICs in the sockets.

Full board testing and debugging will be handled in the laboratory.

### 1-□ Check the solder side of the board for proper solder connections. Specifically: look for solder bridges and cold solder joints.

Solder bridging is when a piece of solder “bridges” across to adjacent terminals that should not be connected.

Cold solder joints are recognized by their dull luster. A cold solder joint typically makes a flaky electrical connection. Make sure that all of the solder joints are shiny with a silver color.

Make sure that joints do not have too much solder.

- 2- Check continuity (resistance) between power and ground of your board. Power may be obtained from the cathode of D1 and ground from the black lead of the battery pack.

Resistance should increase as the board capacitor charges. The board resistance should measure between  $1\text{k}\Omega$  and  $10\text{k}\Omega$ . *If a reading of zero ohms is observed, the board probably has a power to ground short.* Do not proceed with testing until this is corrected.

- 3- Insert 4 AA batteries into battery holder.

- 4- Turn on board power switch.

- 5- Examine the yellow LED: it should be glowing slightly. If not, turn off board power immediately. Check for power short.

- 6- Measure board voltage (as above with continuity check). You should have approximately 5.5 volts.

- 7- Install ICs in the board. *Be careful not to damage the component leads when installing the chips into their sockets!* Make sure to get the orientation correct—refer to Figure 1.12 if necessary.

- U1**–68HC11A0 microprocessor
- U2**–62256LP 32K static RAM
- U3**–74HC373 (already soldered to board)
- U4**–74HC138
- U5**–74HC374
- U6**–74HC244
- U7**–74HC132
- U8**–74HC4053
- U9**–74HC10
- U10**–74HC390
- U11**–LM386
- U12**–74HC04

- **U13,14**–L293D + L293B assembly with heatsink
- **U15,16**–L293D + L293B assembly with heatsink

### 1.2.3 After Board Checkout

The following final assembly step should be done only after the board has been shown to work properly. It is difficult to debug a board once the battery pack has been bolted on.

- 1–□ Use  $2 \times 56$  metal bolts, nylon washers, and nuts to attach battery pack to board. Insert bolts into battery pack and then up from the underside of the board so that the nuts screw down from the top of the board. *Make sure* to use nylon washers to protect the top of the board from the metal nuts.

## 1.3 The Expansion Board

The 6.270 Expansion Board plugs on top of the 6.270 Microprocessor Board, using the Expansion Bus connector. The Expansion Board adds the following capabilities:

- analog multiplexers to provide eight times more analog inputs;
- four DIP configuration switches;
- a user-adjustable “frob knob” for analog input;
- drivers for two additional motors;
- drivers for two LED/lamp circuits;
- a general purpose prototyping construction area.

Figure 1.15 is a component placement guide for the Expansion Board.

### 1.3.1 Assembling the Expansion Board

- 1–□ **Get the 6.270 expansion board, and determine which is the component side.**

As with the microprocessor board, the side that has white component markings is the component side. The obverse is the solder side.

- 2–□ **Resistor Pack.**

**RP6**,  $1k\Omega \times 7$ , 8 pins, polarized, *marked “E1K $\Omega$ .”* Mount so that marked end of resistor pack goes in square hole on board.

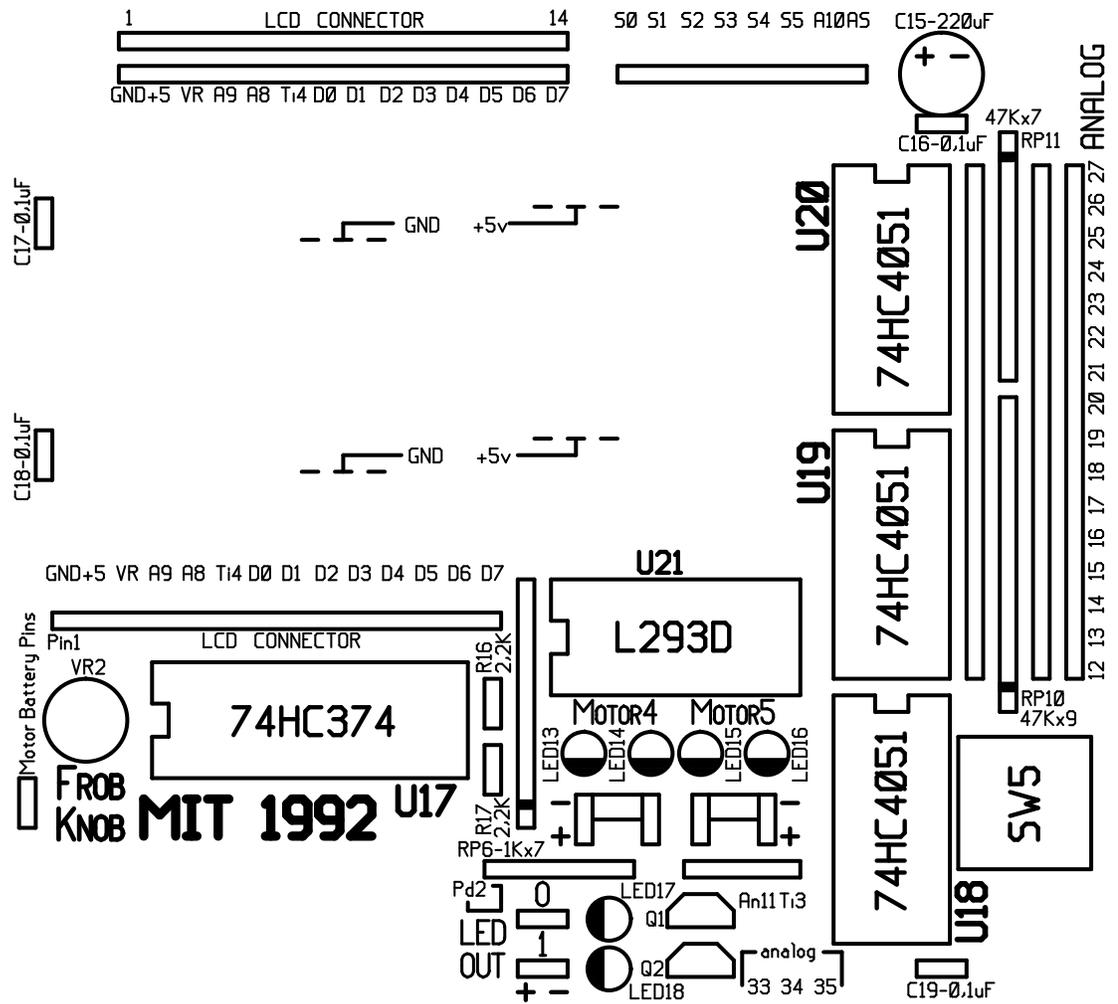


Figure 1.15: Expansion Board Component Placement

**3-□ IC Sockets.**

Align notch marking of socket with notch on printed chip outline on board.

- DIP17**–20 pins
- DIP18**–16 pins
- DIP19**–16 pins
- DIP20**–16 pins
- DIP21**–16 pins, gold-plated pins

**4-□ LEDs.**

Install LEDs so that the short lead mounts in the shaded half of the placement marking. Be careful to get polarity correct.

- LED13**–red
- LED14**–green
- LED15**–red
- LED16**–green
- LED17**–red
- LED18**–red

**5-□ Trimpot.**

Install **VR2**, 100k $\Omega$ .

**6-□ Resistors.**

- R16**, 2.2k $\Omega$ , *red, red, red*. Value of “4.7K” printed on board is incorrect.
- R17**, 2.2k $\Omega$ , *red, red, red*. Value of “4.7K” printed on board is incorrect.

**7-□ Capacitors.**

- C15**–220 $\mu$ F, *polarized*. Be sure to mount with correct polarity.
- C16**–0.1 $\mu$ F, non-polarized
- C17**–0.1 $\mu$ F, non-polarized
- C18**–0.1 $\mu$ F, non-polarized
- C19**–0.1 $\mu$ F, non-polarized

**8-□ Transistors.**

Install transistors **Q1** and **Q2** (type MPS2222A) where indicated on the Expansion Board. The transistors mount so that their flat edge is above the flat edge of the placement marking.

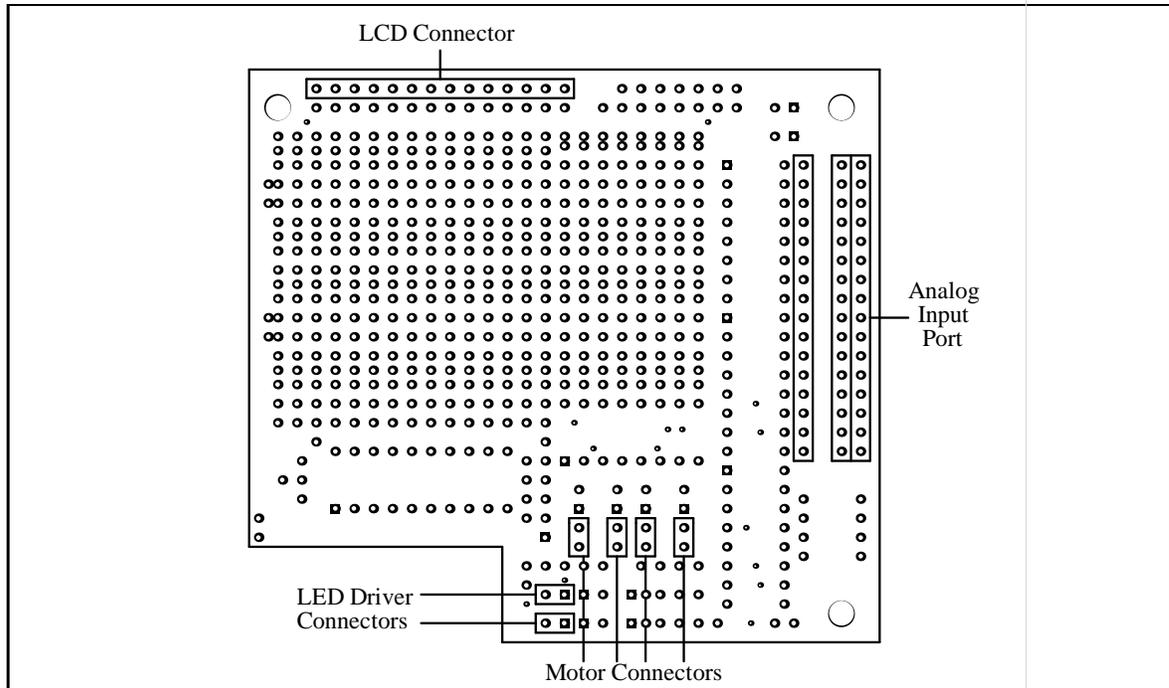
**9-□ Female socket headers.**

Figure 1.16: Expansion Board Female Header Mounting

Refer to Figure 1.16 to be sure of placement of these parts.

- Cut three 16-long strips. Install the **Analog Input Port**. Solder.
- Cut one 14-long strip. Install the **LCD Connector**. *Note:* The correct position for this header is *not* the location marked *LCD CONNECTOR* on the board. The correct position is indicated properly in Figure 1.16, at the top edge of the board.
- Cut six 2-long strips. Install **Motor Connectors** and **LED Driver Connectors**.

**10-□ DIP Switches.**

Install **SW5**, 4-position DIP switch. Install so that switch handles face outward over edge of board and are easy to manipulate.

## 11-□ Male Header Pins.

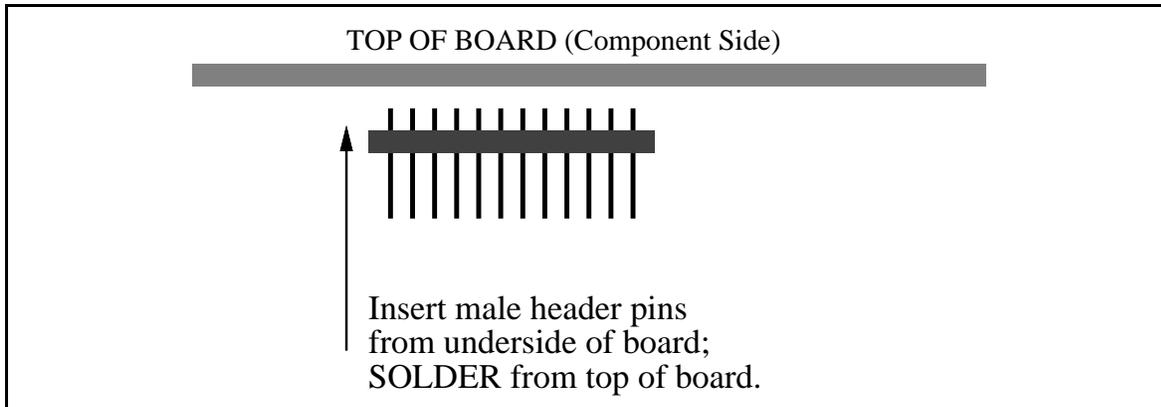


Figure 1.17: Mounting Method for Male Header Pins

The following steps deal with the interface pins that protrude from the Expansion Board to the Microprocessor Board.

When mounting these pins, insert upward from the underside of the board so that the maximal pin lengths protrude downward (see Figure 1.17). These pins are then soldered from the top, component side of the board.

Be careful to make sure the pins are mounted perfectly normal to the surface of the Expansion Board, as there are quite a few pins that must all mate properly with the Microprocessor Board.

For the following instructions, refer to Figure 1.18 for pin placement.

- **Motor Battery Pins**—a 2-long strip
- **Port D Connector**—a 5-long strip
- **Analog Port Connector**—a 4-long strip
- **Expansion Bus Connector**—one 14-long and one 8-long strip

### 1.3.2 Testing the Expansion Board

As with the Microprocessor Board, run through the following checklist before mounting the chips into the Expansion Board. Thorough testing will be performed in lab.

- 1-□ Check the solder side of the board for proper solder connections. Specifically: look for solder bridges and cold solder joints.

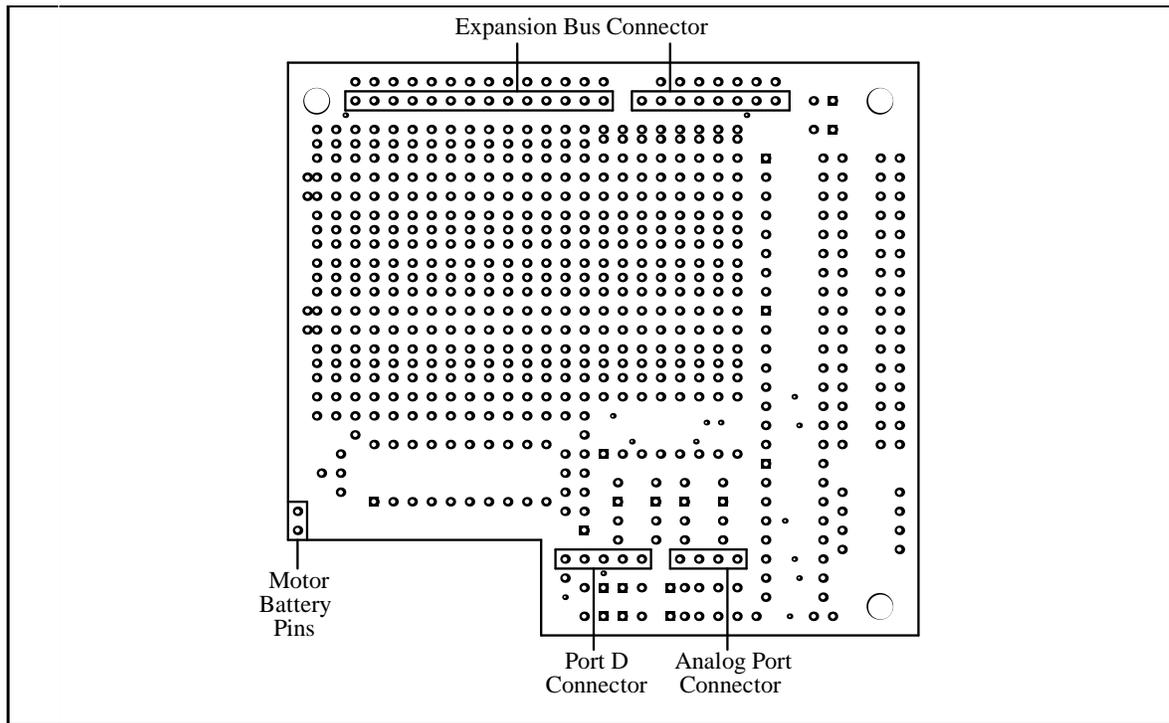


Figure 1.18: Expansion Board Male Header Pin Placement

- 2-□ Check continuity (resistance) between power and ground of the board. Power and ground can be located in the prototyping area.

Resistance should increase as the board capacitor charges. There should be a reading of between one and ten kilo-ohms. *If there is a reading of zero ohms, or near zero ohms, the board has a power short. Do not proceed with testing until this is corrected.*

- 3-□ Install ICs in the board, observing correct polarity:

- **U17**-74HC374
- **U18**-74HC4051
- **U19**-74HC4051
- **U20**-74HC4051
- **U21**-L293D. Slide gold heat sink onto chip before installing in socket.

- 4-□ See a lab TA for final board checkout.

### 1.3.3 After Board Testing

After both the Microprocessor Board and the Expansion Board have been tested and are working, the two boards may be bolted together at three points with the  $6\text{-}32 \times \frac{1}{2}$ " nylon standoffs and screws.

The standoff that is installed near the piezo beeper will need to be whittled a bit in order to mount properly.

## 1.4 The LCD Display

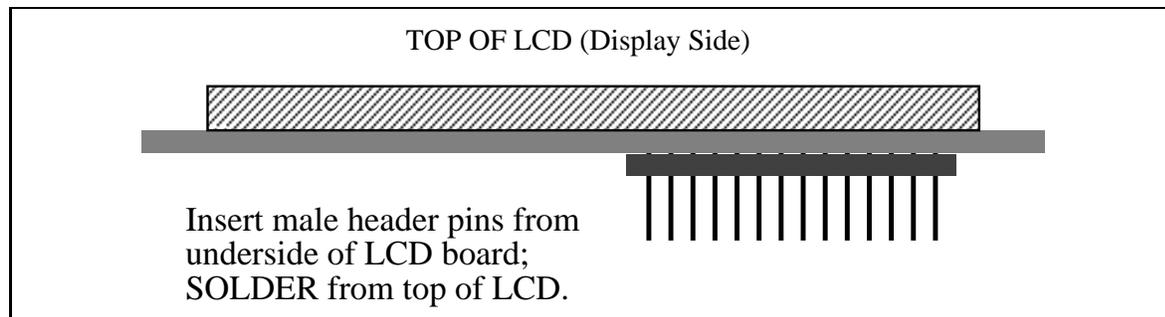


Figure 1.19: LCD Connector Mounting

The LCD display provided in this year's 6.270 kit can display two rows of 16 characters. The system software makes it easy to write code that prints messages to this display, for status, debugging, or entertainment purposes.

The display needs to have a 14-pin male header soldered to its interface. Figure 1.19 shows how these pins should be installed, in a similar fashion to the pins protruding from the Expansion Board.

Cut a 14-long male header strip and mount and solder to the LCD as indicated in the figure.

## 1.5 The Battery System

The 6.270 Robot Controller system has two battery power supplies. The first is the four AA alkaline cells that snap into the Microprocessor Board. These are used to run the microprocessor and sensors. They are also used to keep the program in the RAM when the board is switched off.

These batteries should power the microprocessor board for about fifty hours of operation before needing to be replaced. The board should not be left on inadvertently because the batteries will be drained.

The second power supply plugs into the motor power jack. The reason for having a separate battery for the motors is to provide isolation between the two supplies. When a motor turns on or reverses direction, it draws a huge surge of current. This causes fluctuations in the battery voltage. For motors, this is not a problem, but a microprocessor circuit could fail. For this reason, separate batteries are used for the motors and the microprocessor.

The motor battery is a bank of three Gates 2 volt lead-acid cells wired in series, yielding a 6 volt supply. Each cell is rated for 2.5 ampere-hours of operation.

These lead-acid cells are extremely powerful devices. Car batteries are constructed of similar lead-acid technology. *When handling the batteries, be extremely careful not to short the (+) an (-) terminals of the battery together.* A huge surge of current will flow, melting the wire and causing burns. In extreme cases, batteries can explode and cause serious injury.

The Gates cells were donated to 6.270 by Gates Energy Products, Inc.

The following instructions explain how to build the battery recharger and how to wire the Gates cells into power-packs. Note that contest rules prohibit using the Gates cells in any configuration other than what is presented here.

### 1.5.1 The Battery Charger

The battery charger can charge two 6 volt battery packs simultaneously. Each pack can be charged at either of two rates:

- **Normal charge.** Marked SLOW on the charger board, this is the normal charge position. A battery pack will recharge completely in about ten to fourteen hours. When the batteries become slightly warm they are fully charged.

When operating in normal mode, a green LED will be lit to indicate proper charging. In this mode, it is safe to leave batteries on charge for periods of up to 24 hours without causing damage.

- **Fast charge.** Marked FAST on the charger board, this position will recharge a battery pack in five to seven hours.

Batteries being charged in fast mode should be monitored closely; as soon as the pack becomes warm to the touch, the batteries are completely charged and should be removed from the charger.

Permanent damage to the battery pack can occur if left on fast charge for more than ten hours. Needless to say, this mode should be used with care.

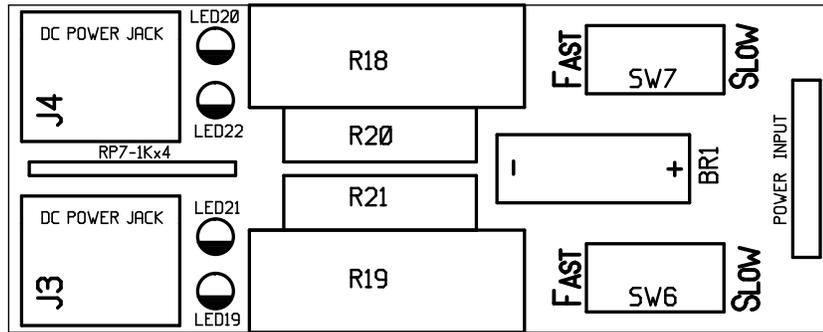


Figure 1.20: Battery Charger Component Placement

### Assembly Instructions

Figure 1.20 shows component placement on the battery charger board.

1-□ **Get the battery charger board and determine which is the component side.** The component side is marked with the placement guidelines.

2-□ **Resistor Pack.**

Install **RP7**,  $1.2\text{k}\Omega \times 4$ , 8 pins, marked “B122GA.” The board is labelled for  $1\text{k}\Omega$ ; this marking is incorrect. This resistor pack consists of four isolated resistors so orientation is not significant.

3-□ **LEDs.**

Mount LEDs so that the short lead is inserted in the shaded half of the placement marking.

- **LED19**—red
- **LED20**—red
- **LED21**—green
- **LED22**—green

4-□ **DC Power Jacks.**

Install **J3** and **J4**, DC power jacks. When soldering, use ample amounts of solder to fill the mounting holes completely.

5-□ **Power Resistors.**

- **R18**— $7.5\Omega$ , 5 watts

- **R19**–7.5 $\Omega$ , 5 watts
- **R20**–15 $\Omega$ , 2 watts, *brown, green, black*
- **R21**–15 $\Omega$ , 2 watts, *brown, green, black*

#### 6–□ **Slide Switches.**

- **SW6**–miniature SPDT slide switch
- **SW6**–miniature SPDT slide switch

#### 7–□ **Bridge Rectifier.**

Install **BR1**, rectangular bridge rectifier. *Observe polarity:* make sure (+) symbol on bridge rectifier is inserted into hole marked (+) on circuit board.

#### 8–□ **Power Cord.**

Get large DC power adapter. Clip off and discard the unusual power plug near the base of the plug. Strip  $\frac{1}{4}$ " of insulation from power wires. Insert stripped wires into holes marked POWER INPUT from component side of board; solder from solder side.

The polarity of the power connection is not significant.

#### 9–□ **After Checkout.**

After the battery charger is checked in lab, add a glob of hot glue as a strain relief at the base of the POWER INPUT connection.

### 1.5.2 Battery Pack Construction

The 6.270 kit includes 6 Gates cells, enough to make two battery packs. It is recommended that contest robots be designed in a fashion that facilitates battery pack swapping. This will prove invaluable in the final days of the course, when robots are being worked on continuously. One battery pack can be used to operate the robot while the other is being charged.

Two obvious alternatives for battery pack construction are depicted in Figure 1.21: a rectangular configuration and a triangular one. Other possibilities may be explored.

#### Wiring the Battery Cable

Figure 1.22 illustrates how to wire the battery plug and cable assembly.

- Cut a 12" to 16" length of the black/red twisted pair cable for use in making the battery cable. Strip and tin the wire ends.

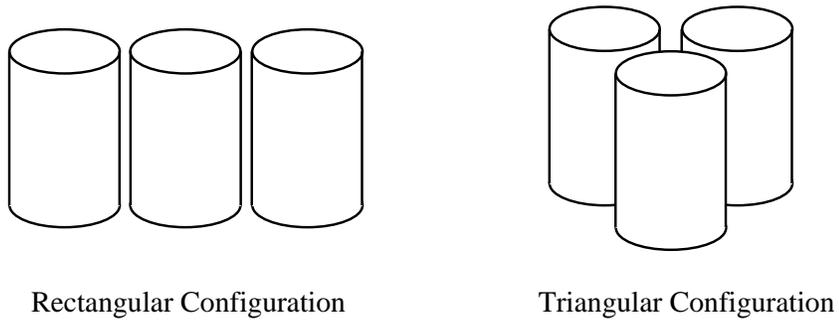


Figure 1.21: Two Battery Pack Configurations

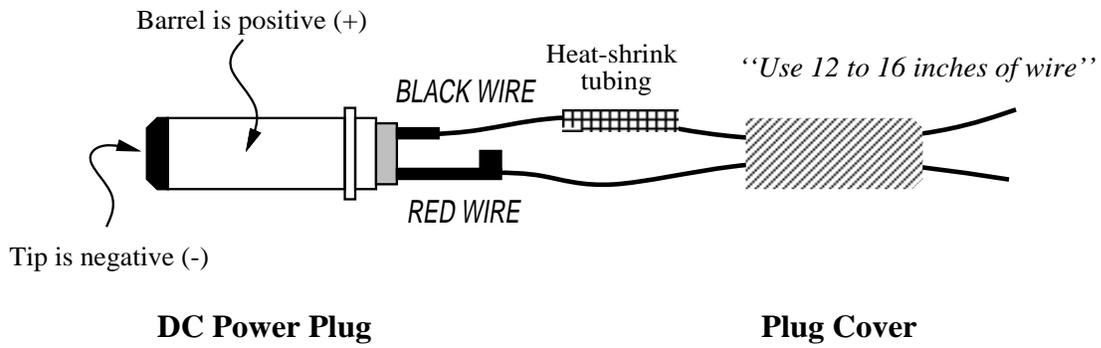


Figure 1.22: Battery Plug and Cable Wiring Diagram

- Heat shrink tubing is used on the shorter terminal of the DC power plug. The tubing acts as an insulator to minimize the likelihood of an electrical short at the plug terminals. *It is essential that this wiring be performed carefully because a short in the power plug will short out the battery terminals and create a serious hazard.*
- Proper polarity is important. The use of red wire to signify the (+) terminal and black wire to signify the (−) terminal is an international standard. Mount the black wire to the short terminal and the red wire to the long terminal.
- After soldering, slide the heat shrink tubing down over the short terminal and shrink it. Also, crimp the prongs of long terminal onto the red wire as a stress relief.
- Screw the plug cover onto the plug.
- *Before installing the cable onto a battery pack, use an ohmmeter to make absolutely sure that the cable is not shorted.* The cable should measure open circuit or infinite resistance. If a short is placed across the terminals of lead-acid batteries (like the Gates cells), a huge surge of current will flow, melting the wire causing the short and possibly causing the battery to explode.

### Constructing the Battery Pack

Wire the 3-cell pack to the battery cable as indicated in Figure 1.23. Use the red and black wire to make the two jumpers between the cells (color of these jumpers does not matter). *Make sure to get polarities correct.*

After the battery pack is wired, an overall configuration (as suggested in Figure 1.21) can be selected. The battery pack may be held in the desired configuration using a variety of materials, including rubber bands, cable ties, glue, and/or electrical tape.

## 1.6 The Infrared Transmitter

The infrared (IR) transmitter board emits modulated infrared light that can be detected by the Sharp IR sensors (of type GP1U52). The board has infrared transmitting LEDs that are driven by an oscillator circuit (the 74HC390 chip) and a power op-amp (LM386) on the Microprocessor Board.

Each infrared LED is wired in series with a visible LED, so that if current is flowing through the infrared LED, it must also flow through the corresponding visible LED. It should therefore be easy to determine if the IR LEDs are emitting light.

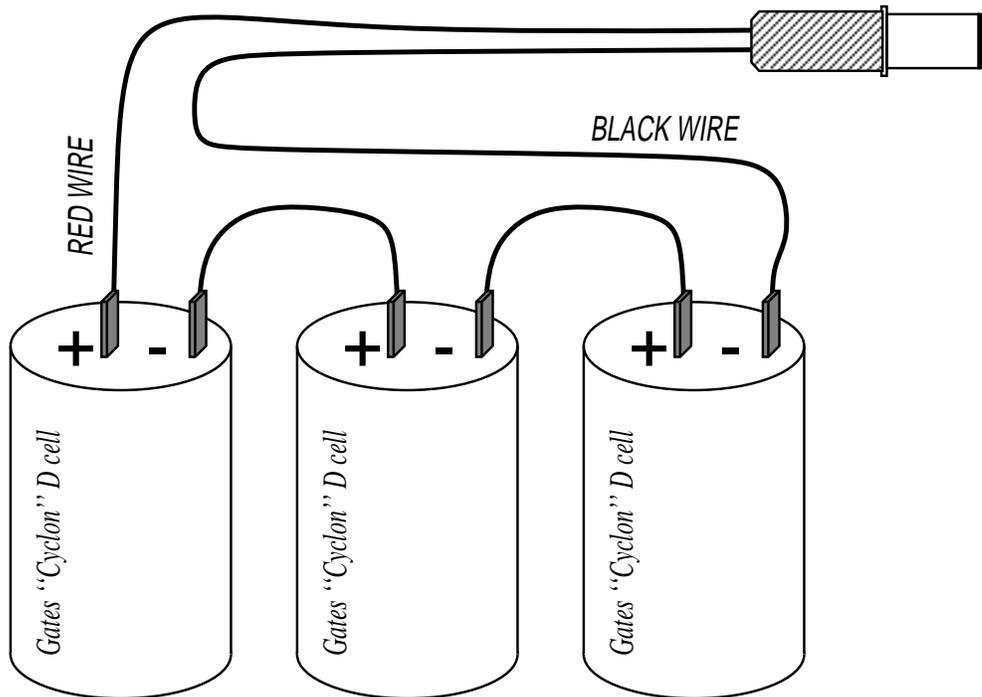


Figure 1.23: Battery Pack Wiring Diagram

### 1.6.1 Assembly Instructions

Figure 1.24 illustrates component placement on the infrared transmitter board. *Note that the LED numbering that was printed on the actual boards is incorrect. The numbering shown in the figure is correct.*

#### 1-□ Resistor Packs.

Both of the resistor packs are polarized. Mount so that the marked end of the resistor pack is placed into the square pad on circuit board.

- RP8-47Ω×4, 5 pins, marked “E47Ω.”
- RP9-47Ω×4, 5 pins, marked “E47Ω.”

#### 2-□ Visible LEDs.

The visible LEDs used on the infrared transmitter board have *clear lenses*. That is, they should look either clear or milky-white when they are unlit. *Be sure to use this variety of LED here.* These LEDs can handle more current than the LEDs that have been used in other circuitry. The LEDs will glow red when powered.

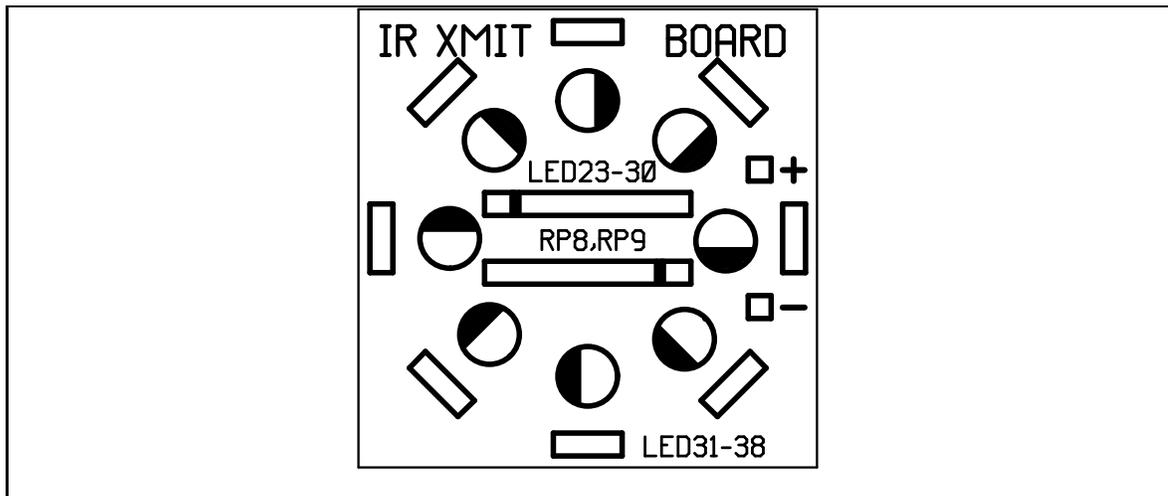


Figure 1.24: Infrared Transmitter Component Placement

Mount LEDs so that the short lead is inserted in the shaded half of the placement marking.

- **LED23**—clear lens, red element
- **LED24**—clear lens, red element
- **LED25**—clear lens, red element
- **LED26**—clear lens, red element
- **LED27**—clear lens, red element
- **LED28**—clear lens, red element
- **LED29**—clear lens, red element
- **LED30**—clear lens, red element

### 3-□ Infrared LEDs.

The infrared LEDs come in a small rectangular package.

When mounting, make sure that *the face with a small bubble* aims *outward* from the ring of LEDs. The bubble is the lens in front of the actual emitter element.

The face with the colored stripes must be on the *inside* of the ring.

- **LED31**—MLED71 IR LED
- **LED32**—MLED71 IR LED
- **LED33**—MLED71 IR LED

- LED34–MLED71 IR LED
- LED35–MLED71 IR LED
- LED36–MLED71 IR LED
- LED37–MLED71 IR LED
- LED38–MLED71 IR LED

#### 4–□ Cable and Connector.

- Cut a 12” length of the twisted-pair red/black cable. Strip  $\frac{1}{4}$ ” of insulation from the wire on both ends.
- From *underside* of IR board, insert red wire into hole marked (+) and black wire into hole marked (–). Solder from *top* of board.
- Mount other end of wire to outside pins of a three-pin male connector. Use guideline shown in Section 1.8.

The infrared transmitter plugs into the connector labelled IR OUT on the Microprocessor Board (see Figure 1.13), with the red lead inserted into the end marked (+) on the board.

## 1.7 The Motor Switch Board

The Motor Switch Board allows manual control of up to four motors. This is useful when testing and debugging mechanisms because the motors can be switched on forward, backward, and off easily.

It is important to realize that the amount of power delivered to the motors by the Motor Switch Board will be different than the amount delivered when the motors are driven by the electronics on the Microprocessor Board. The Motor Switch Board has diode circuitry to simulate the power loss of the Microprocessor Board’s control electronics, but there will still be a difference.

Motors driven from the Expansion Board will operate at even less power than those driven by the Microprocessor Board.

The careful designer will test mechanisms both from the Switch Board and from the Microprocessor Board before committing to them.

### 1.7.1 Assembly Instructions

Figure 1.25 provides a reference to parts mounting on the Motor Switch Board.

- 1–□ **Get Motor Switch Board, and determine which side is the component side.** The component side is marked with the parts placement layout.

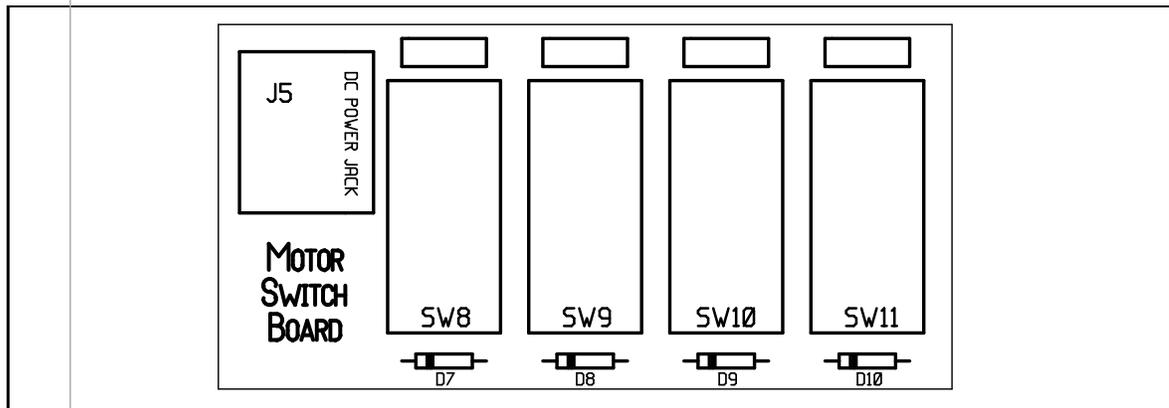


Figure 1.25: Motor Switch Board Component Placement

## 2-□ Diodes.

These diodes have black epoxy bodies. Polarity matters: Install the diodes with the banded end as marked on the circuit board.

- **D7**-1N4001
- **D8**-1N4001
- **D9**-1N4001
- **D10**-1N4001

## 3-□ DC Power Jack

Install **J5**, a DC power jack. Fill mounting holes completely with solder when soldering.

## 4-□ Switches.

- **SW8**-2 pole, 3 position slide switch
- **SW9**-2 pole, 3 position slide switch
- **SW10**-2 pole, 3 position slide switch
- **SW11**-2 pole, 3 position slide switch

## 5-□ Female Socket Headers.

Cut four 3-long pieces of female socket header. Mount in remaining holes on board where marked.

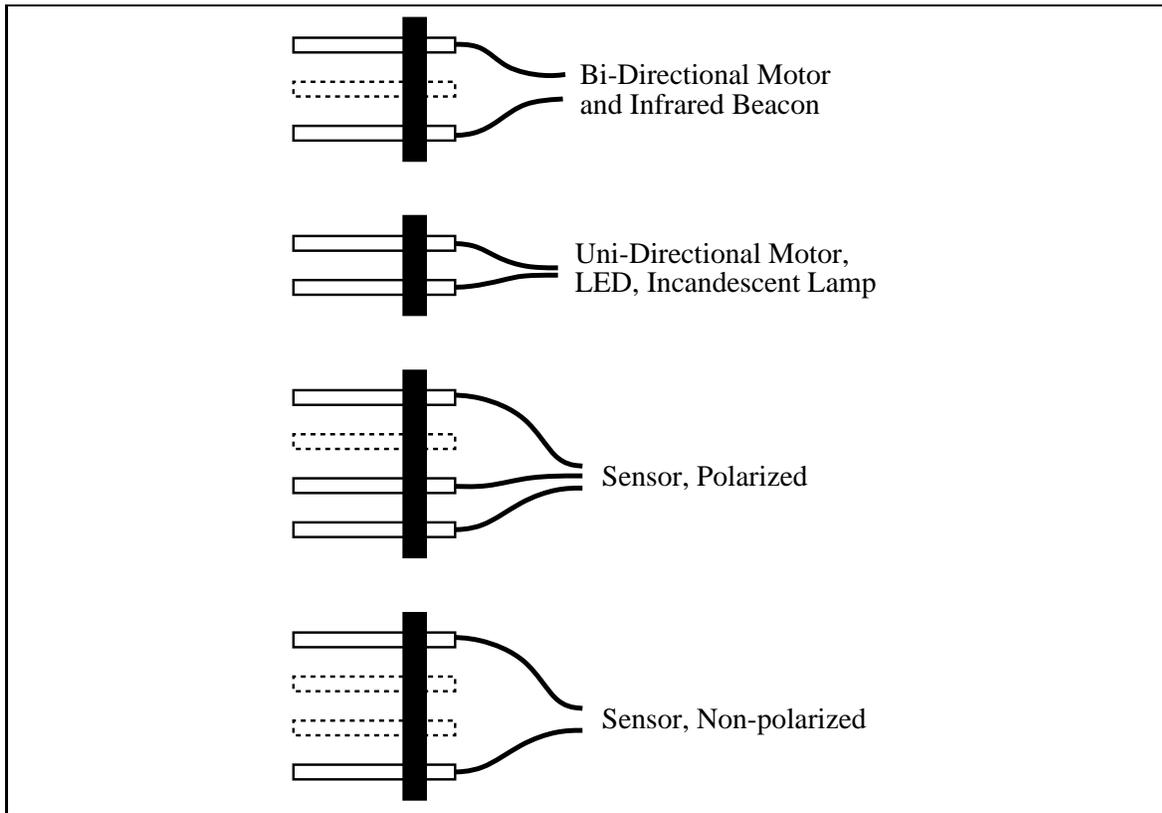


Figure 1.26: Standard Connector Plug Configurations

## 1.8 Cable and Connector Wiring

This section explains how to build reliable cables and connectors for the motors and sensors that will plug into the robot's controller boards.

Sturdy and reliable connectors are critical to the success of a robot. If a robot's connectors are built sloppily, hardware problems will occur. Well-built connectors will help make the robot more reliable overall and will ease development difficulties.

Sensors and motors are built with integral wiring; that is, a sensor or motor will have a fixed length of wire terminating in a connector. It is possible to build extension cables, but it is more time-efficient to build cables that are the proper length already.

The average robot has its control electronics near the physical center of the robot; hence, motors and sensor cables need to reach from the center of the robot to their mounting position. Given this geometry, most robots will need sensor and motor cables between 6 and 12 inches long.

Several different connector styles are used depending on the device which is being connected to. Figure 1.26 shows the connector configurations used for bidirectional motors, unidirectional motors, sensors, and the infrared beacon.

The ribbon cable provided in the 6.270 kit is best for making sensor and motor cables.

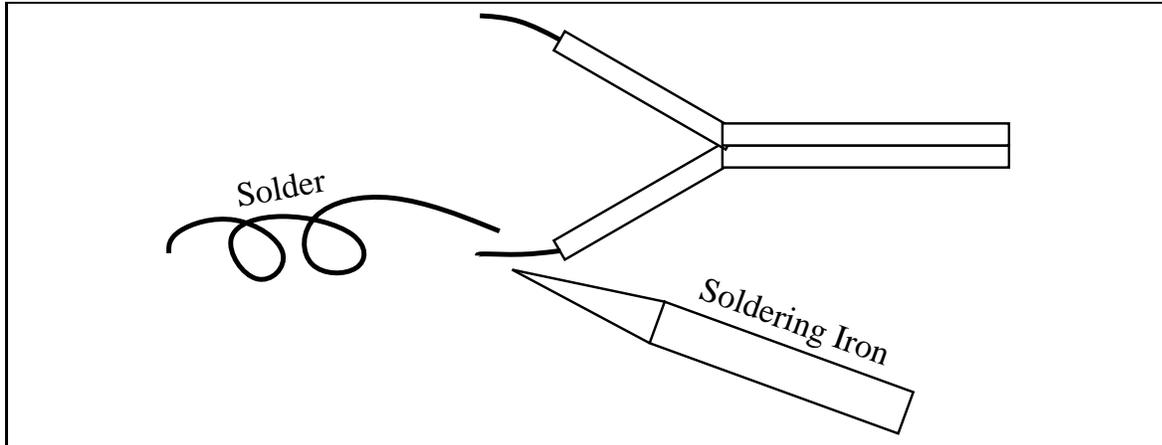
Figures 1.27 through 1.30 illustrate the recommended method for wiring to a connector plug. When assembled properly, this method will provide for a sturdy, well-insulated connector that will be reliable over a long period of use.

The example shows wiring to opposite ends of a three-pin plug, as would commonly be used when wiring to a motor. The method, however, is suitable for all kinds of connectors.

## 1.9 Sensor Wiring

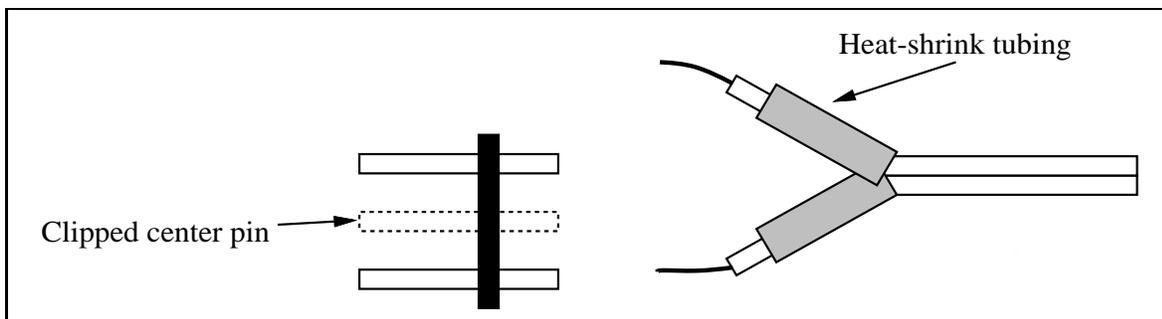
This section shows wiring diagrams for the following sensors:

- Touch sensors
- Photocell light sensor
- Infrared reflectance sensor
- Bend sensors
- Sharp infrared sensor
- Potentiometer position sensors
- Slotted optical switch



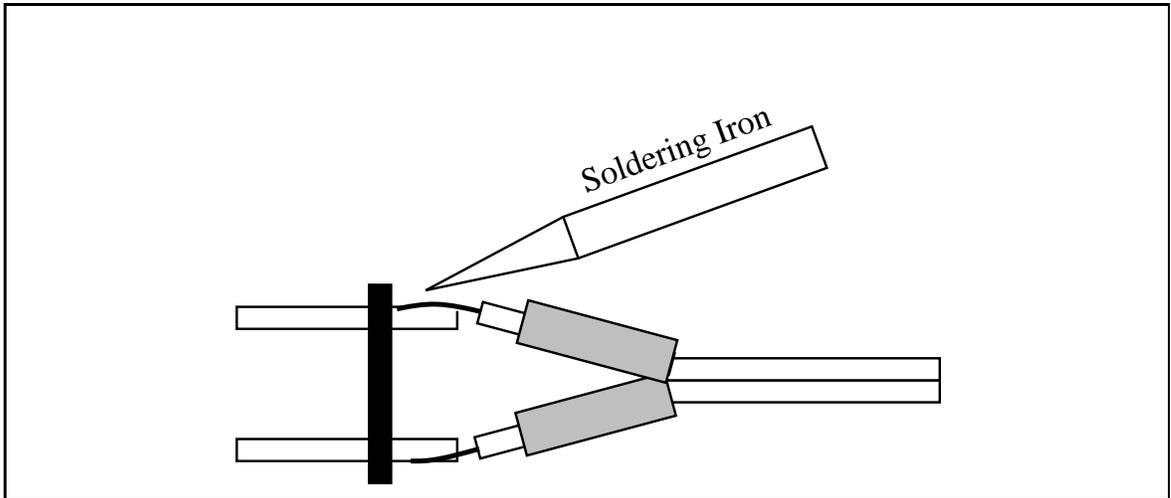
Strip a small amount of insulation off the wire ends. Tin the wire ends by applying a thin coat of solder to them.

Figure 1.27: Step One of Connector Wiring



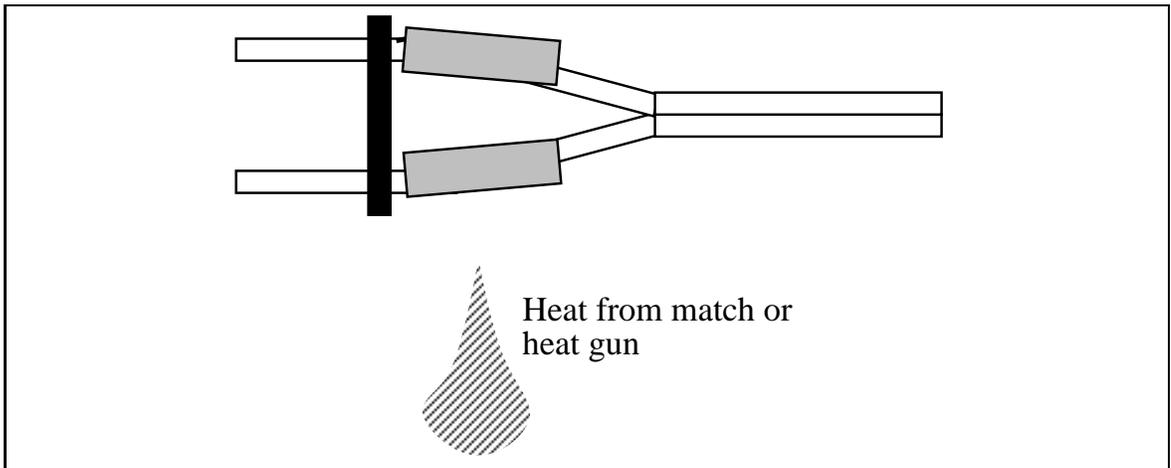
Cut the male connector to size. This example shows a plug that can be used to wire a motor or the infrared transmitter. Cut  $\frac{1}{2}$  inch length pieces of heat-shrink tubing, and slide over tinned wires.

Figure 1.28: Step Two of Connector Wiring



Hold the connector and wires in place and solder together. It may be helpful to clip wires to the length of the male pins before soldering.

Figure 1.29: Step Three of Connector Wiring



Slide pieces of heat-shrink tubing over connections. Shrink using heat gun, flame from a match or lighter, or the side of a soldering iron.

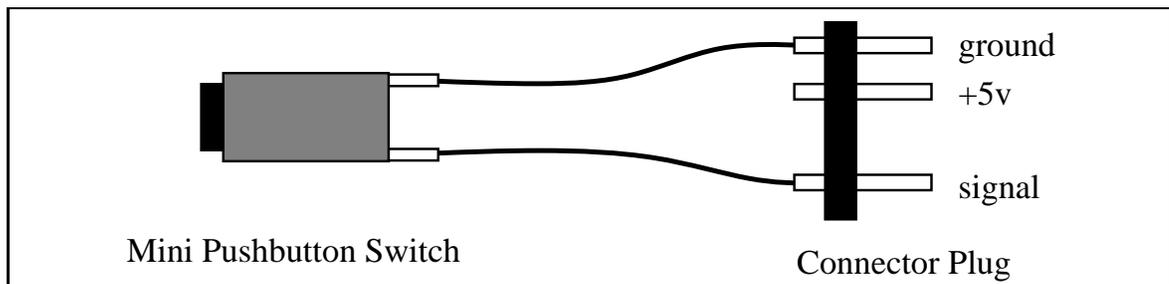
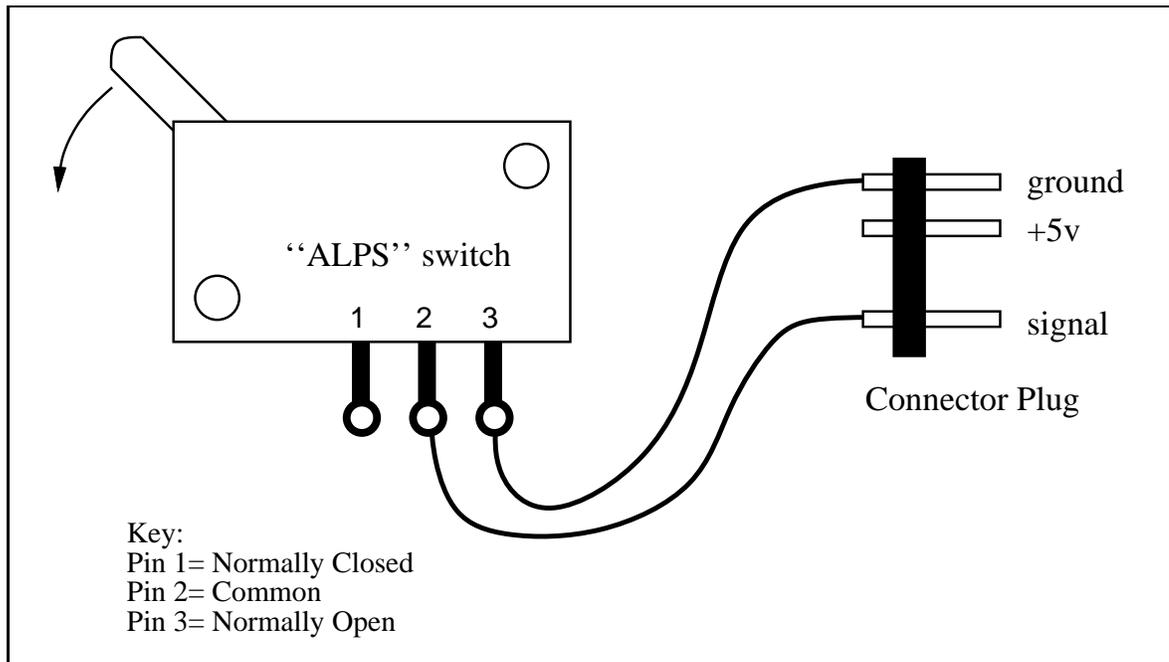
Figure 1.30: Step Four of Connector Wiring

- Hall effect (magnetic) sensor

In most cases, some discussion of the sensor's principle of operation accompanies the wiring diagram. Further information on the use of all sensors is elaborated in Chapter 5 on robotic sensors.

### 1.9.1 Touch Sensors

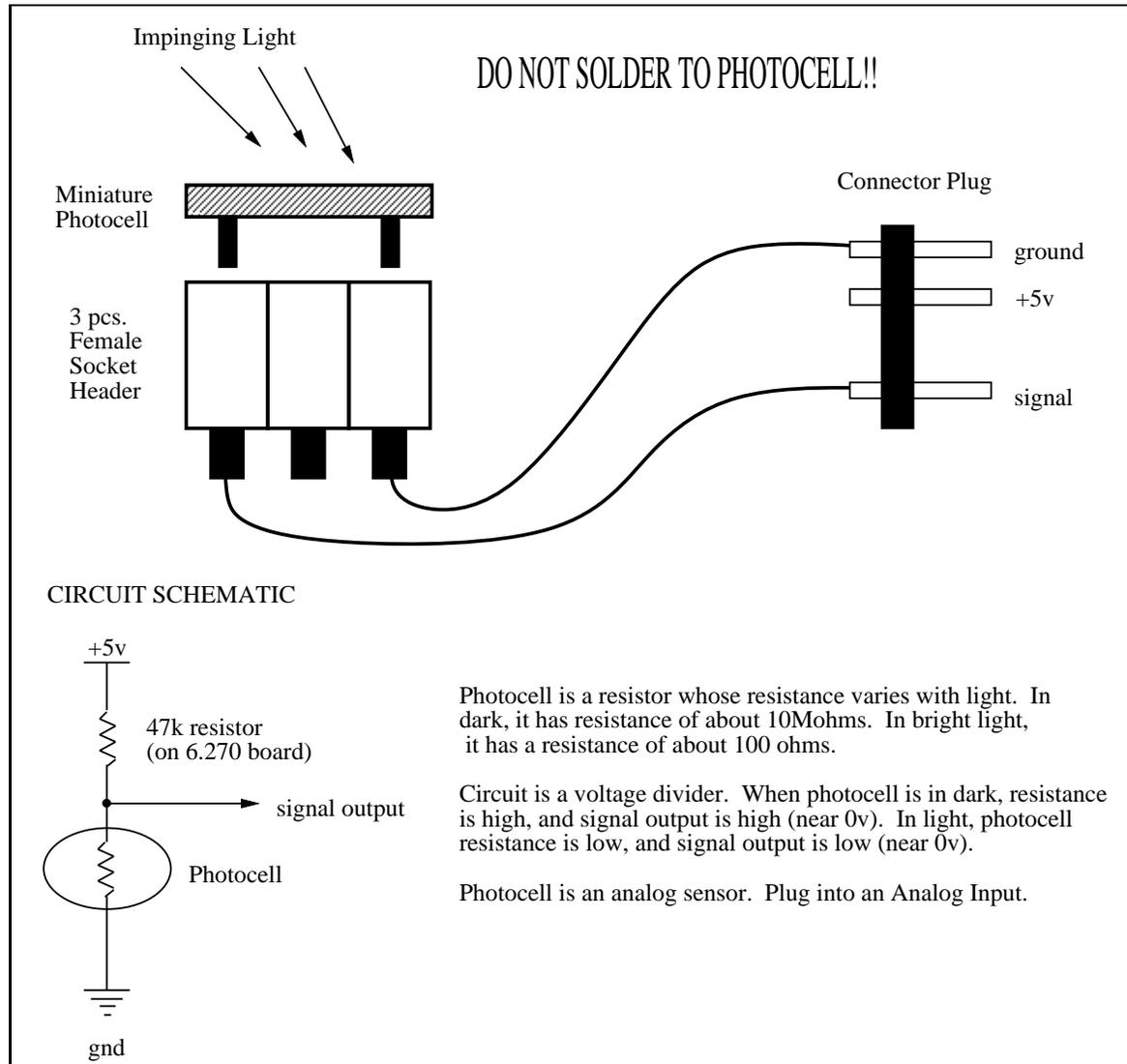
Several switches in the 6.270 kit may be used to make touch sensors. Diagrams for the white “ALPS” switch and the small black button switch are shown.



Touch switches should be wired in a normally open configuration, so that the signal line is brought to ground only when the switch is depressed.

## 1.9.2 Photocell Light Sensor

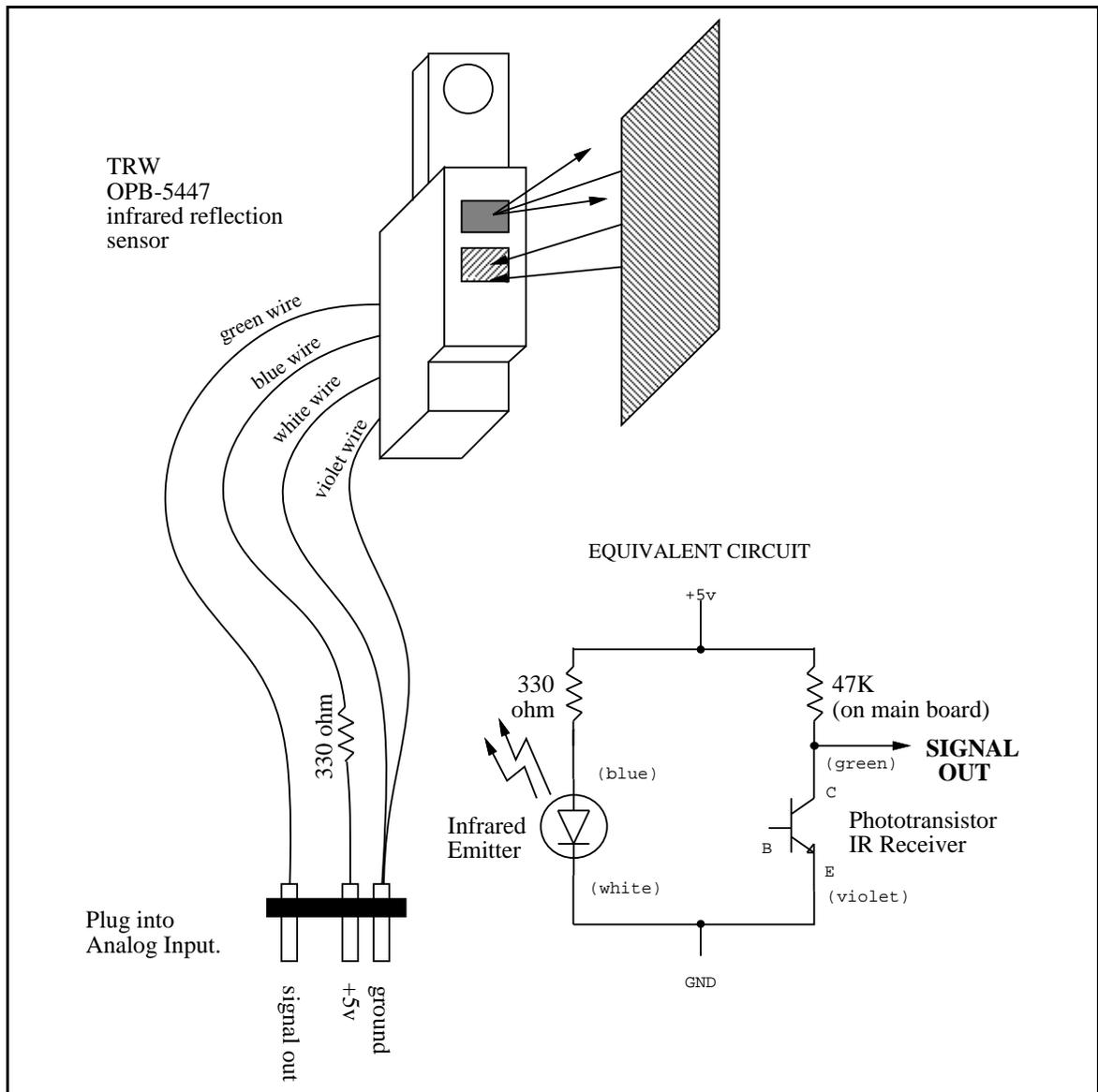
The photocell is a special type of resistor which responds to light. The more light hitting the photocell, the lower its the resistance.



The output signal of the photocell is an analog voltage corresponding to the amount of light hitting the cell. Higher values correspond to less light.

### 1.9.3 Infrared Reflectance Sensor

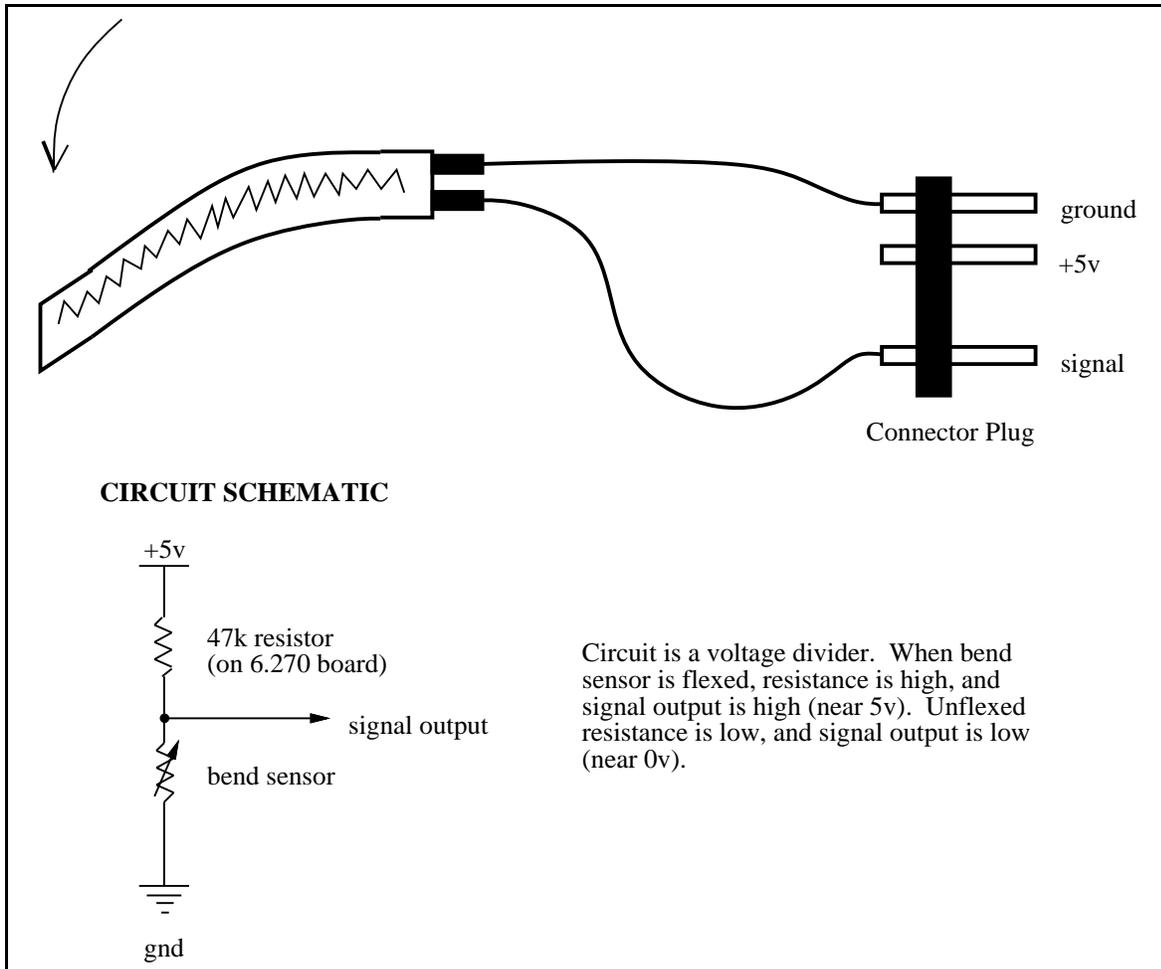
The TRW model OPB-5447 is a matched pair infrared transmitter and infrared receiver. It works by measuring the amount of light that is reflected into the receiver. Because the receiver also responds to ambient light, the device works best when well shielded from ambient light.



The output signal of the phototransistor receiver is an analog voltage corresponding to the amount of light hitting the phototransistor. Higher values correspond to less light, and hence a smaller degree of surface reflectivity.

### 1.9.4 Single Bend Sensor

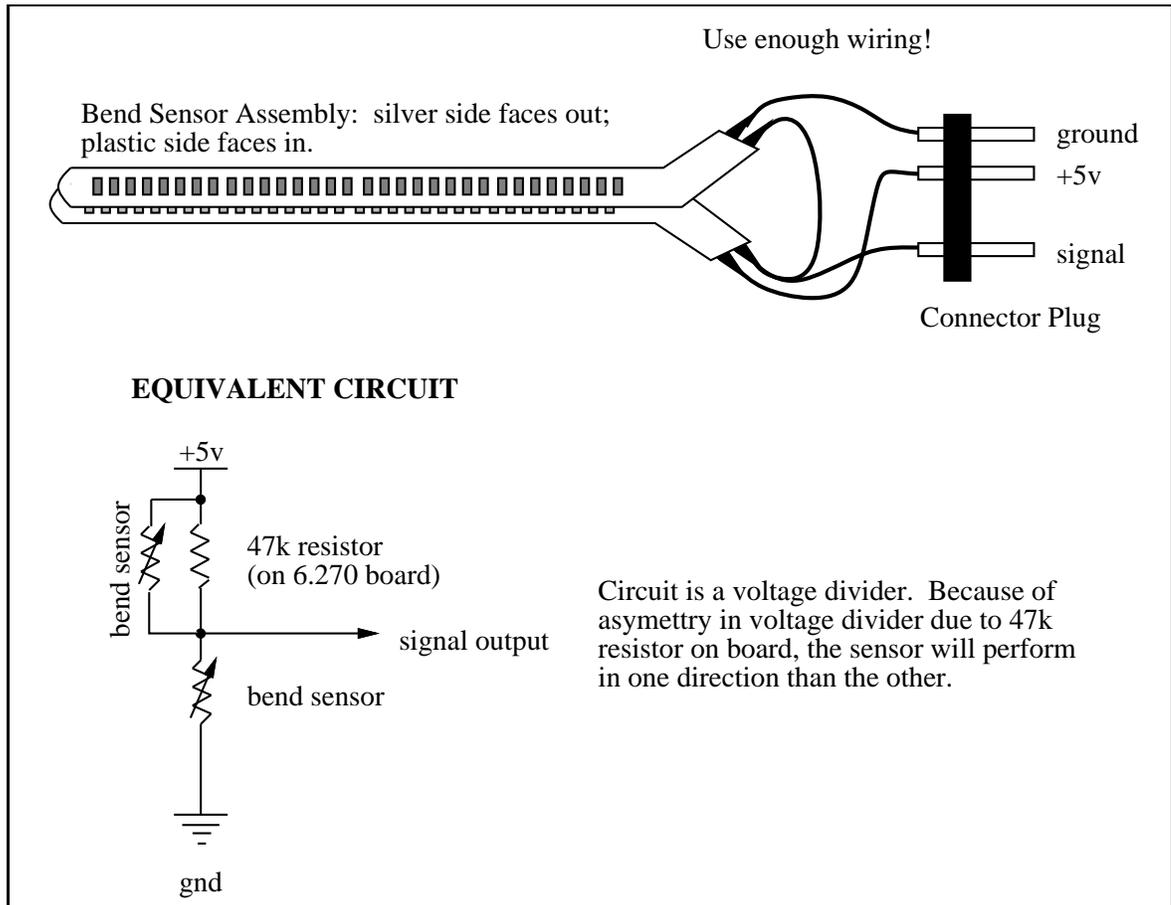
The bend sensor element is a resistive device that changes in resistance when it is deformed. The sensors are only sensitive to being bent in one direction—the one that stretches the silver material. Bending them in this direction increases their resistance. Bending them in the opposite direction does not change the resistance.



The sensors have a relaxed resistance of about  $50\text{k}\Omega$  and a fully bent resistance of about  $300\text{k}\Omega$ .

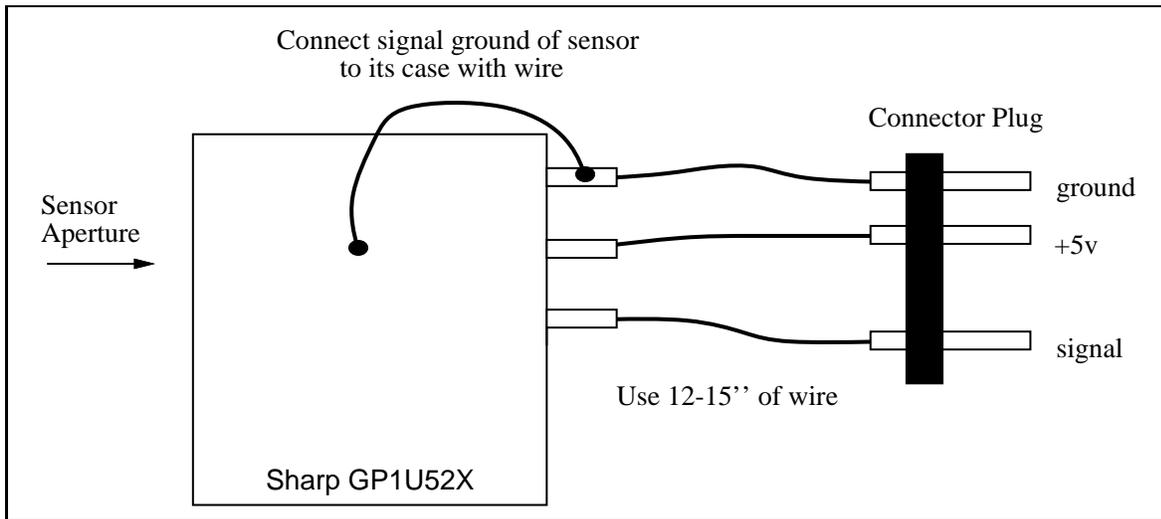
### 1.9.5 Dual Bend Sensor

By using two bend sensor strips that are mounted back-to-back, a sensor assembly can be created that is sensitive to bending in either direction.



### 1.9.6 Sharp Infrared Sensor

The Sharp GP1U52X sensor detects infrared light that is modulated (e.g., blinking on and off) at 40,000 Hz. It has an active low digital output, meaning that when it detects the infrared light, its output is zero volts.

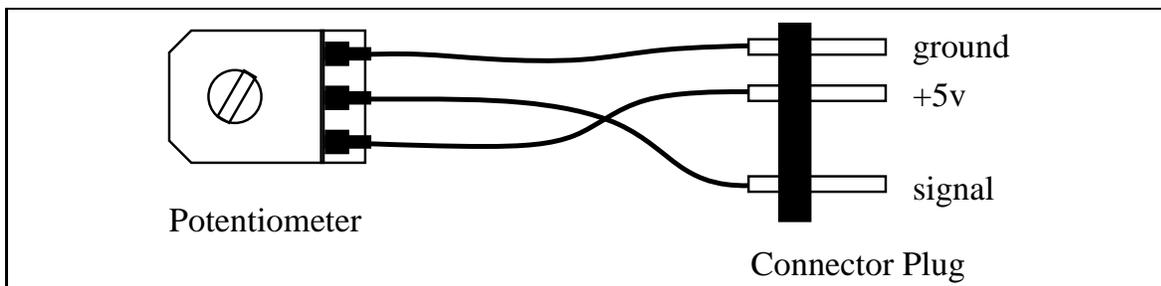


The metal case of the sensor must be wired to circuit ground, as indicated in the diagram. This makes the metal case act as a Faraday cage, protecting the sensor from electromagnetic noise.

Use of the Sharp sensor is discussed in Section 7.8.2. An explanation of how the infrared system works is given in Section B.7.

### 1.9.7 Potentiometer

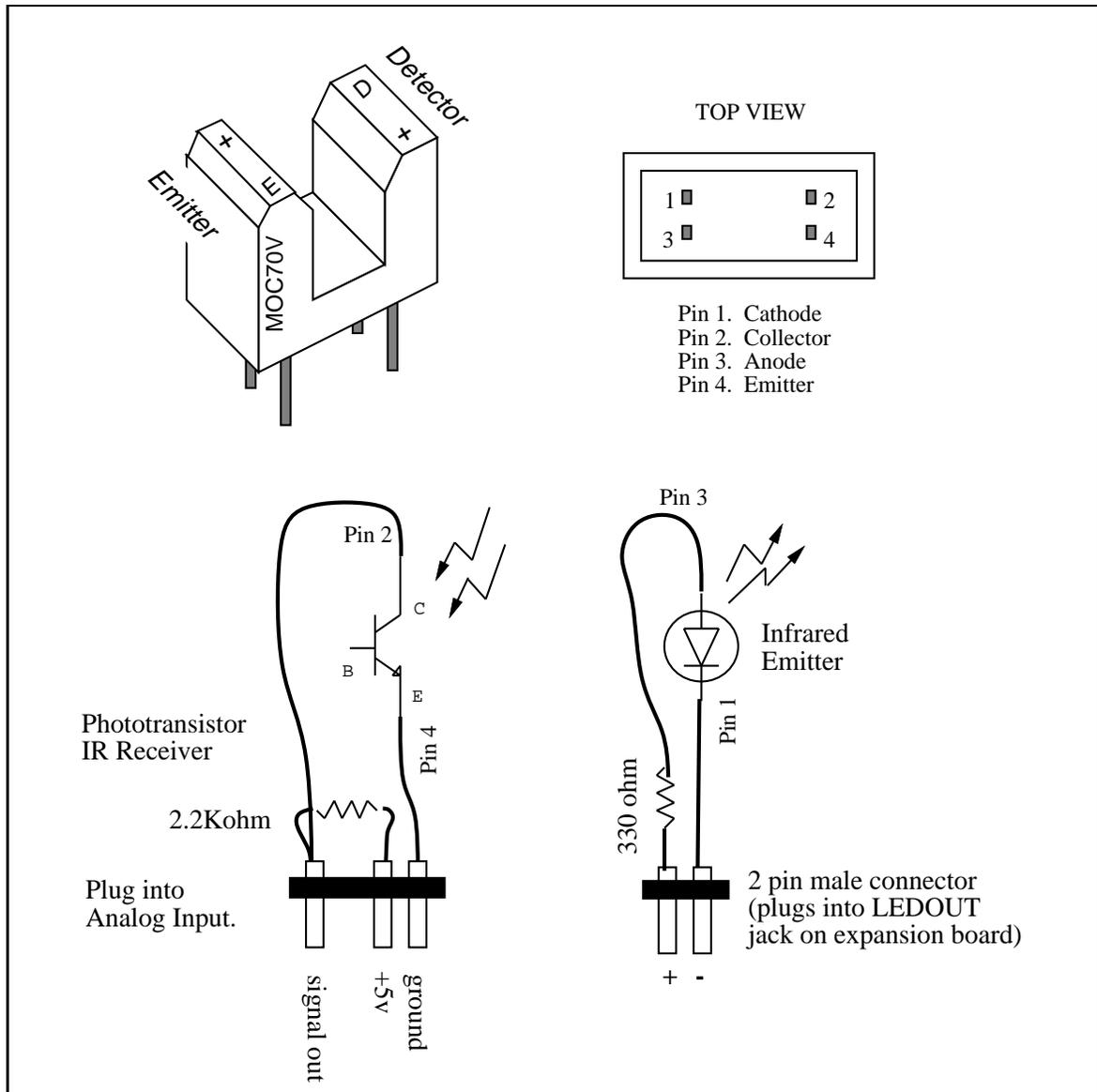
The potentiometer can be used as a rudimentary rotary position sensor.



The linear potentiometer may also be wired in this fashion.

### 1.9.8 Slotted Optical Switch

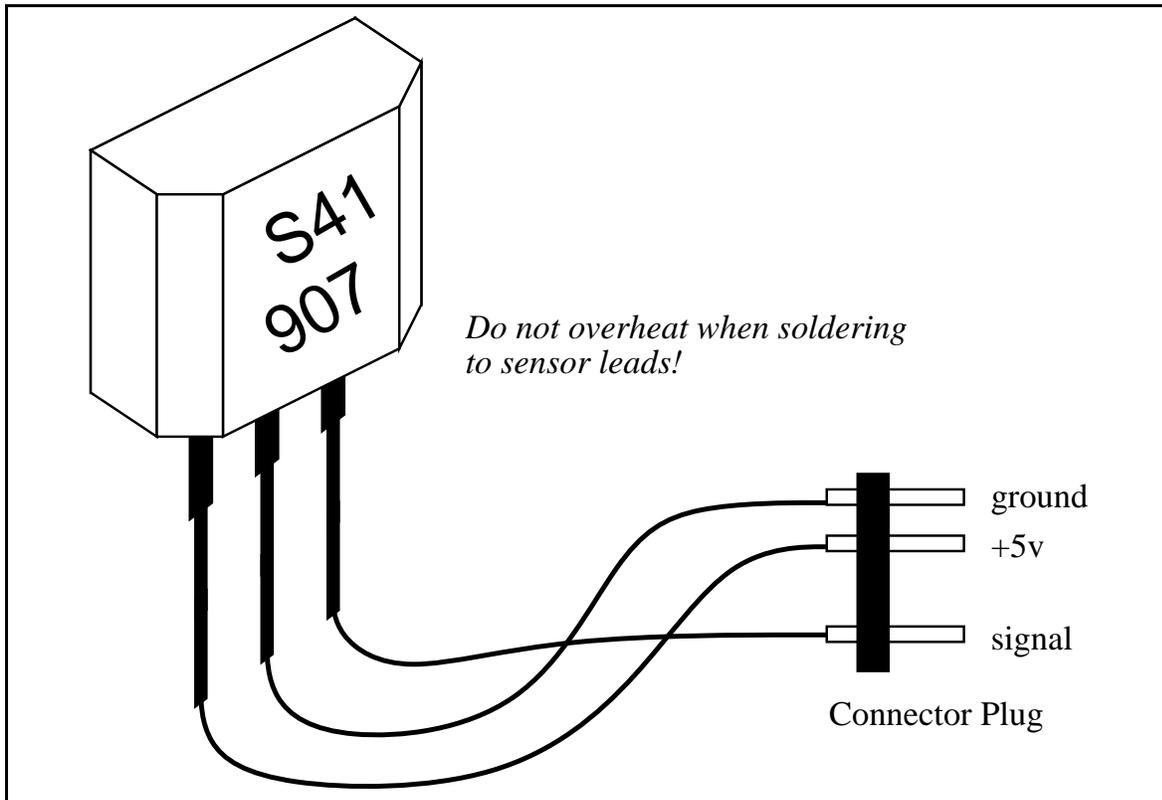
The Motorola model MOC70V is a matched pair infrared transmitter and infrared receiver. It works by measuring the amount of light transmitted from the transmitter to the receiver. Best results are achieved when ambient light is shielded from the device.



The output signal of phototransistor element is an analog voltage that corresponds to the amount of light hitting the phototransistor. Higher values indicate less light. This output signal can be sampled by software to count holes in a wheel rotating through the slot.

### 1.9.9 Hall Effect Sensor

The SS41 series Hall effect sensor detects the presence of a magnetic field. After it latches a south pole field (positive magnetic gauss) it produces a zero volt output. It then requires a north pole field to be reset.



The device can be easily damaged during soldering. Be careful not to apply too much heat or stress the device leads.

## 1.10 Motor Wiring

This section explains how to wire the Polaroid motors and the servo motor, and how to prepare the Polaroid motor for mounting on a LEGO device.

### 1.10.1 The Polaroid Motor

The Polaroid motors are used to eject film in their instant cameras and are particularly powerful DC motors. They are manufactured by Mabuchi, a leading Japanese motor manufacturer. The Polaroid motors have been donated to the 6.270 course by Polaroid.

The process of preparing the motor can be broken into three separate parts. The first part is to place a LEGO gear on the shaft of the motor. The second part is to mount the motor onto a platform. The third part is wiring a cable and plug to the motor assembly.

These instructions will specify that an eight-tooth LEGO gear be permanently installed on the motor shaft. In general, this is the most useful motor configuration. Other possibilities include mounting a LEGO pulley wheel or a larger diameter gear on the shaft.

It is recommended that two motors with the eight-tooth gear mount be built and evaluated. Later, it can be decided if the remaining motors should be build differently. Only for unusual applications will another configuration be preferable.

#### Attaching a Gear to the Polaroid Motor

- The motors come with a metal gear that is press-fit onto the shaft of the motor. The first step is to remove this gear.

The gear is removed using a pair of wire strippers. Place the jaws of the strippers between the motor and the gear. When the strippers are closed, the bevel in the cutters should pry off the gear.

The cutters should provide a uniform force around the gear so that it does not get stuck on the shaft when being pried off.

- Cut a piece of thin plastic tubing to the size of the length of one 8-tooth gear and place it on the shaft. Make sure that the tubing is not rubbing against the motor housing when the shaft turns. This could cause a slight decrease in the performance of the motor. The LEGO gear should be able to fit snugly over the shaft and the tubing.
- Place a drop of super glue around the outer area of the tubing that is furthest away from the motor housing as shown in Figure 1.31. *Make sure that too much*

*glue is not used. If there is too much super glue, it may leak into the motor housing and jam up the motor.* Using a paper napkin, pat off any of the excess super glue.

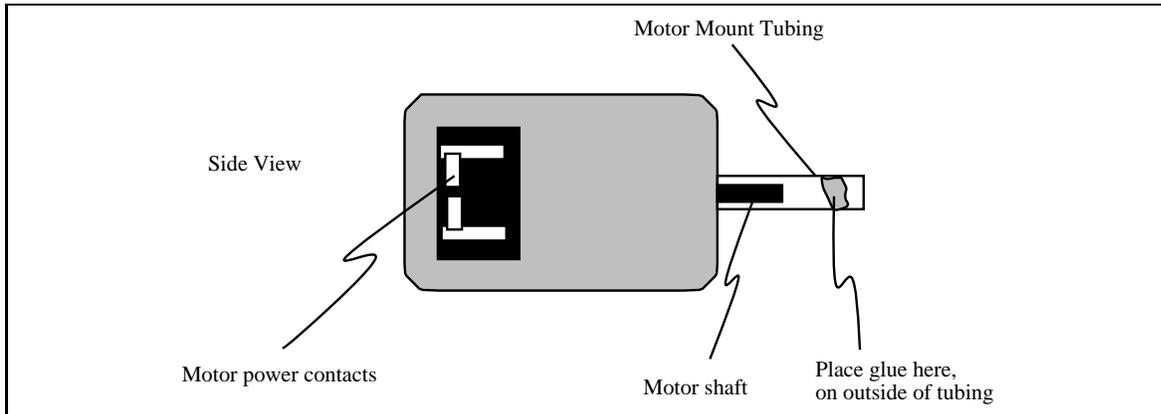


Figure 1.31: Motor Housing with Tubing

- Get an 8-tooth LEGO gear. It has two inside surfaces: a beveled surface and a flat surface. The gear will slide on easier when the beveled side is placed on first. Push the gear in all the way. It should fit without slippage. The gear should not be wobbly.
- Using a hot glue gun, glue the outer edge of the tubing to the gear by placing the glue into the inside of the gear.
- After the glue has dried, remove the gear from the shaft. The tubing should come off with it and should be inside the gear. Cut off any excess tubing (tubing that is longer than the gear) from the side that mounts onto the motor.
- *This is the most crucial step.* Place  $\frac{1}{2}$  drop to 1 drop of super glue into the center of the tubing and push the gear back onto the shaft. Too much glue can ruin the motor. Place the motor with the gear facing down to let the excess super glue dry away from the motor housing.

### Attaching the Polaroid Motor to a LEGO Base

The purpose of this step is to permanently affix the motor to LEGO parts so that it will mesh properly with gear mechanisms built from other LEGO pieces.

To make sure that the motor is mounted properly, it will be placed on a platform in the correct orientation to mesh with other LEGO gears.

This platform or jig is shown in Figure 1.32. It is constructed from two  $2 \times 8$  beams, one  $6 \times 8$  flat plate, one  $2 \times 4$  plate, two 24-tooth gears, and two axes.

The motor is placed on a 2×4 flat plate and mounted so that its 8-tooth gear is nestled between the two 24-tooth gears at the proper horizontal and vertical LEGO spacing.

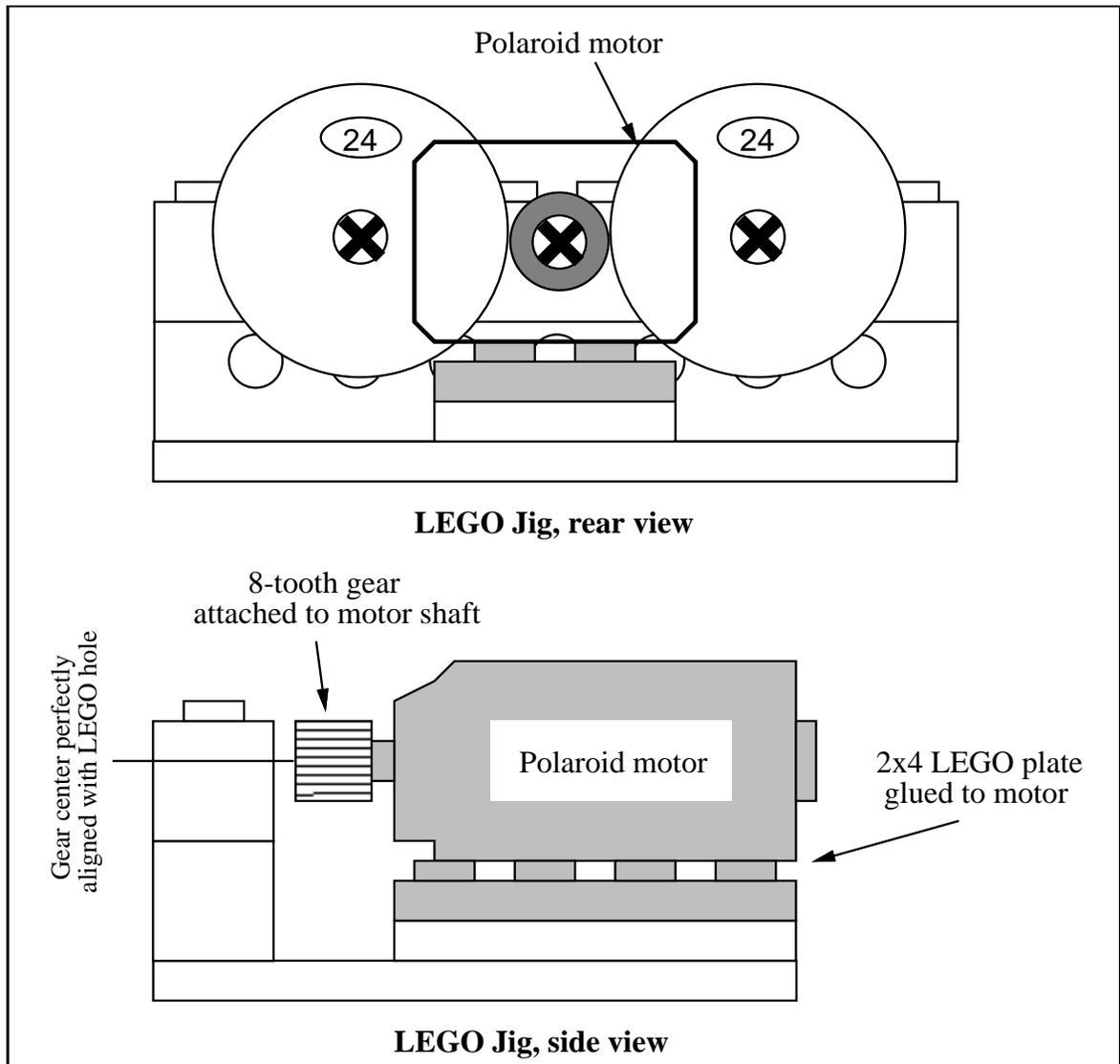


Figure 1.32: LEGO Jig for Mounting Polaroid Motor

- Assemble the jig as shown in Figure 1.32. A second 2×4 plate will be mounted to the motor.
- Mix some epoxy and place it on the top connectors of the base plate. Put enough to hold the motor.

- One face of the motor has two notched openings. Place the motor on the 2×4 plate such that the notches are facing up. This is to prevent the epoxy from getting into the motor.
- Position the motor on its plate so that the 8-tooth gear is meshed between the two 24-tooth gears, and the center line of the motor shaft is parallel with the axles of the 24-tooth gears.
- Wait five to fifteen minutes for the epoxy to harden.

### Wiring a Cable and Plug to the Polaroid Motor

- Motor cables may be constructed with either two strands of ribbon cable wire or the twisted pair red/black cable. Cut an 8 inch to 12 inch length of whichever wire is preferable.
- Strip and tin both ends of the wire.
- On the side of the motor there should be two metal lead/pads. Solder one wire lead to each pad. After proper soldering, hot glue may be used to hold the wire to the side of the motor for a stress relief.
- Motor plugs may be wired for bidirectional or unidirectional use, as shown in Figure 1.26. (For most purposes, motors will need to be operated bidirectionally.)

Cut a two- or three-long strip of male socket headers as will be needed.

- Using the connector plug wiring technique shown in Figure 1.27 through Figure 1.30, wire the motor plug. Polarity does not matter since the plug may be inserted into a motor power jack in either orientation.

### 1.10.2 Servo Motor

Figure 1.33 illustrates a typical servo motor similar to the one provided in the 6.270 kit. The servo motor has a short cable that terminates in a three-lead connector, as illustrated. The functions of these lead are power, ground, and the control signal.

Figure 1.34 illustrates the cable that should be built to interface the servo motor to the 6.270 board. As indicated in the diagram, the servo cable plugs into the PWR OUT port located on the Microprocessor Board, and the PORT D2 signal located on either the Microprocessor Board or the Expansion Board.

The adapter cable should be made as long as needed.

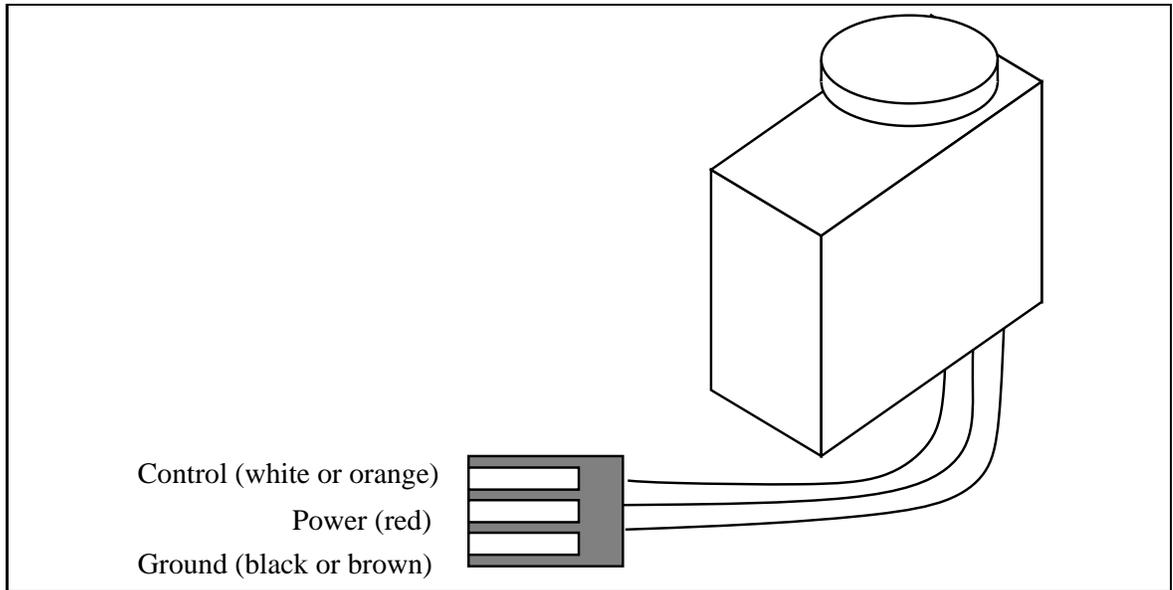


Figure 1.33: Servo Motor and Integral Connector Plug

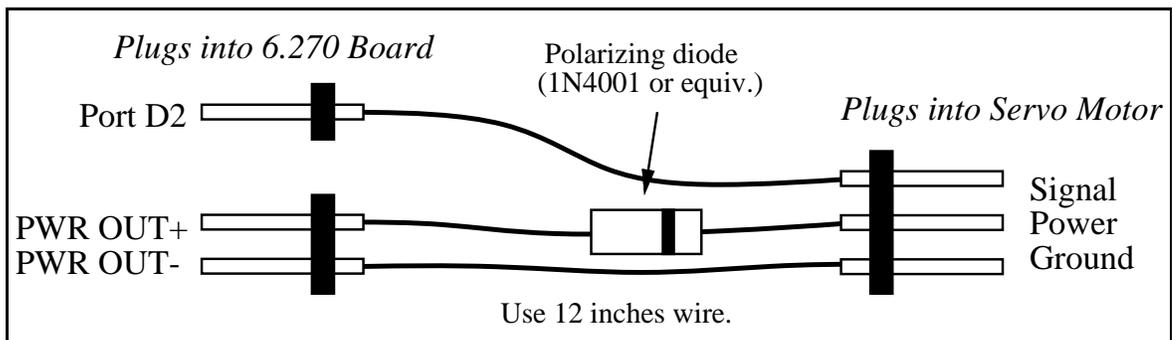


Figure 1.34: Servo Motor Cable Wiring Diagram

## 1.11 Computer Cable Wiring

The double-headed modular plug cable supplied in the 6.270 kit is used to connect the board with VAXstation 3100 and DECstation 3100 computers. This cable has a normal modular plug on one end (this end plugs into the board) and a off-center-keyed plug on the other end (this end plugs into the computer).

This section provides directions for building a cable to interface the 6.270 board with a VAXstation 2000 computer. The following parts are used:

- length of modular cord with plug attached to one end
- DB-9 female computer connector
- DB-9 connector housing

When building the cable adapter, be careful to follow the wiring directions precisely. Refer to Figure 1.35 as needed.

- Begin by cutting the spade lug connectors from the modular plug cable. Strip a bit of insulation from the ends of the four wires and tin the wire ends.

The cable provided in the kit is about 15 feet long. It may be cut to a shorter length if desired.

- Thread the wire ends through the metal housing as indicated in Figure 1.35.
- Tie a knot in the end of the cable about one to two inches from the free wire ends. This knot will act as a stress relief when the cable is pulled.
- Connect the wire leads to the three terminals of the DB-9 connector as indicated in the diagram.

*Note:* Make sure that the wiring of the modular cable you are using matches the diagram; e.g., that the black wire is on the left when looking down on the modular plug as in the diagram. *If the black wire is on the right, reverse the wiring of the black and yellow connections.*

- Install the connector assembly in the plastic plug housing (this is not pictured in the diagram). The two screw handles must be installed before the housing is snapped together.
- Snap the housing together and the job is done. The one-inch long aluminum tube may be discarded; it may not be used as a kit part.

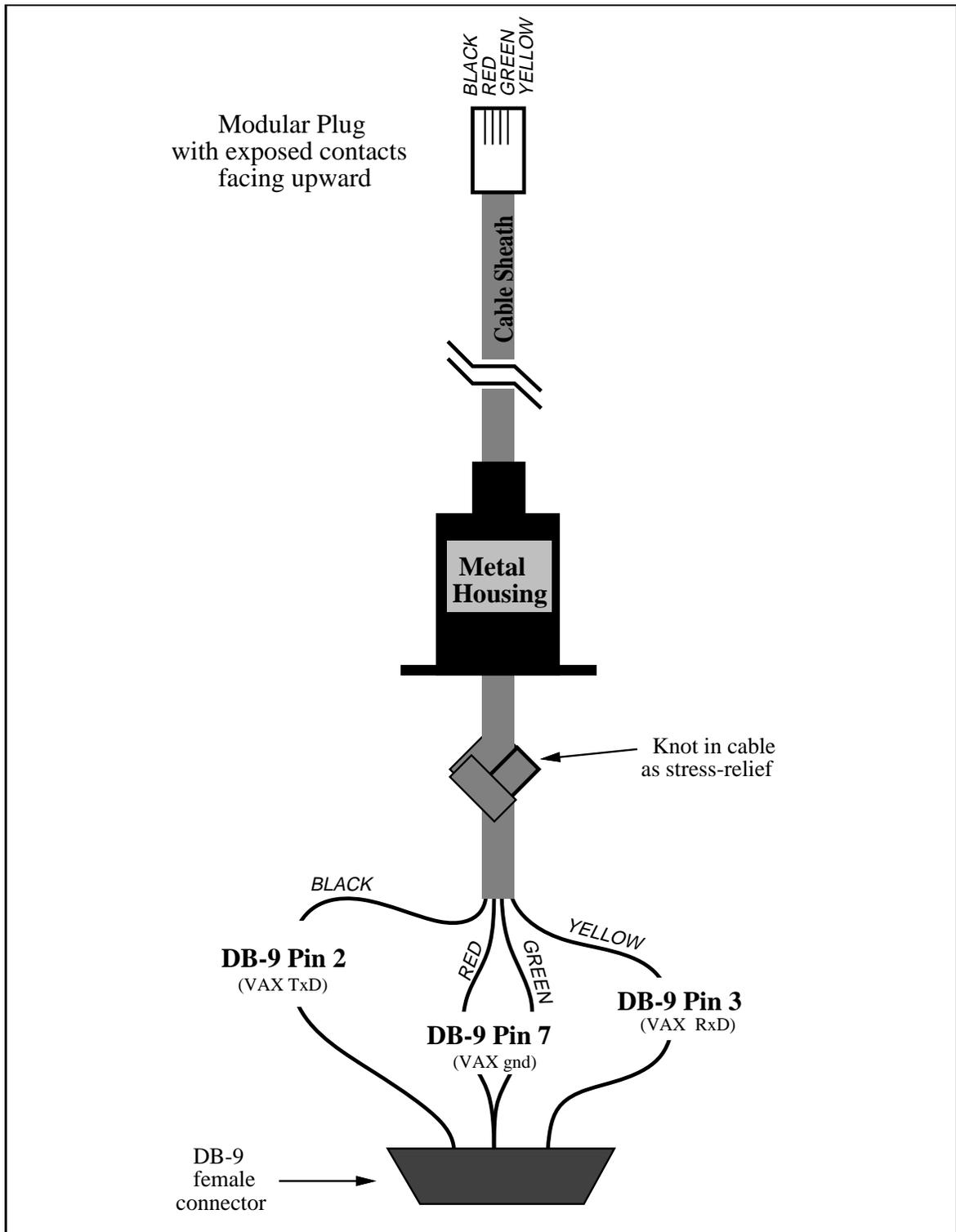


Figure 1.35: VAXstation 2000 Computer Cable Wiring Diagram



# Chapter 2

## Motors

This chapter introduces several types of motors commonly used in robotic and related applications.

**DC motors** are inexpensive, small, and powerful motors that are used widely. Geartrain reductions are typically needed to reduce the speed and increase the torque output of the motor.

**Stepper motors** are not part of the 6.270 kit but commonly used in robotics, particular in mechanisms that perform linear positioning, such as floppy and hard disk drive head motors and X-Y tables.

**Servo motors** are used in radio control airplanes to control the position of wing flaps and similar devices. A servo motor includes a built-in geartrain and is capable of delivering high torques directly. The output shaft of a servo does not rotate freely as do the shafts of DC motors and stepper motors, but rather is made to seek a particular angular position under electronic control.

### 2.1 DC Motors

DC motors are widely used in robotics for their small size and high energy output. They are excellent for powering the drive wheels of a mobile robot as well as powering other mechanical assemblies.

#### 2.1.1 Ratings and Specifications

Several characteristics are important in selecting a DC motor. The first two are its input ratings that specify the electrical requirements of the motor.

**Operating Voltage.** If batteries are the source of power for the motor, low operating voltages are desirable because fewer cells would be needed to obtain the specified voltage. However, the electronics to drive motors are typically more efficient at higher voltages.

Typical DC motors may operate on as few as 1.5 volts on up to 100 volts. Roboticists often use motors that operate on 6, 12, or 24 volts.

**Operating Current.** Ideally one would like a motor that produces a great deal of power while requiring a minimum of current. Typically however the current rating (in conjunction with the voltage rating) is a good indication of the power output capacity of a motor.

Motors that draw more current will deliver more power. Also, a given motor draws more current as it delivers more output torque. Thus current ratings are often given when the motor is stalled. At this point it is drawing the maximal amount of current.

A low voltage (e.g., 12 volt or less) DC motor may draw from 100 milliamps to several amps at stall, depending on its design.

The next three ratings describe the motor's output characteristics:

**Speed.** Usually this is specified as the speed in rotations per minute (RPM) of the motor when it is unloaded, or running freely, at its specified operating voltage.

Typical DC motors run at speeds from several thousand to ten thousand RPM.

**Torque.** The torque of a motor is the rotary force produced on its output shaft.

When a motor is stalled it is producing the maximum amount of torque that it can produce. Hence the torque rating is usually taken when the motor has stalled and is called the *stall torque*.

The motor torque is measured in ounce-inches (in the English system). A rating of one ounce-inch means that the motor is exerting a tangential force of one ounce at a radius of one inch from the center of its shaft.

Torque ratings may vary from less than one ounce-inch to several dozen ounce-inches for large motors.

**Power.** The power of a motor is the product of its speed and torque. The power output is greatest somewhere between the unloaded speed (maximum speed, no torque) and the stalled state (maximum torque, no speed).

Figure 2.1 lists some specifications of the Polaroid motor provided in the 6.270 kit (the speed and torque ratings were subjectively determined through comparisons with similar sized DC motors and could stand closer measurement).

Specification	Rating	Comment
Voltage	5 volts	normal low voltage
Current	up to 4 amps	high current capacity
Speed	4000-6000 RPM	slightly slow (?)
Torque	unknown	unusually high

Figure 2.1: Polaroid Motor Specifications

The motor is used to eject film in Polaroid instant cameras. For this application, the fact that it is low voltage (5 volts) is very important so that only a few cells are needed to run the motor. The motor is used with a several stage geartrain to reduce its speed and generate the torque need to eject the film. The fact that it is high torque is very desirable.

For an application in powering a mobile robot, the motor is very suitable. Again the low voltage is desirable, as is the high torque output. Probably the only undesirable characteristic is the high current draw; however, this is the only way to achieve the high torque at low voltages.

### 2.1.2 Measuring Motor Torque

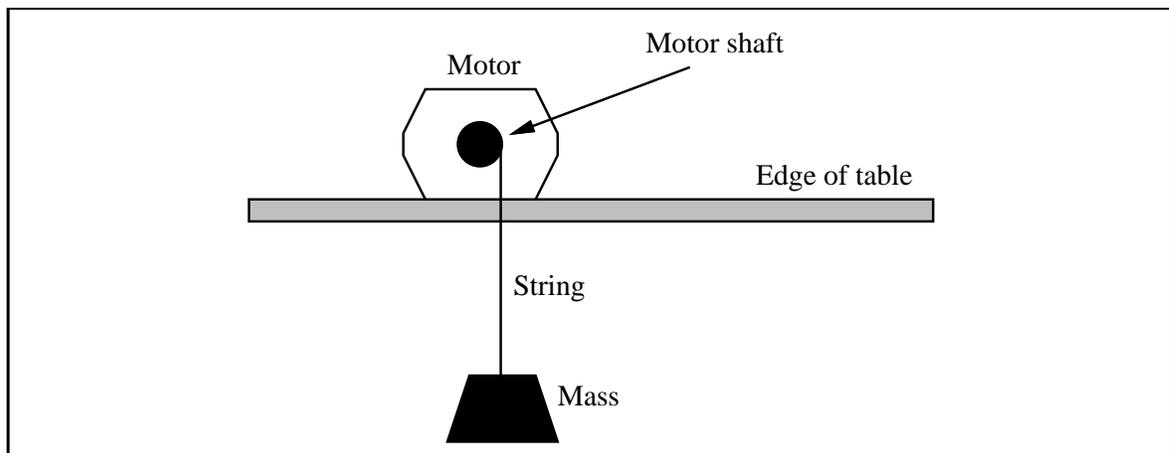


Figure 2.2: Experiment to Measure Motor Torque

A simple experiment can be performed to accurately determine the torque rating of a motor. All that is needed is a motor to be measured, a power supply for the motor, a piece of thread, a mass of known weight, a table, and a ruler.

The mass is attached to one end of the thread. The other end of the thread is attached to the motor shaft so that when the motor turns the thread will be wound

around the motor shaft. The motor shaft must be long enough to wind the thread like a bobbin.

The motor is put near the edge of a table with the mass hanging over the edge, as illustrated in Figure 2.2. When the motor is powered on, it will begin winding up the thread and lifting the mass. At first this will be an easy task because the moment arm required to lift the mass is small—the radius of the motor shaft.

But soon, the thread will wind around the shaft, increasing the radius at which the force is applied to lift the mass. Eventually, the motor will stall. At this point, the radius of the thread bobbin should be measured. The torque rating of the motor is this radius per amount of mass that was caused the stall.

### 2.1.3 Speed, Torque, and Gear Reduction

It was mentioned earlier that the power delivered by a motor is the product of its speed and the torque at which the speed is applied. If one measures this power over the full range of operating speeds—from unloaded full throttle to stall—one gets a bell-shaped curve of motor power output.

When unloaded, the motor is running at full speed, but at zero torque, thus producing zero power. Conversely, when stalled, the motor is producing its maximum torque output, but at zero speed—also producing zero power! Hence the maximum power output must lie somewhere in between.

A typical DC motor operates at speeds that are far too high to be useful, and torques that are far too low. *Gear reduction* is the standard method by which a motor is made useful.

Using gear reduction, the motor shaft is fitted with a gear of small radius that meshes with a gear of large radius. The motor's gear must revolve several times in order to cause the large gear to revolve once (see Figure 4.7). It is evident that the speed of rotation is decreased, but, overall power is preserved (excepting losses due to friction) and therefore the torque must increase.

By ganging together several stages of this gear reduction, an immensely strong torque can be produced at the final stage.

The challenge when designing a high-performance gear reduction for a competitive robot is to determine the amount of reduction that will allow the motor to operate at highest efficiency.

If the normal operating point of a motor/geartrain assembly is faster than the peak efficiency point, the geartrain will be able to accelerate quickly, but will not be operating at peak efficiency once it has reached the maximum velocity.

Depending on the mass of the robot and the performance desired, different gear ratios might be appropriate. Experimentation is probably the best way to choose the best geartrain.

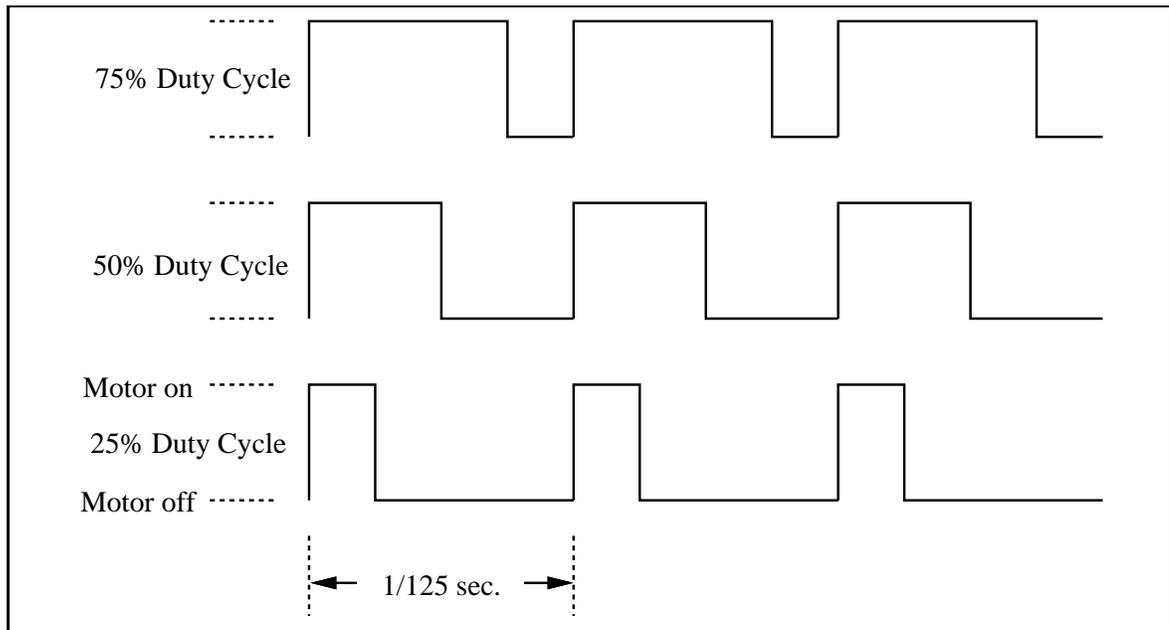


Figure 2.3: Example of Several Pulse Width Modulation Waveforms

### 2.1.4 Pulse Width Modulation

Pulse width modulation is a technique for reducing the amount of power delivered to a DC motor. This is typically used in mechanical systems that will not need to be operated at full power all of the time. For a 6.270 robot, this would often be a system other than the main drivetrain.

Instead of reducing the voltage operating the motor (which would reduce its power), the motor's power supply is rapidly switched on and off. The percentage of time that the power is on determines the percentage of full operating power that is accomplished.

Figure 2.3 illustrates this concept, showing pulse width modulation signals to operate a motor at 75%, 50%, and 25% of the full power potential.

A wide range of frequencies can be used for the pulse width modulation signal. 6.270 system software used to control the motors operates at 1000 Hertz.

A PWM waveform consisting of eight bits, each of which may be on or off, is repetitiously used to control the motor. Every  $\frac{1}{1000}$  of a second, a control bit determines whether the motor is enabled or disabled. Every  $\frac{1}{125}$  of second the waveform is repeated.

Because one to eight bits may be set in the PWM waveform, the motors may be adjusted to eight power levels between off and full on.

## 2.2 Stepper Motors

Stepper motors have several electromagnetic coils that must be powered sequentially to make the motor turn. By reversing the order that the coils are powered, a stepper motor can be made to reverse direction. The rate at which the coils are respectively energized determines the velocity of the motor up to a physical limit.

Typical stepper motors have two or four coils. The shaft of a stepper motor moves between discrete rotary positions that correspond to the particular coil that was last energized. Because of this precise position controllability, stepper motors are excellent for applications that require high positioning accuracy.

Stepper motors are used in X-Y scanners, plotters, and machine tools, floppy and hard disk drive head positioning, computer printer head positioning, and numerous other applications.

Unfortunately, the 1992 6.270 kit does not include a servo motor.

## 2.3 Servo Motors

Servo motors incorporate several components into one device package:

- a small DC motor;
- a gear reduction drive for torque increase;
- an electronic shaft position sensing and control circuit.

The output shaft of a servo motor does not rotate freely, but rather is commanded to move to a particular angular position. The electronic sensing and control circuitry—the servo feedback control loop—drives the motor to move the shaft to the commanded position. If the position is outside the range of movement of the shaft, or if the resisting torque on the shaft is too great, the motor will continue trying to attain the commanded position.

Servo motors are used in model radio control airplanes and helicopters to control the position of wing flaps and other flight control mechanisms. They also have been used to drive the legs of Genghis, the MIT A.I. Laboratory's walking robot.

The gear reduction unit incorporated into most servo motors is quite powerful. The servo motor provided in the 6.270 kit delivers approximately 50 ounce-inches of torque.

### 2.3.1 Control

A servo motor has three wires: power, ground, and control. The power and ground wires are simply connected to a power supply. Most servo motors operate from five volts.

The control signal consists of a series of pulses that indicate the desired position of the shaft. Each pulse represents one position command. The length of a pulse in time corresponds to the angular position.

Typical pulse times range from 0.7 to 2.0 milliseconds for the full range of travel of a servo shaft. Most servo shafts have a 180 degree range of rotation. The control pulse must repeat every 20 milliseconds.

### **2.3.2 Application**

For 6.270 purposes, servo motors would be excellent for operating a rotating sensor platform. A 1:2 gear-up from the servo motor to the platform could be used to yield a full 360 degrees of rotation. Because the servo includes position sensing circuitry, an external sensor to measure the position of the sensor platform would not be needed.

Servo motors would also be excellent for meshing with a gear rack, accomplishing highly controllable rectilinear motion.



# Chapter 3

## Batteries

Robots may be powered by a variety of methods. Some large robots use internal combustion engines to generate electricity or power hydraulic or pneumatic actuators.

For a small robot, however, battery power offers a number of advantages over any other method. Batteries are cheap, relatively safe, small, and easy to use. Also, motors convert electrical power into mechanical power with relative efficiency.

There are many different types of batteries, each with its own tradeoffs. This chapter introduces a variety of batteries, explains standard ways of rating batteries, and discusses the design of the 6.270 battery charger.

### 3.1 Cell Characteristics

Two terms that are often used interchangeably, but actually have a different meaning, are the words *battery* and *cell*. Technically, a cell is the unit that houses a single chemical reaction to produce electricity. A battery is a bank of cells.

#### 3.1.1 Voltage

Cells use chemical reactions to produce electricity. Depending on what materials are used to create the reaction, a different voltage will be produced. This voltage is called the *nominal cell voltage* and is different for different battery technologies.

For example, a standard flashlight cell uses a carbon-zinc reaction and has a cell voltage of 1.5 volts. Car batteries have six lead-acid cells, each with a cell voltage of 2.0 volts (yielding the 12 volt battery).

#### 3.1.2 Capacity

In general, the larger a cell is, the more electricity it can supply. This *cell capacity* is measured in *ampere-hours*, which are the number of hours that the cell can supply a

certain amount of current before its voltage drops below a predetermined threshold value.

For example, 9 volt alkaline batteries (which consist internally of six 1.5 volt alkaline cells) are generally rated at about 1 ampere hour. This means that the battery can continuously supply one ampere of current for one hour before “dying.” In the capacity measurement, the 9 volt alkaline battery “dies” when the battery voltage drops below 5.4 volts.

However, the amp-hour measurement is usually taken to assume a twenty hour discharge time. Then the 9 volt battery would need to be tested by having it supply 1/20th of its rated capacity—this would be 50 milliamps—for twenty hours. If it were drained more quickly, as in the one-hour test, the capacity would turn out to be quite a bit less.

### 3.1.3 Power Density

There are large differences in capacity per unit weight—the cell’s *power density*—across battery types. This is a of the cell’s most important rating.

Inexpensive carbon-zinc cells have the lowest power density of all cell types. Alkaline cells have about ten times the power density of carbon-zinc cells. Nickel-cadmium cells have less power density than alkalines, but they are rechargeable.

### 3.1.4 Discharge Curve

When a cell discharges, its voltage lessens over the course of the cell life. The characteristic discharge curve varies considerably over different types of cell.

For example, alkaline cells have a fairly linear drop from full cell voltage to zero volts. This makes it easy to tell when the cell is weakening.

Nickel cadmium cells have a linear voltage drop region that then drops off sharply at some point. For this reason, when consumer products use nickel cadmium cells, the device will suddenly “die” with no warning from the cells. One minute, they are fine, the next, they are dead. For a ni-cad cell, this is normal, but it can be annoying.

### 3.1.5 Internal Resistance

A cell can be modelled as a perfect voltage source in series with a resistor. When current is drawn out of the cell, its output voltage drops as voltage is lost across the resistor.

This cell characteristic, called the *internal resistance*, is important because it determines the maximum rate at which power can be drawn out of the cell.

For example, lead acid cells have very low internal resistance. This makes them well suited for the application of being a car battery, because huge amounts of current can be drawn from the cells to operate the car's starter motor.

Another example comes from a consumer photography flash. During the recycle time of a standard flash unit, the flash's cells are supplying charge as quickly as they can. The rate is limited largely by the cells' internal resistance. Alkaline cells have higher internal resistance than nickel-cadmium cells. Thus, the flash unit takes longer to recycle when alkaline cells are used.

Cells that have low internal resistance, in particular, lead acid and nickel cadmium cells, can be dangerous to work with, because if the cell is shorted, huge currents can flow. These currents will heat the metal wire they are flowing through to very high temperatures, easily melting the insulation from them. The cells will also become very hot and potentially may explode.

For this reason it is very important not to short a lead acid or nickel cadmium cell. Alkaline cells and carbon zinc cells, with their high internal resistances, will still deliver quite a bit of current when shorted, but nowhere near the amounts of the other two types of cells.

### 3.1.6 Rechargeability

Another important characteristic of a cell is whether or not it is rechargeable, and if so, how many times. Because cells are quite toxic to the environment, use of rechargeable cells is an important issue.

Unfortunately, the cells with the highest power densities—alkaline and lithium—are not rechargeable. But advances in rechargeable technologies are catching up.

#### The Memory Effect

The term “memory effect” refers to a phenomenon observed in rechargeable nickel cadmium cells in which cells that are only partially discharged before being recharged have a tendency to “remember” the level of discharge, and, over time, only become usable to that discharge level.

There is disagreement amongst cell manufacturers as to whether or not this phenomenon actually exists, but most concur that nickel cadmium cells should be discharged fully before being recharged.

Some cell technologies, such as lead acid cells and the new nickel hydride, do not exhibit this effect. Lead acid cells typically last for several hundred cycles of full discharge, and a thousand cycles of partial discharge.

### 3.1.7 Cost

Last but not least is cost. It would be wonderful if the best cells did not cost substantially more than the cells with worst performance, but this is not the case.

For consumer purposes, it is generally agreed that nickel cadmium cells, which cost several times as much as alkaline cells, are much less expensive over the cells' lifetimes. Nickel cadmium cells can be recharged several hundred times while alkaline cells are disposed of after one use. On the other hand, nickel cadmium cells exhibit the "sudden death" property mentioned earlier.

Some new battery technologies, like the very high capacity, rechargeable nickel hydride cells, are very expensive, but offer twice the capacity of either lead acid or nickel cadmium cells.

Cell Type	Voltage	Power Density	Internal Resistance	Rechargeable	Cost
Carbon-Zinc	1.5 volts	low	high	no	low
Alkaline	1.5 volts	high	high	no	moderate
Lithium	1.5 volts	very high	low	no	high
Nickel-Cadmium	1.2 volts	moderate	low	yes	moderate
Lead-Acid	2.0 volts	moderate	low	yes	moderate
Nickel-Hydride	1.2 volts	high	low	yes	very high

Figure 3.1: Table of Cell Characteristics

Figure 3.1 summarizes the characteristics of commonly available cell technologies.

Probably the worst thing one can say about all types of battery is that "it doesn't last long enough." Unfortunately this is more or less true, but things in the battery technology field are improving. The advent of laptop computers and the need for convenient electric cars have created a real market need for improved batteries.

## 3.2 Battery Packs

There are two ways that cells may be combined to make batteries: series connections and parallel connections.

When cells are connected in series, their voltages add but their amp-hour capacity does not. Series batteries should be composed of cells of equal capacities.

When cells are connected in parallel, their voltages remain the same, but their capacities add.

### 3.3 6.270 Battery Charger

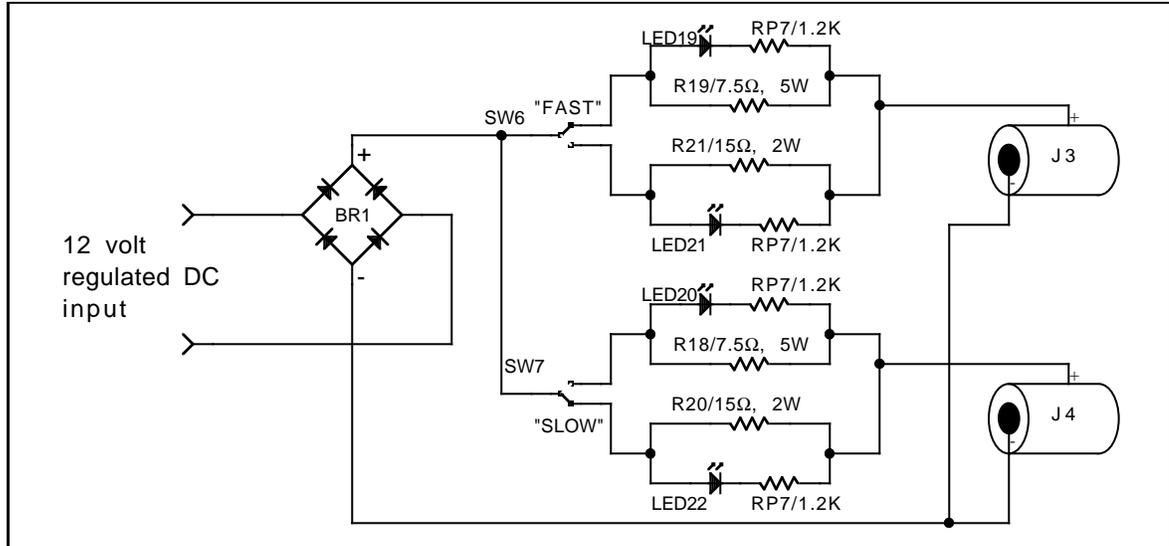


Figure 3.2: Battery Charger Schematic Diagram

The rule of thumb for charging batteries is to charge them at a rate equal to one-tenth of the amp-hour capacity of the battery. For example, if a battery is rated for 2.5 amp-hours (as are the Gates cells included in the 6.270 kit), then it would normally be charged at a rate of 250 milliamps.

Figure 3.2 shows the schematic diagram of the battery recharger. The essence of the charger is simply a resistor in series with the battery hooked up to a regulated voltage power supply.

The resistor limits the amount of current that can be delivered to the battery as a function of the battery voltage. Suppose that the battery is at its nominal 6 volt level. Then the voltage across the resistor is the voltage supply minus 6 volts. The current can be calculated as  $V/R$ , where  $V$  is the voltage drop and  $R$  is the resistor's value.

The 6.270 battery charger allows switching between two resistors for each of the two battery charge circuits. The 15Ω resistor limits current to about 250 to 300 milliamperes for a six volt battery. This is the normal charge rate. The 7.5Ω resistor limits current to about 500 to 600 milliamperes. This is a quick charge rate and should not be maintained after the battery is fully charged.

The resistors dissipate a fair bit of energy as heat and hence must be physically large. The amount of power dissipated is measured in watts and is calculated by the law  $W = V \times I$ , where  $V$  is voltage across the resistor and  $I$  is the current travelling through it. Since  $I = V/R$ , the power dissipation rate is  $W = V^2/R$ .

Assume a 4.8 volt drop across either resistor (12 volt supply minus 6 volt battery level minus 1.2 volts diode drop). For the  $7.5\Omega$  resistor, the power dissipation is then  $4.8^2/7.5$ , which is approximately 3 watts. A 5 watt resistor was selected for use so as to allow a margin of error and to provide better heat dissipation.

A similar calculation can be made for the  $15\Omega$  resistor, for which a 2 watt rating was chosen.

The status LEDs are lit by the voltage drop across the resistor in use.

The bridge rectifier acts to polarize the voltage input, so that either an AC or DC supply can be used. It also drops about 1.2 volts from the supply as per normal diode characteristics.

# Chapter 4

## LEGO Design

by Fred G. Martin and Randy Sargent

LEGO Technics are fun to play with and allow the construction of great things, but they are not always easy to use. In fact, it is often quite challenging to build a LEGO device that does not fall apart at the slightest provocation.

A well-designed LEGO device should be reliable, compact, and sturdy. If it makes extensive use of gears, the geartrain should be able to rotate cleanly and easily. If it is a structural element, it should hold together squarely and resist breaking apart.

This chapter will assist the reader in his or her endeavors to create well-designed LEGO devices. It will introduce some properties of the LEGO Technic system that are not obvious at first glance.

### 4.1 Fundamental LEGO Lengths

Question: how long is the LEGO unit brick?

Answer: One Fundamental LEGO Unit (FLU)!

Actually, the Fundamental LEGO Unit can be expressed in other (more standard) lengths, such as the millimeter. More interestingly, the ratio between the length or width of a brick and its height is not one, but a ratio of two small integers: 6 to 5 (see Figure 4.1).

This ratio, coupled with the existence of one-third height flat pieces, allows the creation of vertical spacings that perfectly match unit horizontal ones (see Figure 4.2). By using these perfect LEGO spacings, vertical stacks of bricks can be reinforced with cross-beams, creating sturdy structures that will not fall apart.

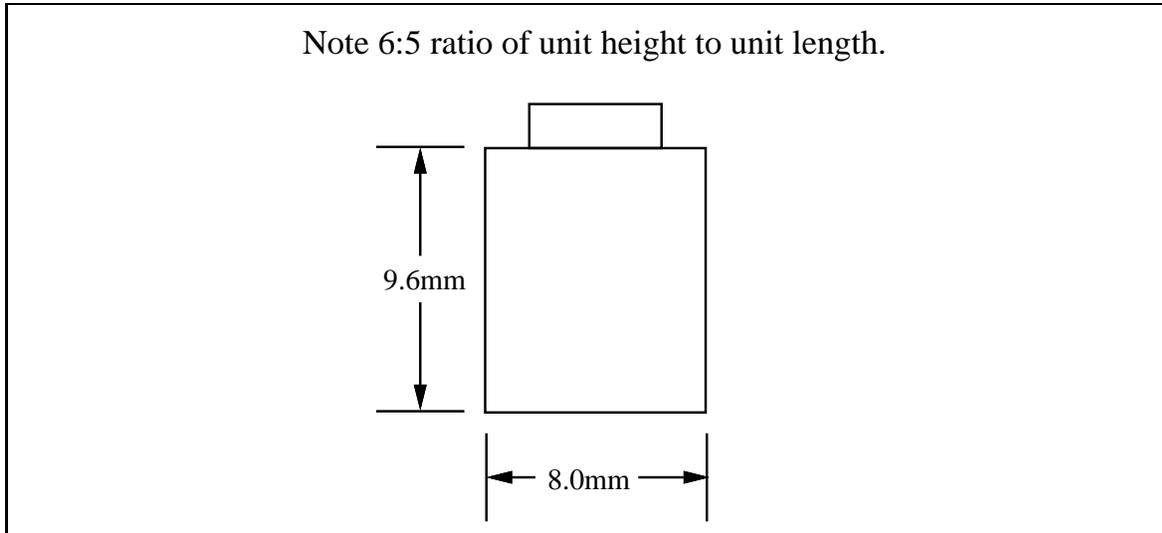


Figure 4.1: The Unit LEGO Brick

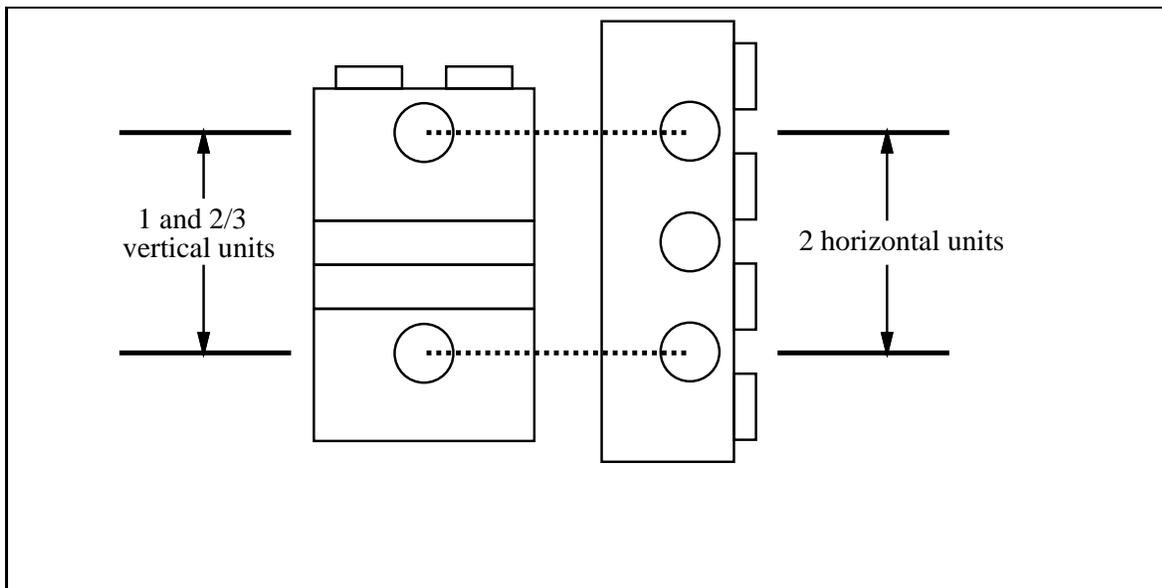


Figure 4.2: Perfect 2-Unit Vertical LEGO Spacing

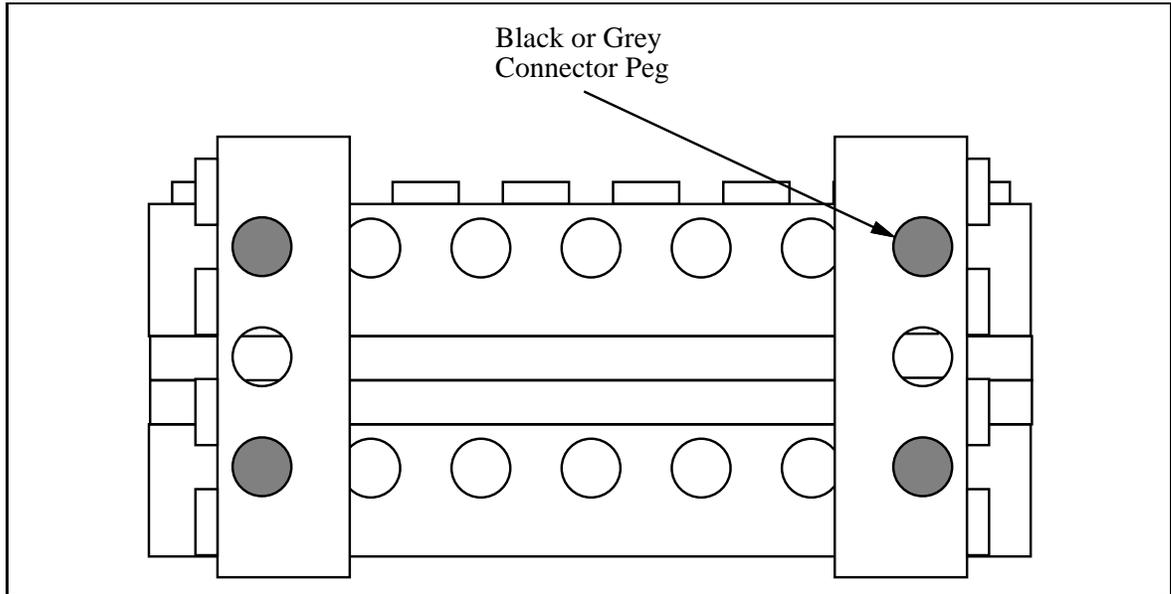


Figure 4.3: Clamping Two Beams at Perfect Vertical Spacing

Figure 4.3 shows an example of two 8-long LEGO beams (separated by a two-unit perfect spacing) braced at the ends by two 4-long LEGO beams. This structure is extremely sturdy.

Other combinations of perfect vertical spacings can be created thanks to the one-third height bricks.

A little mathematics helps us compute all of these standard combinations. Suppose  $a$  represents the number of full-height vertical units and  $b$  the number of one-third height vertical units. Then the height of a LEGO assembly (in mm) would be

$$9.6(a + \frac{1}{3}b) \quad (4.1)$$

since a full vertical unit is 9.6 mm high.

If  $c$  represents the number of horizontal units, then  $8c$  is the length of a LEGO beam in mm. We then need to find integral solutions that make these two quantities equal:

$$9.6(a + \frac{1}{3}b) = 8c \quad (4.2)$$

which reduces to

$$2(3a + b) = 5c \quad (4.3)$$

The following table lists some solutions to this integer equation:

Full Height Units	One-Third Units	Horizontal Units
		
1	2	2
3	1	4
5	0	6
6	2	8
8	1	10

Bracing LEGO structures using the perfect vertical spacings is a key method of creating a structurally sturdy machine.

## 4.2 LEGO Gearing

Making a good LEGO geartrain is indeed a fine art. However, this art can be learned, and having some simple information can make a big difference.

One of the first things to notice about LEGO gears is their diameter, which indicates at what spacings they can be meshed together.

The natural units for the sizes of LEGO gears is the horizontal LEGO spacing unit. The following table shows the radii of the various LEGO gears:

Gear Teeth (number)	Gear Radius (horizontal units)
8	0.5
16	1
24	1.5
40	2.5

Notice that three of the gears (namely, the 8-tooth, 24-tooth, and 40-tooth) have radii that, when used together in pairs, an integral spacing is formed. So, for example, the 8-tooth gear may be used with the 24-tooth or the 40-tooth, but not the 16-tooth.

Figure 4.4 shows how an 8-tooth gear would mesh with a 24-tooth gear along a LEGO beam.

The 16-tooth gears only mesh with each other according to this logic.

Gears may be meshed together at odd diagonals. However, this requires great care, as it is difficult to achieve a spacing that is close enough to the optimal spacing (which can be computed by adding the gears' radii). If the gears are too close, they will bind or operate with high frictional loss; if they are too far, they will slip.

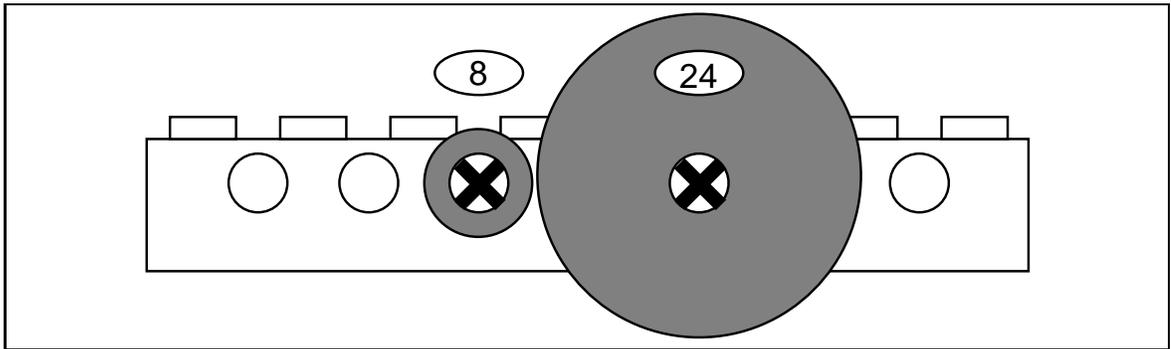


Figure 4.4: Meshing of an 8-Tooth Gear and a 24-Tooth Gear

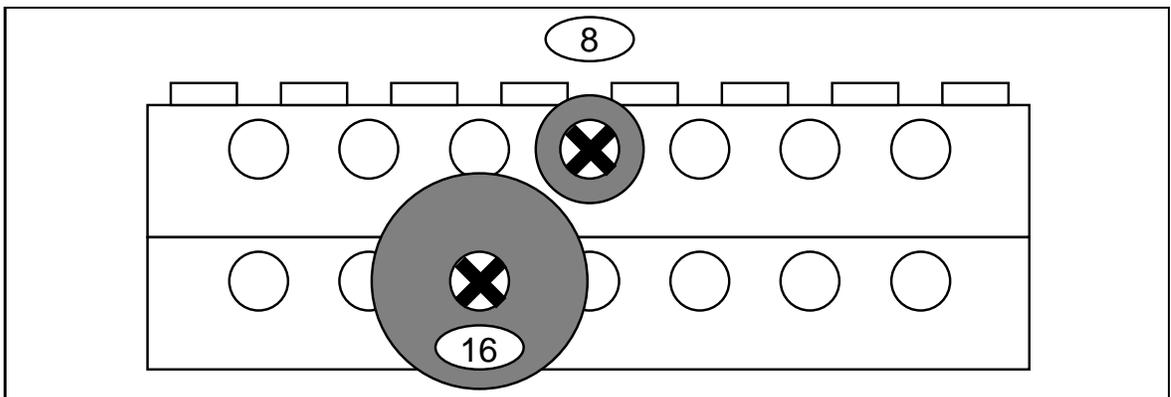


Figure 4.5: Diagonal Meshing of an 8-Tooth Gear and a 16-Tooth Gear

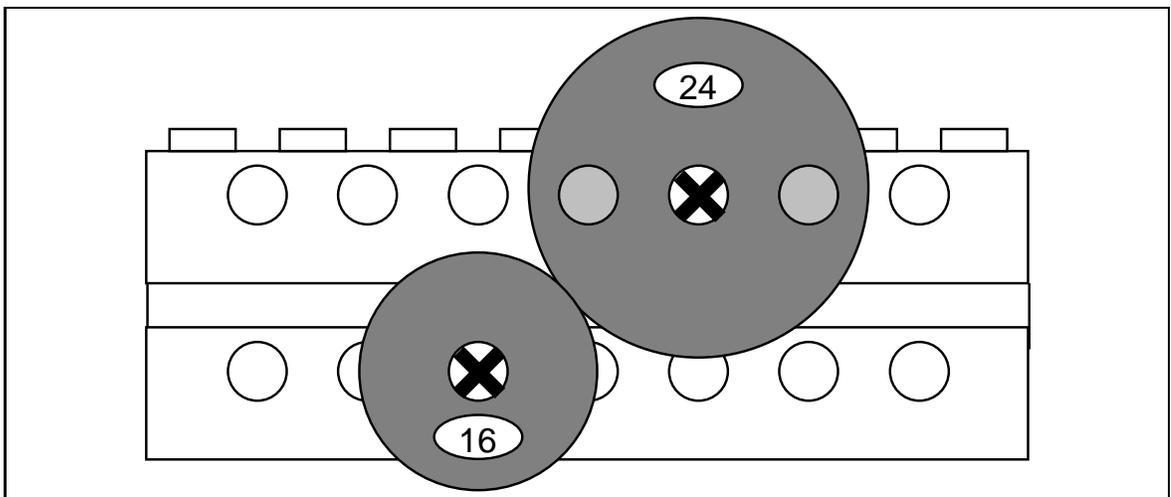


Figure 4.6: Diagonal Meshing of a 16-Tooth Gear and a 24-Tooth Gear

Figures 4.5 and 4.6 show examples of diagonal gearing that have been tested to work well. Other combinations that have good performance may be discovered.

A very high performance geartrain will be necessary in order to drive a robot. For this type of geartrain, the following rules are suggested:

- Do not use the 14-tooth bevel gear (too flimsy), the worm gear (high friction), or the right-angle 24-tooth crown gear (slips under high stresses).
- Do not make a pulley drive using the LEGO rubber bands. They are inefficient (especially in the later stages of a geartrain), they slip, and the rubber bands break or fall off at very inopportune times.
- Do use the 8-tooth and 24-tooth gears. The 40-tooth gears are also good, if they can be fit in despite their large size.
- Try to space the axles at perfect LEGO spacing, or a close diagonal approximation. This is easy to do if the axles are mounted horizontally adjacent on a beam, or vertically using perfect LEGO spacing.
- Try to have each axle supported inside at least two girders. It's also nice to space these support girders from each other. If these two rules are followed, the axles will stay straight and not bind up inside the girders and create a lot of friction.
- Where multiple girders support the same axle, make sure that these girders are firmly attached to each other. If they are not perfectly aligned, the same binding problem described above may happen, and the geartrain could lose a lot of power.
- The axles can bend. Try not to have a gear dangling at the end of an unsupported axle. Either put gears between the girders supporting the axles, or very close to the girders on the outside of the girders. Both are illustrated on the example geartrain. If the gear is two or more LEGO units away from the outside of the girders, problems may arise.
- Don't make the axles fit too tightly. After gears and spacers are put on an axle, make sure the axle can slide back and forth a little bit. It is very easy to lose a lot of power if spacers or gears are pressing up against the girders.

### 4.2.1 Gear Reduction

The purpose of gearing, in addition to transmitting mechanical energy, is to transform it. For the purposes of a drivetrain, the gears will change high speed and low torque

of an electric motor and create the low speed and high torque that is required to move a robot.

It is important to experiment with different gear ratios. The gear ratio determines this important tradeoff between speed and torque.

Figure 4.7 illustrates a sample LEGO geartrain. This geartrain achieves a gear ratio of 243:1 through the use of five ganged pairs of 8-tooth to 24-tooth gear meshings (this is probably a bit overkill for a robot drive).

It is suggested that a copy of this geartrain be built for evaluation—it is an efficient design that follows many of the rules that have been given.

### 4.2.2 Chain Drives

Use of chain drives requires a fair bit of patience on the part of the designer. A fair bit of trial and error design is necessary to find gear spacings that will work for the chain. If the chain is too loose, it may skip under heavy load. If it is too tight, it will lose power.

Experimentation is suggested. The chains tend to work better on the larger gears.

### 4.2.3 Testing a Geartrain

To test a geartrain to see if it is really good, try backdriving it. Take off the motor (if it's on), place a wheel on the slow output shaft, and try to turn the wheel. It should be possible to make all the gears spin freely from this slow axle. If the geartrain is very good, the gears will continue spinning for a second or two after the output shaft is released.

### 4.2.4 Low-Force Geartrains

When building geartrains that will only transmit small forces, many of the design rules don't apply. Some "problems" may turn out to be advantages—it may be desirable to have a transmission which "slips" when it is stuck (so that the motors do not stall) and then a rubber band and pulley drive would be appropriate.

The rubber bands are also useful for mechanisms which need to store energy.

The 24-tooth crown gear (in addition to being perfectly useable as a normal 24-tooth gear) will function at the intended 90 degree angle, as long as it is only transmitting small forces.

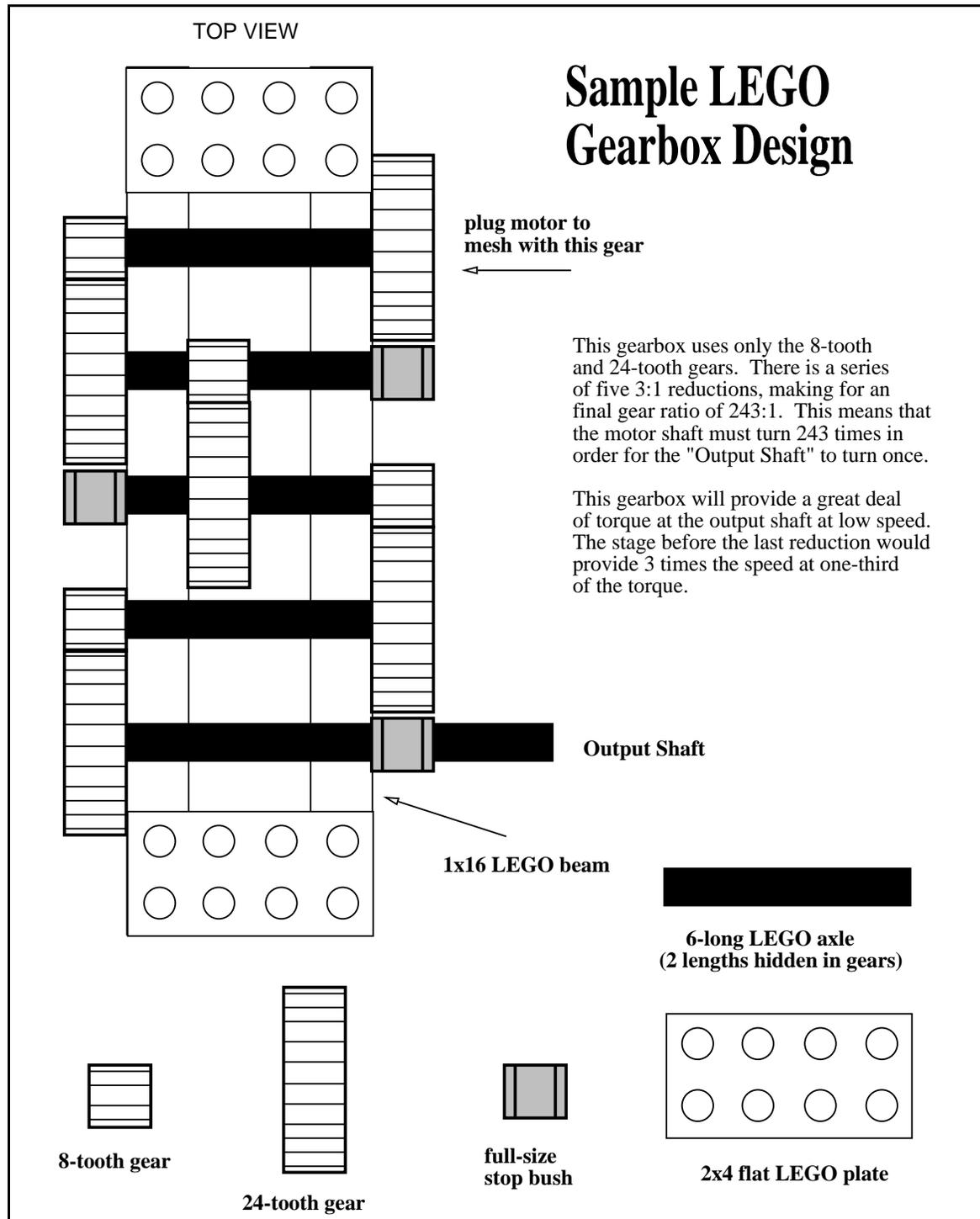


Figure 4.7: LEGO Gearbox Example

# Chapter 5

## Sensor Design

Without sensors, a robot is just a machine. Robots need sensors to deduce what is happening in their world and to be able to react to changing situations.

This chapter introduces a variety of robotic sensors, explaining electrical use and practical application. While many ways to use the various sensors in the 6.270 are mentioned, please do not be limited by the ideas contained in this chapter! The sensor applications presented here are not meant to be exhaustive, but merely to suggest some of the possibilities.

Assembly instructions for the kit sensors are given in Section 1.9.

### 5.1 Sensors as Transducers

The fundamental property of an electronic sensor is to measure some feature of the world, such as light, sound, or pressure, and convert that measurement into an electrically represented quantity.

Typical sensors respond to stimulus by changing their resistance (photocells), changing their current flow (phototransistors), or changing their voltage output (the Sharp IR sensor). One electrical output of a given sensor can be easily converted into other electrical representations.

#### 5.1.1 Analog and Digital Sensors

A distinction is often made between a sensor that is *analog* and a sensor that is *digital*.

An analog sensor produces a *continuously varying* change in value over its range of measurement. For example, a particular photocell might have a resistance of  $1\text{k}\Omega$  in bright light and a resistance of  $300\text{k}\Omega$  in complete darkness.

Digital sensors, on the other hand, have only two states, “on” and “off.” Perhaps the simplest example of a digital sensor is the touch switch. A typical touch switch has infinite resistance when it is not pressed, and zero resistance when it is.

Some sensors that produce a digital output are more complicated. These sensors produce *pulse trains* of transitions between the 0 volt state and the 5 volt state. With these types of sensors, the frequency characteristics of this pulse train convey the sensor's measurement.

An example of this type of sensor is the Sharp modulated infrared light detector. With this sensor, the actual element measuring light is an analog device, but signal-processing circuitry integral to the sensor produces a digital output.

## 5.2 Sensor Inputs on the 6.270 Board

All of the sensor inputs of the 6.270 board measure voltage levels. Some of the sensor inputs measure analog voltages. This means that they measure the voltage and convert it to a number corresponding to the voltage level.

Other sensor inputs are digital. These inputs discriminate only two states: the zero volt state and the five volt state.

### 5.2.1 Analog Inputs

The analog inputs of the board measure voltages from zero volts to five volts. They convert the voltage measured to a number from 0 to 255, where 0 corresponds to a reading of zero volts and 255 corresponds to a reading of five volts. The conversion scale is linear, so a voltage of 2.5 volts would generate a conversion value of 127 or 128.

The C library function `analog(port-#)` is used to return the value of a particular sensor port. For example, the C statement

```
val = analog(3);
```

sets the value of the variable `val` equal to the reading on analog port `#3`.

### 5.2.2 Digital Inputs

According to digital logic standards, if a voltage level on a digital input is less than 2.5 volts, it is considered to be a logic low, or binary zero. If the voltage level is greater than 2.5 volts, it is considered to be a logic high, or binary one. The digital inputs on the 6.270 have a pull-up resistor that makes the voltage input equal to 5 volts when nothing is connected.

However, many devices used as digital sensors are wired to be *active low*, meaning that they generate a 0 volt value when they are active (or true).

Thus a hardware-level measurement taken on a digital input port has the opposite truth value of the state of such a sensor. For example, when a touch sensor is depressed, it will register a 0 volt or logic zero value.

The C library function `digital(port-#)`, used to return a true-or-false value associated with a particular sensor port, performs a logical inversion of the signal measured on a digital port. Hence the depressed touch switch (measuring 0 on the hardware) causes the `digital()` function to return a 1 (logic true).

For example, the C statement

```
if (digital(2)) do_it();
```

calls the function `do_it()` if the value at port #2 was zero volts (indicating a depressed switch).

### 5.2.3 Connector Plug Standard

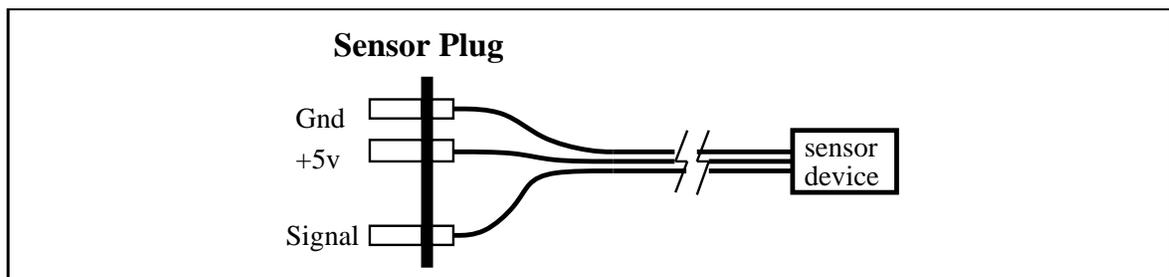


Figure 5.1: Wiring a Generic Sensor

A standard plug configuration has been developed to connect sensors to the 6.270 board, as shown in Figure 5.1.

Notice that one pin is removed from the plug, making it asymmetric and therefore *polarized*. This means that once the plug is wired correctly, it cannot be inserted into a sensor port backwards. This makes the plug much easier to use.

The sensor is connected to the plug with three wires.

Two of the wires are used to supply power from the 6.270 board to the sensor. These are the wires labelled “+5v” and “Gnd.” The third wire, labelled “Signal” is the voltage output of the sensor.

It’s the job of the sensor to use the power wires (if necessary) and return its “answer”, as a voltage, on the `signal` in wire.

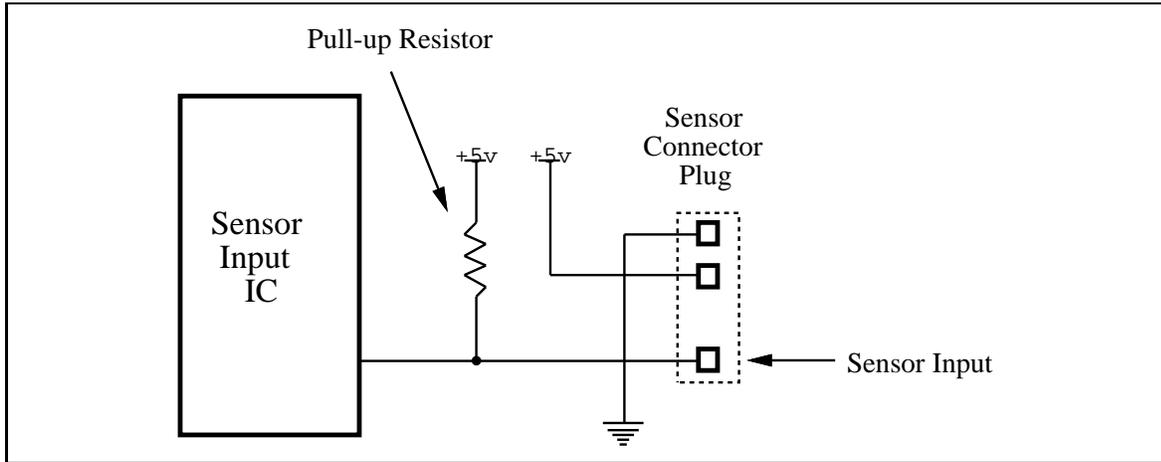


Figure 5.2: Sensor Input Wiring, One Sensor

### 5.2.4 Sensor Input Wiring

Figure 5.2 shows a diagram of circuitry associated with each sensor. This circuitry, residing on the 6.270 board, is replicated for each sensor input channel.

The key thing to notice about this diagram is the *pull-up resistor* wired from the sensor input signal to the 5 volt power supply.

There are several reasons why this resistor exists. One function it has is to provide a “default value” (value when no sensor is plugged in) to the sensor input. Many IC’s do not perform well when their inputs are left unconnected. The pull-up resistor acts when nothing is plugged into the sensor port. It biases the voltage on the sensor input line, causing it to be 5 volts in this situation.

Because of the pull-up resistor, the default value of an analog input is 255. The default value of a digital input is binary 1 or logic true (although this value is inverted by the `digital()` library function, as explained earlier).

The pull-up resistor also acts in *voltage divider* circuits, explained immediately following, which are used in many analog sensors. It eliminates the need for a resistor on each of the sensors.

### 5.2.5 The Voltage Divider Circuit

Most of the sensors used in the 6.270 kit make use of the voltage divider circuit, shown in Figure 5.3. In the voltage divider, the voltage measured at the tie point of the two resistors,  $V_{out}$ , is a function of the input voltage,  $V_{in}$ , and the values of the two resistors,  $R_1$  and  $R_2$ .

Using Ohm’s Law,  $V = I \times R$ , the output voltage  $V_{out}$  can be solved for.  $i$ , the amount of current flowing through the circuit shown in the diagram, is  $\frac{V_{in}}{R_1 + R_2}$

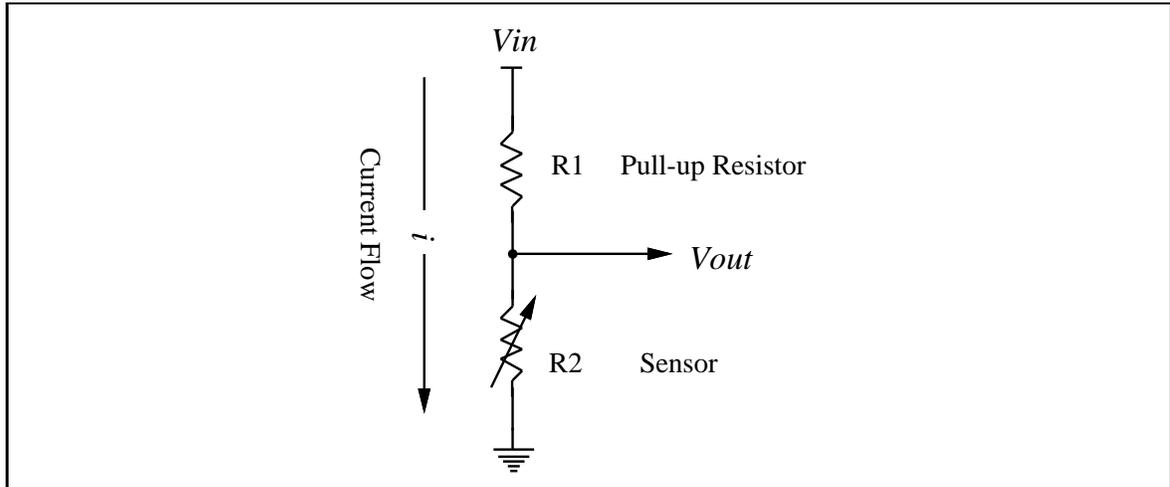


Figure 5.3: Voltage Divider Schematic

(calculated using the rule that series resistances add). Then  $V_{out}$ , the voltage drop across  $R_2$ , is  $R_2 \times i$ , which yields the result:

$$V_{out} = V_{in} \frac{R_2}{R_1 + R_2} \quad (5.1)$$

In the 6.270 application,  $R_1$  is a fixed value as part of the sensor input circuitry (as shown in Figure 5.2), and  $R_2$  is the resistive sensor.  $V_{in}$  is the positive voltage supply, fixed at 5 volts.

Thus the  $V_{out}$  signal can be directly computed from  $R_2$ , the resistive sensor. From looking at the equation, it is easy to see that if  $R_2$  is large with respect to  $R_1$ , the output voltage will be large, and if  $R_2$  is small with respect to  $R_1$ , the output voltage will be small.

## 5.3 Tactile Sensors

Several types of sensors to detect tactile contact are exist in the 6.270 kit:

**Touch Switch.** A simple pushbutton or other momentary switch.

**Bend Sensor.** A unique device that measures amount of flex in a thin plastic strip.

**Potentiometer.** Rotary and linear potentiometers (variable resistors) can be used.

### 5.3.1 Touch Switch

The most primitive, but often very useful, sensor is a touch switch. It is simply a pushbutton or other momentary switch that is mounted on a robot so that when the robot runs into something, the switch is triggered. The robot can detect that it has made contact with some object.

Touch sensors used as collision detectors are imperfect in that it is hard to design a touch switch and bumper mechanism that can detect contact of any object from any angle. Creative mechanical design in implementing the bumper mechanism is important.

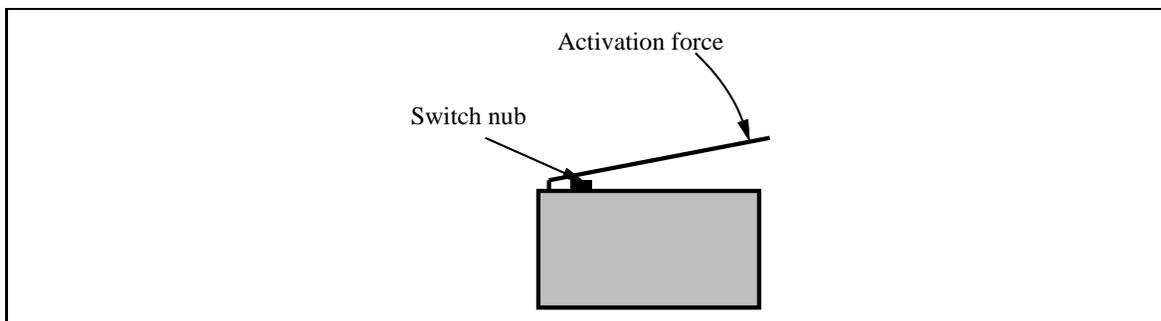


Figure 5.4: A Typical Microswitch

*Microswitch* is the brand name of a variety of switch that became very popular and is a generic term for this type of switch now manufactured by many companies. Microswitches are an especially good type of switch to use for making touch sensors.

A microswitch is housed in a rectangular body and has a very small button (the “switch nub”) which is the external switching joint. Usually, microswitches are also equipped with lever arms to reduce the force needed to actuate the switch (see Figure 5.4).

Because of the lever arm, very little force is needed to actuate the switch. A very sensitive touch bumper can be made by connecting a mechanism as an extension of the microswitch’s lever arm, as illustrated in Figure 5.5.

### Limit Switch

Touch sensors can also be used to act as *limit switches* for devices that are confined to a known path of travel. For example, when using a mechanism riding on a gear rack, touch switches could detect when the mechanism reached the limit of travel on the rack.

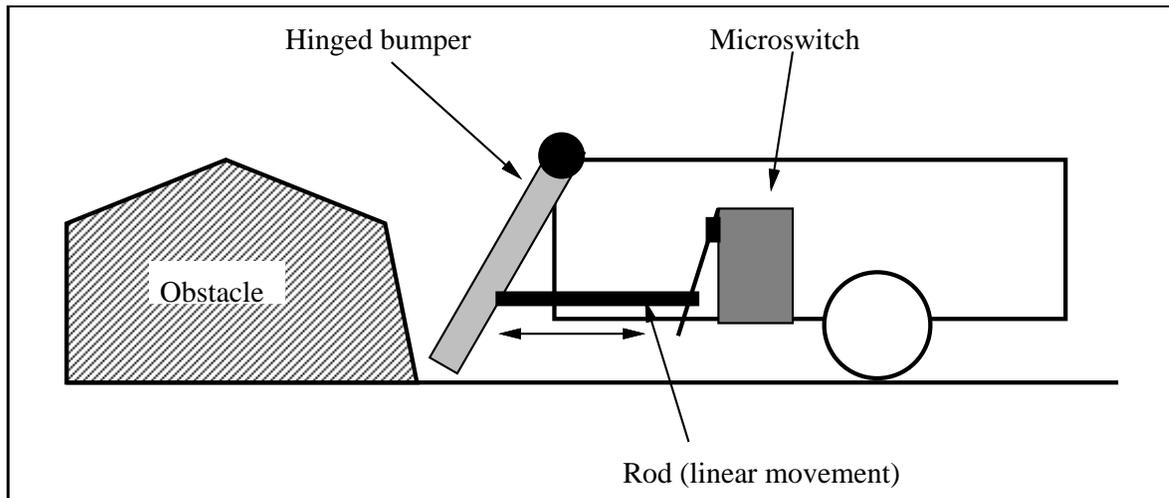


Figure 5.5: Robotic Platform Employing Bumper Coupled to Touch Sensor

### Switch Circuitry

Figure 5.6 shows how a switch is wired to a sensor input port. When switch is open (as it is shown in the diagram), the sensor input is pulled to the 5 volt supply by the pull-up resistor. When the switch is closed, the input is tied to ground, generating a zero volt signal.

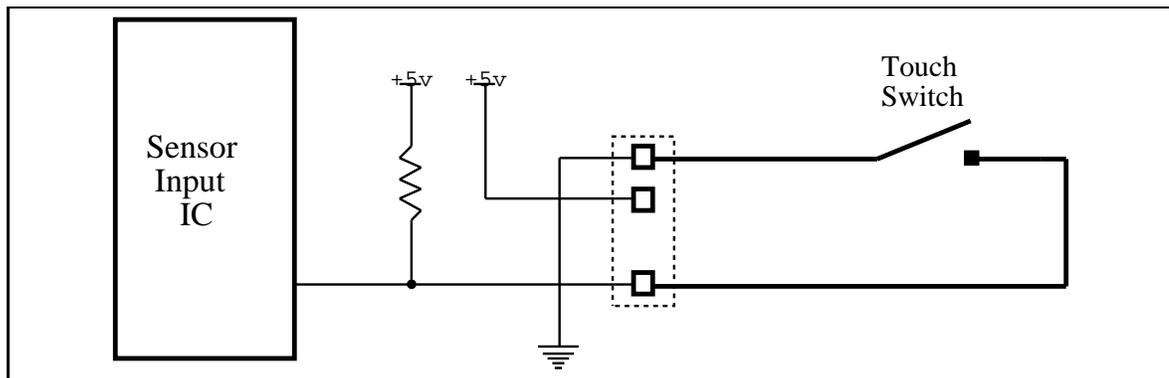


Figure 5.6: Touch Switch Circuit

Most pushbutton-style switches are “normally open,” meaning that the switch contacts are in the open-circuit position when the switch is in the non-depressed state. Microswitches often have both normally open and normally closed contacts and a common contact. When wiring a touch sensor with a microswitch, it is customary to use the normally open mode because this is more common. Also, this configuration is the active-low mode expected by the standard library software used to input values

from digital sensor ports.

### 5.3.2 Bend Sensor

The bend sensor included in the 6.270 kit is a unique device that was developed for use in the Nintendo (R) Power Glove, a product that allows hand and finger movement as input to the popular video game. The bend sensor is used in this device to measure the degree of flex in the fingers of the Power Glove.

The bend sensor device was developed by Abrams-Gentile Entertainment, Inc. (AGE), the manufacturers of the Power Glove. AGE is a sponsor of the 6.270 contest and has donated the bend sensors to the contest.

The bend sensor is a variable resistor. When the bend sensor strip is bent (so that the printed element is on the outside curve of the bend), the resistance of the element increases.

The bend sensor has two printed layers. The lower layer is a carbon-based material of constant resistance, similar to the material that is used to fabricate simple resistors. The upper layer is a silver-colored granulated conductive material.

When the strip is flat, the granules of the silver material lie close to each other and conduct electricity, reducing the amount of resistance created by the resistive material. When the strip is bent, the granules are forced apart, and the resistive material carries more of the current, causing an increase in resistance.

### Applications

Bend sensors are often much more useful than touch sensors because they can indicate not only the presence of an obstacle, but its distance from the robot. It is quite possible to have a robot follow along a wall edge by using bend sensors to measure the robot's distance to the wall.

### Electrical Data

The bend sensor is a simple resistor. It is interfaced to the 6.270 board by functioning as resistor  $R_2$  in the voltage divider circuit shown in Figure 5.3.

The bend sensor assembly instructions propose the construction of a dual bend sensor, in which two bend sensors are laid back to back, so that the combined element can sense bend in either direction.

There are a few ways the configuration could be wired to the sensor input circuitry. One possibility is to wire the two bend sensors in parallel; if either one of them is bent in the active direction, the parallel resistance will decrease.

Another possibility (this one is recommended in the assembly notes) is to wire the two bend sensors into a voltage divider. The pullup resistor will effectively be

in parallel with one of the bend sensors, skewing the response of the overall circuit. Still, this configuration will indicate which of the two bend sensors has been flexed, by having the voltage measurement increase or decrease from the relaxed state.

### 5.3.3 Potentiometer

A potentiometer (or “pot,” for short) is a manually-adjustable, variable resistor of the type that is commonly used for volume and tone controls in stereo equipment. In robotics, a potentiometer can be used as a position sensor.

#### Rotary Potentiometer Use

A rotary potentiometer (the most common type) can be used to measure the rotation of a shaft. It is easiest to use if the shaft being measured does not need to rotate continuously, but rather would rotate back and forth (as in a sensor “radar dish”).

Use of gears is a simple way to lock the rotation of the shaft being measured to the potentiometer. By using a gear ratio other than 1:1, a shaft that needed to rotate more than the 270 degrees or so of the pot could be measured. It is also possible to modify the potentiometer to allow continuous rotation, although there would be a portion of the rotation for which no measurement would be available.

#### Linear Potentiometer Use

A linear pot could be used to measure the position of a linear mechanism or other mechanisms for which a linear measurement could be taken.

#### Electrical Data

Potentiometers are variable resistors. They have three terminals with the center terminal being a center tap contact that slides across an element of constant resistance.

The assembly instructions suggest wiring the potentiometer in the voltage divider configuration, with the on-board pullup resistor in parallel with one of the potentiometer’s two effective resistances (Figure 5.7). This will yield readings of greater precision (although they will not be linear) than if the pot were used as a two-terminal variable resistor.

## 5.4 Light Sensors

Light sensors may well be the most common form of robotic sensor (other than touch switches). A variety of light sensors are provided in the 6.270 kit:

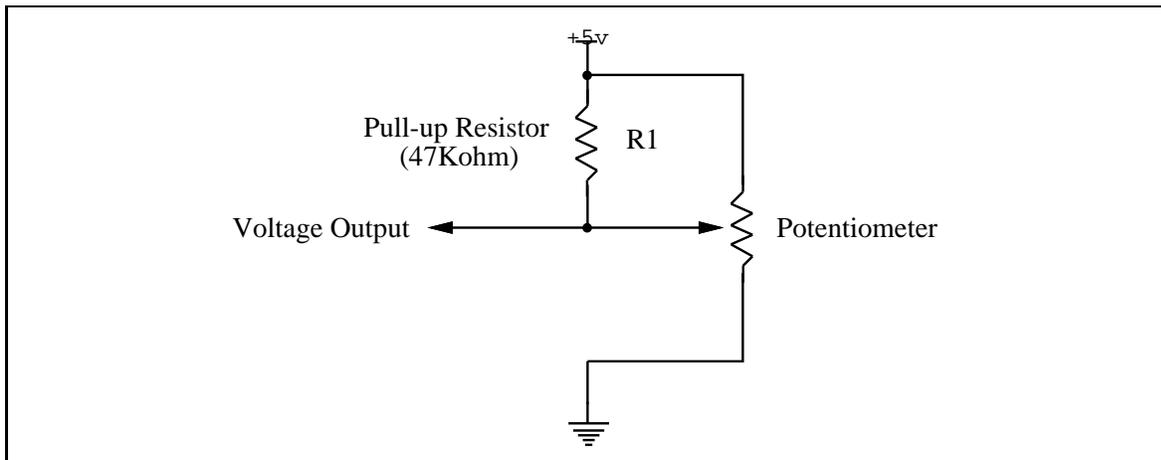


Figure 5.7: Potentiometer Circuit

**Photocell.** This device is a light-dependent resistor. It is most sensitive to red visible light.

**Infrared Reflectance Sensor.** This device combines a phototransistor (sensitive to infrared light) and an infrared emitter into a single package. The amount of light reflected from the emitter into the phototransistor yields a measurement of a surface's reflectance (when other factors, such as the distance from the sensor to the surface, are held constant).

This sensor can be used to detect features drawn on a surface or segments on a wheel used to encode rotations of a shaft.

**Infrared Slotted Optical Switch.** Similar to the aforementioned device, this device uses an infrared emitter aimed directly at an infrared phototransistor. The phototransistor detects makes and breaks of the infrared beam of light.

This sensor can also be used to make a shaft encoder, using a wheel that has been drilled with holes inside the circumference edge.

**Modulated Infrared Light Detector.** This device senses the presence of infrared light that has been modulated (e.g., blinks on and off) at a particular frequency.

These devices are typically used to decode the signals of TV remote controls, but are used in the 6.270 application to detect the infrared beacon of the opponent robot.

### 5.4.1 Photocell

Photocells are made from a compound called cadmium sulfide (CdS) that changes in resistance when exposed to varying degrees of light. Cadmium sulfide photocells are most sensitive to visible red light, with some sensitivity to other wavelengths.

Photocells have a relatively slow response to changes in light. The characteristic blinking of overhead fluorescent lamps, which turn on and off at the 60 Hertz line frequency, is not detected by photocells. This is in contrast to phototransistors, which have frequency responses easily reaching above 10,000 Hertz and more.

#### Applications

Photocells can be used to detect the incandescent lamp that acts as a contest start indicator. They also can be used to measure the reflectivity of the playing field surface if coupled with a light source such as a red LED or an incandescent lamp.

Photocells can be used to detect room ambient lighting. With the use of polarizing filters placed immediately in front of the photocell element, an array of such photocells can be used to detect the polarization angle of polarized light present in ambient lighting. Robot-level control programs can be used to seek out such sources of polarized light.

#### Electrical Data

The photocell acts as resistor  $R_2$  in the voltage divider configuration discussed in Section 5.2.5.

Photocells yield a decrease in resistance with an increase in illumination. Because of the wiring of the voltage divider (the photocell is on the lower rung), an increase in light will correspond to a decrease in voltage measurement.

### 5.4.2 Infrared Reflectance Sensor

The infrared reflectance sensor is a small rectangular device that contains an phototransistor (sensitive to infrared light) and an infrared emitter. The amount of light reflected from the emitter into the phototransistor yields a measurement of a surface's reflectance (when other factors, such as the distance from the sensor to the surface, are held constant).

The phototransistor has peak sensitivity at the wavelength of the emitter (a near-visible infrared), but is also sensitive to visible light and infrared light emitted by visible light sources. For this reason, the device should be shielded from ambient lighting as much as possible in order to obtain reliable results.

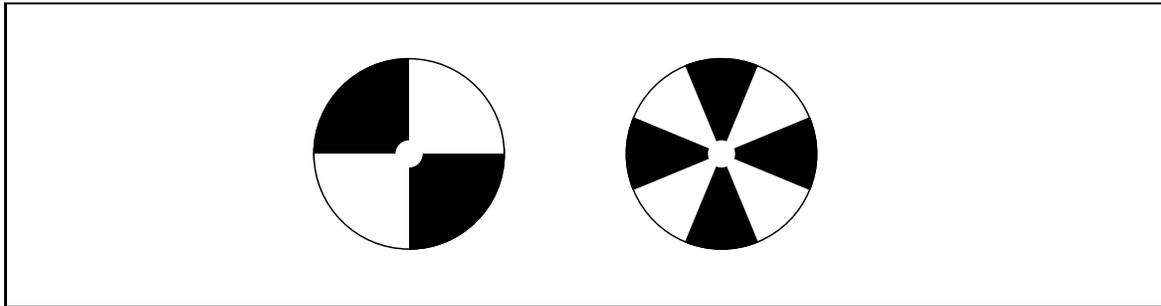


Figure 5.8: Two Optical Shaft Encoder Wheels

### Applications

**Reflectance Measurement** The reflectance sensor is typically used to measure the reflective properties of a surface. If other factors that affect amount of reflected light are held constant, such as distance between the sensor and the surface and presence of ambient lighting, then the amount of light reflected by the surface will be measured.

It's important to remember that the reflectivity measurement is an indication of the surface's reflectivity at a particular wavelength of light (the near visible infrared). A surface's properties with respect to visible light may or may not be indicators of infrared light reflectance. In general, though, surfaces that absorb visible light (making them appear dark to the naked eye) will absorb infrared light as well.

**Visible Light Measurement** Since the phototransistor responds to visible light sources, the device can also be used to measure the light from the contest starting lamp.

**Optical Shaft Encoder** By measuring light and dark segments of a wheel mounted on a rotating shaft, the angular displacement and velocity of the shaft can be measured. Either LEGO encoder wheels designed for this purpose or handmade substitutes, as shown in Figure 5.8 can be used.

Standard 6.270 software library functions perform the action of counting transitions on the encoder wheel as measured by the reflectance sensor.

### Electrical Data

**Phototransistor** The phototransistor acts as a transistor with the base voltage determined by the amount of light hitting the transistor. Hence it acts a variable current source. Greater amounts of light cause greater currents to flow through the collector-emitter leads.

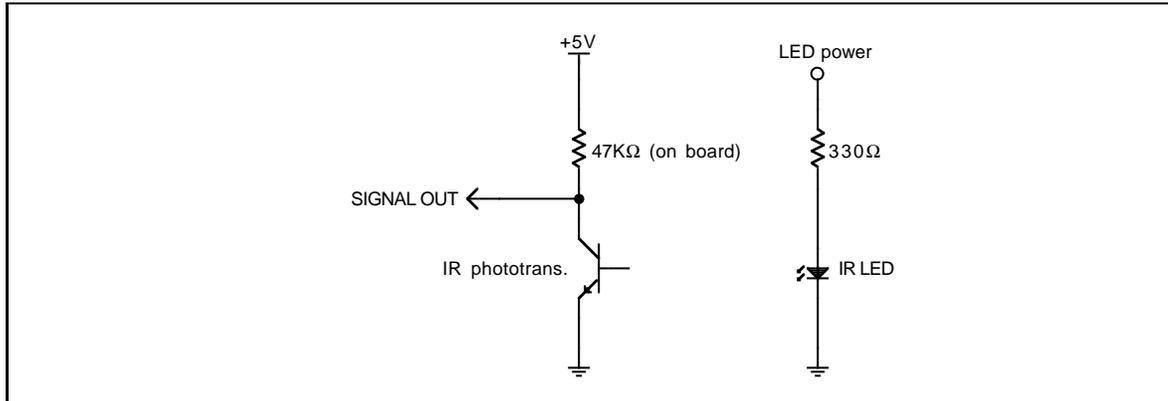


Figure 5.9: Phototransistor and Infrared Emitter Circuit

As shown in Figure 5.9, the phototransistor is wired in a similar configuration to the voltage divider. The variable current travelling through the resistor causes a voltage drop in the pullup resistor. This voltage is measured as the output of the device.

**Infrared Emitter** The emitter element uses a resistor to limit the current that can flow through the device to the proper value to about 10 milliamps.

### 5.4.3 Infrared Slotted Optical Switch

The infrared slotted optical switch is similar to the infrared reflectance sensor except that the emitter is pointed directly into the phototransistor.

#### Applications

The optical switch is commonly used to build shaft encoders. A wheel with holes or slots is necessary.

The optical properties of making and breaking a beam of light versus measuring reflectance are often superior.

#### Electrical Data

The optical switch operates in the same fashion as the infrared reflectance sensor, with the exception that a different value of pullup resistor must be added externally for the particular model of optical switch in use.

### 5.4.4 Modulated Infrared Light Detector

The modulated infrared light detector is a device that combines an infrared phototransistor with specialized signal processing circuitry to perform the function of detecting a particular frequency of light.

The 6.270 kit includes the Sharp GP1U52 sensor, which detects the presence of infrared light modulated at 40,000 Hz.

This sensor is used in the 6.270 application to detect infrared light that is emitted by the opponent robot.

Principles of operation and use are explained in Section B.7, which discusses also the infrared transmission circuit. An explanation of the software interface to the Sharp sensors is given in Section 7.8.2.

## 5.5 Other Sensors

### 5.5.1 Magnetism Sensor

A device called a Hall effect sensor is used to detect the presence of magnetic fields. The sensor provided in the 6.270 kit (SS41 series by Micro Switch) is a digital device that detects magnetic field.

The SS41 sensor is magnetically bipolar, meaning that it requires positive gauss (south pole) to latch to zero volts output, and negative gauss to release. The sensor latches at +40 gauss and releases at -40 gauss.

#### Application

The Hall effect sensor can be used to detect the magnetic strips present on the lower wall at the circumference of the 6.270 playing table. Some ingenuity may be required to make the device release after sensing the magnetic field. It might be possible to power the Hall sensor from an LED output. Turning the LED output power off should reset the device.

Hall effect sensors can be used to make a magnetic shaft encoders, by mounting a small piece of magnet on a wheel that rotates past the sensor element.

They can be used to build a proximity sensor or bounce-free switch, detecting when a magnet mounted on a moving member is near to the sensor element.

They can be used to make an inclination sensor by dangling a magnet above the sensor.

#### Electrical Data

The SS41 series Hall effect sensor included in the 6.270 kit is a digital device that operates off of a 5 volts power supply. It uses 4 mA of current for standard operation.

It can sink 10 mA of current into its output, creating a logic low. The sensor cannot drive a logic high and therefore requires a pullup resistor for proper operation.

### 5.5.2 Motor Current Sensor

Circuitry implemented on the motor output drivers of the 6.270 Microprocessor Board detects the amount of current being used by a motor. This corresponds to the load of the motor, and hence can be used as a stall sensor.

This functionality is explained in Section B.4.1.



# Chapter 6

## Robot Control

Most people who have seen past 6.270 contests (or any of the raw videotape footage of past contests), remember round after round of wedged robots—robots that got stuck against a wall, or stuck against another robot, and stayed that way for the duration of a round.

Often these robots did not suffer from mechanical or electronic failures, but rather from poor software design. Perhaps a robot were running into the wall, and its front touch sensor did not trigger. This robot was not physically stuck, but it was “mentally stuck”: its program did not account for this situation, and did not provide a way for the robot to get free.

Robot after robot has failed in this way, infuriating their creators and boring the 6.270 audience. This chapter will discuss the “hard truth” about robot sensors and present a framework for thinking about control that will help the reader prevent his or her robot from suffering the same fate.

A few words of advice: most people severely underestimate the amount of time that is necessary to write control software. A program can be hacked together in a couple nights, but if a robot is to be able to deal with a spectrum of situations in a capable way, more work will be required.

Also, it’s very painful to be developing contest-ready software at the same time as one is making changes hardware. Any hardware change will necessitate software changes. Some of these changes may be obvious but others will not. One sad story is of a team that revamped their robot’s entire geartrain in between the first and second nights of the contest. The new geartrain performed much better, but all of the timing parameters in their code needed to be adjusted. The team never recovered from their mechanical “upgrade.”

In the past it has been observed that that the most successful robots were the ones that had their mechanics and sensors completed about a week before contest. We urge you to make this your goal: Finish your robot with a week to go, and develop your software based upon a stable hardware platform.

## 6.1 Basic Control Methods

### 6.1.1 Feedback Control

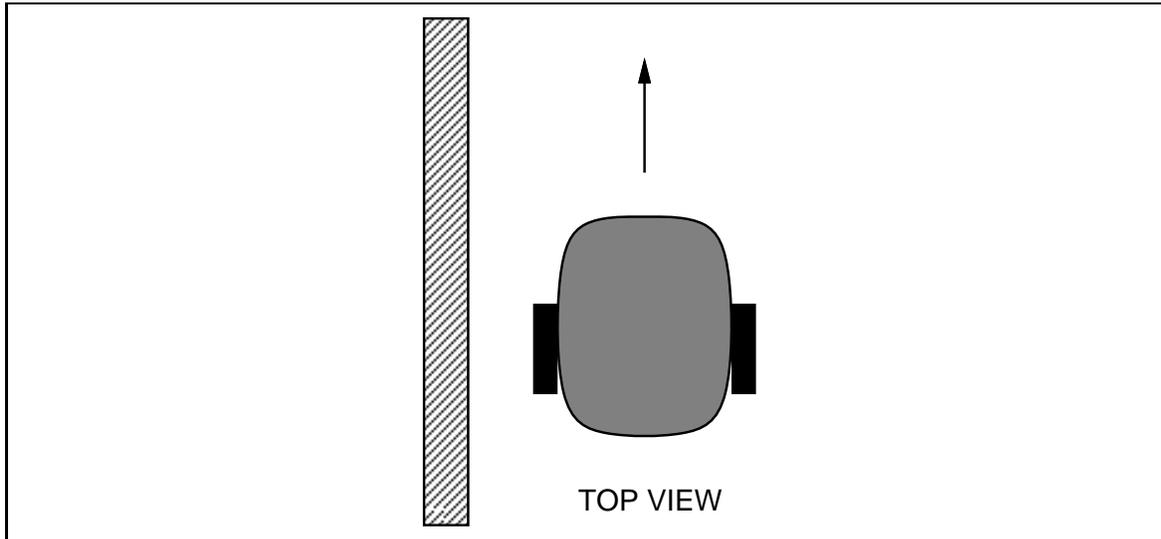


Figure 6.1: Driving along a Wall Edge

Suppose it is desired for the robot to drive with its left side is near the edge a wall, following the wall edge (see Figure 6.1). What's the best way to accomplish this activity?

One solution would be to orient the robot exactly parallel to the wall, and then drive straight ahead. However, this simple solution has two problems: if the robot is not initially oriented properly, it will fail. Also, unless the robot were extremely proficient at driving straight, it will eventually veer from its path and drive either into the wall or into the playing field.

The common and effective solution is to build a negative feedback loop. With continuous monitoring and correction, a goal state (in this case, maintaining a constant distance from a wall) can be achieved.

The bend sensors provided in the 6.270 kit would make an excellent sensor to measure the distance from robot and the wall (see Figure 6.2). The closer the robot is to the wall, the more the bend sensor is bent.

Suppose a function were written using the bend sensor to discerned three states: `TOO_CLOSE`, `TOO_FAR`, and `JUST_RIGHT` (from the wall). Here is a possible definition of the function, called `wall_distance()`:

```
int TOO_CLOSE= -1;
int JUST_RIGHT= 0;
```

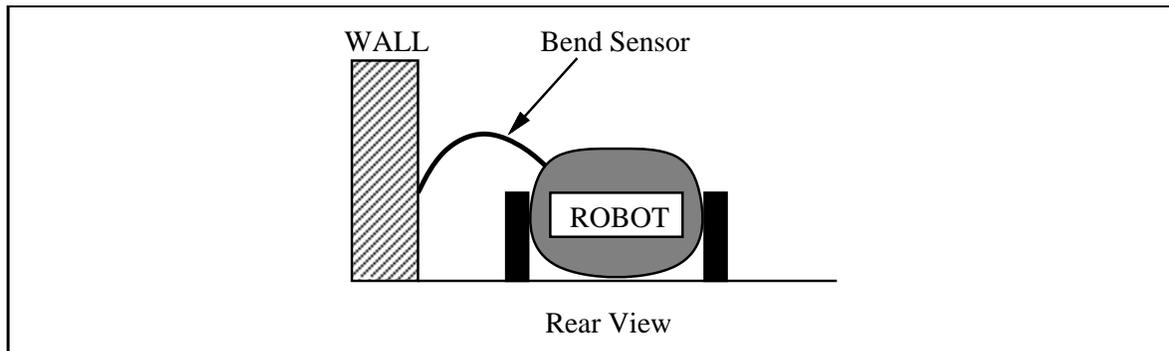


Figure 6.2: Using a Bend Sensor to Measure Distance to a Wall

```
int TOO_FAR= 1;

int wall_distance()
{
    /* get reading on bend sensor */
    int bend_value= analog(BEND_SENSOR);

    /* assume smaller values mean more bending of sensor */
    if (bend_value < TOO_CLOSE_THRESHOLD) return TOO_CLOSE;
    if (bend_value > TOO_FAR_THRESHOLD) return TOO_FAR;
    return JUST_RIGHT;
}
```

Now, a function to drive the robot making use of the `wall_distance()` function would create the feedback. In this example, the functions `veer_away_from_wall()`, `veer_toward_wall()`, and `drive_straight()` are used to actually move the robot, as shown in Figure 6.3.

```
void follow_wall()
{
    while (1) {
        int state= wall_distance();

        if (state == TOO_CLOSE) veer_away_from_wall();
        else if (state == TOO_FAR) veer_toward_wall();
        else drive_straight();
    }
}
```

Figure 6.3: Wall-Following Function

Even if the function to drive the robot straight were not exact (maybe one of the robot's wheels performs better than the other), this function should still accomplish its goal. Suppose the “drive straight” routine actually veered a bit toward the wall. Then after driving straight for a bit, the “follow wall” routine would notice that the robot was too close to the wall, and execute the “veer away” function.

The actual performance of this algorithm would be influenced by many things, including:

- How sharply the “veer away” and “veer toward” functions made the robot turn.
- How well the bend sensor measured the distance to the wall.
- The settings of the the `TOO_CLOSE_THRESHOLD` and `TOO_FAR_THRESHOLD` values.
- The rate at which the `follow_wall()` function made corrections to the robot's path.

Still, use of a negative feedback loop<sup>1</sup>, ensures the construction basically stable and robust performance, once the parameters are tuned properly.

### 6.1.2 Open-Loop Control

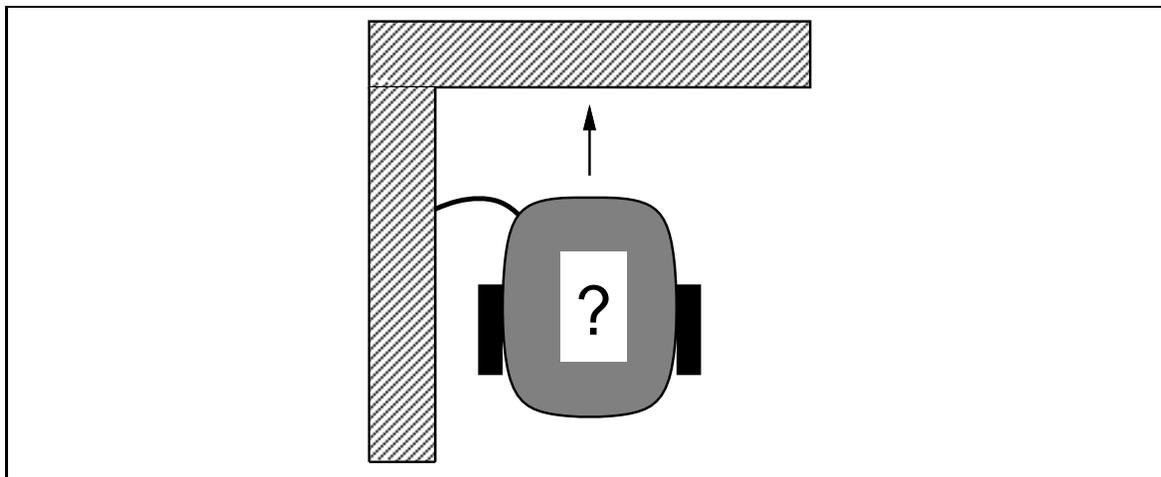


Figure 6.4: Negotiating a Corner

Suppose now the robot has been following the wall, and a touch sensor indicates that it has reached the far edge. The robot needs to turn clockwise ninety degrees to

<sup>1</sup>The type of feedback just described is called *negative* feedback because the corrections subtract from the error, making it smaller. In *positive* feedback, corrections add to the error. Such systems tend to be unstable.

continue following the edge of the wall (see Figure 6.4). How should it accomplish this?

One simple method would be to back up a little and execute a turn command that was timed to accomplish a ninety degree rotation. The following code fragment illustrates this idea:

```
.....
robot_backward(); sleep(.25); /* go backward for 1/4 second */
robot_spin_clockwise(); sleep(1.5); /* 1.5 sec = 90 degrees */
.....
```

This method will work reliably only when the robot is very predictable: if a turn command of 1.5 seconds always produces a rotation of 90 degrees. As might be guessed, this is far from being the case. Many factors affect the performance of a timed turn, including the battery strength, traction on the surface, and friction in the geartrain.

This method is called *open-loop control* (as compared to *closed-loop control*) because there is no feedback from the commanded action as to its effect on the state of the system. If the command is tuned properly and the system is very predictable, open-loop commands can work fine, but generally, closed-loop control is necessary for good performance.

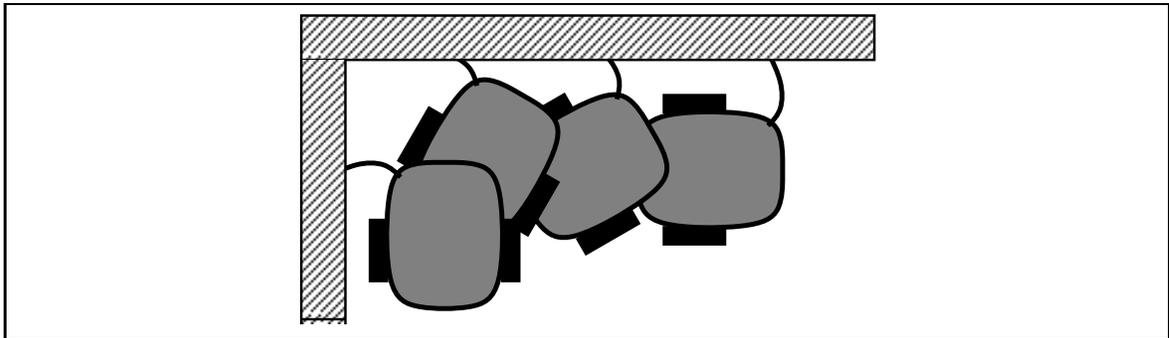


Figure 6.5: Negotiating a Corner with Touch Sensor Feedback

How could the corner-negotiation action be made into a closed-loop system? One approach (which has been used successfully in past 6.270 robots) is to have the robot make little turns, drive straight ahead, bonk the wall, back up, and repeat (see Figure 6.5). The corner is negotiated with a series of little steps.

### 6.1.3 Feed-Forward Control

There are certain advantages to open-loop control, most notably: speed. It's easy to see that a single timed turn would be much faster than a set of small turns, bonks, and back-ups.

One approach when using open-loop control is to use *feed-forward* control. This means that the commanded signal is a function of some parameters that are measured in advance. For the timed turn action, battery strength is probably one of the most significant factors determining the required time of the turn. Using feed-forward control, a battery strength measurement would be used to “predict” how much time is needed for the turn. Note that this is still open-loop control—the feedback is not based on the actual result of a movement command—but a computation is being performed to make the control more accurate.

For this example, the battery strength could be measured or estimated based on usage since the last charge.

### 6.1.4 Summary

For the types of activities commonly performed by 6.270 robots, feedback control is very useful for:

- **Wall following.** As discussed in this section.
- **Line following.** Using one or more reflectance sensors aimed at the surface of the 6.270 playing field.
- **Infrared tracking.** Homing in on a source of infrared light, using the Sharp IR sensors.

Open-loop control should probably be used only sparingly, and in time-critical applications. Ideally, small segments of open-loop actions interspersed between feedback activities should work well.

Feed-forward techniques can enhance the performance of open-loop control, when it is used.

## 6.2 Sensor Calibration

### 6.2.1 Manual Sensor Calibration

The function `wall_distance()` (used an example in the Section 6.1.1) used threshold variables (`TOO_FAR_THRESHOLD` and `TOO_CLOSE_THRESHOLD`) to interpret the data from the bend sensor. Depending on the actual reading from the bend sensors, and the settings of these threshold variables, `wall_distance()` determined if the robot was “too close,” “too far,” or “just the right” distance from the wall.

Proper calibration of these threshold values is necessary so that the robot can perform properly. Often, it is convenient to write a routine to allow interactive manipulation of the robot’s sensors to determine the proper calibration settings.

For the bend sensor, a calibration routine could be included that allowed flexing the sensor for a small amount of bend (the `TOO_FAR_THRESHOLD`), and then depression of one of the user buttons. The routine would then “capture” the value of the bend sensor at that point, and use it as the appropriate threshold. Similarly, a large amount of flex could be applied to the sensor and then captured as the `TOO_CLOSE_THRESHOLD` value.

Later, the values of these thresholds could be noted when the robot is performing particularly well. These “optimal” settings could be hard-coded as default values. The calibration routine could be kept, for use under circumstances, or if other parameters affecting the robot’s performance cause the need for adjustment of the calibration settings again.

### 6.2.2 Dealing with Changing Environmental Conditions

Calibration routines are particularly important when environmental conditions will cause fluctuations in sensor values. Two types of sensor are strongly affected either by external environmental conditions or by the internal state of the robot:

**Light Sensors.** Heavily affected by room lighting (ambient light), unless extremely well-shielded.

**Motor Force Sensing.** Dependent on battery voltage. When battery weakens, force readings increase.

#### Light Sensors

Any light sensor will operate differently in different amounts of ambient (e.g., room) lighting. For best results when using light sensors, they should be physically shielded from room lighting as much as possible, but this is not usually perfect. Given that room lighting will affect nearly all light sensors to some degree, software should be designed to compensate for room lighting.

When using reflectance-type or break-beam light sensing, a good strategy is to be able to control the sensor’s own illumination source. If a sensor reading is taken with the sensor’s own illumination off, the reading due to ambient light is measured. If then a reading is taken with the illumination on, the result is a value due to ambient light plus the sensor’s own illumination. By subtracting these two values, the sensor reading due to its illumination alone can be obtained.

Still, this method will not wholly eliminate the influence of ambient light. Further calibration in an actual performance environment will probably be necessary. For this, the use of global variables and an initial calibration routine is suggested.

## Motor Force Sensing

Ideally, a direct measurement of the battery voltage could be used in a function to eliminate its effect on the motor force readings.

However, a simple calibration sequence might suffice. A routine could be written to take motor force readings while the robot was in motion. By manually providing resistance to the robot (and pressing a user button when ready), the resistance that the robot should interpret as “being stuck” could be directly measured.

This calibration sequence would need to be performed periodically over the life cycle of the motor battery.

### 6.2.3 Using Persistent Global Variables

Persistent global variables are a special type of global variable that keep their state in between pressing reset or turning the robot on and off. These are ideal for use in keeping track of calibration settings: after calibrating the robot once, it would not need to be recalibrated until after a new program were downloaded (in general, downloading of code will destroy the previous values of a persistent global, although this can be circumvented as explained in Section 7.4.3).

In order to use persistent globals, an initialization program would be created that allowed interactive setting of the values of the persistent variables. A menuing program could be written to use the two user buttons (`CHOOSE` and `ESCAPE`) and the “frob knob” to navigate around a series of menus. This program would allow the selection and modification or calibration of any of a number of parameters.

By exiting the initialization program without making any changes, or by simply not calling it at all, the robot can operate under the previous settings made to the persistent variables.

The routine could also allow restoration of the default values of all of the globals, returning them to some tested and known-to-work values.

## 6.3 Robot Control

The section will present some ideas about designing software that will control a robot. The perspective is not on low-level coding issues, but high level concepts about the special situations robots will encounter, and ways to address these peculiarities. The approach taken is to propose and examine some architectures of the control software that will be the brains of the robot.

Probably the biggest problem facing a robot is the flakiness of the overall system. A robot might face any combination of the following failure modes:

- **Mechanical Failures.** These might range from temporarily jammed movements to wedged geartrains to serious mechanical breakdown.
- **Electrical Failures.** Hopefully one can assume that the computer itself will not fail, but loose connections of motors and sensors are a common problem.
- **Sensor Unreliability.** Sensors will provide noisy data (data this is sometimes accurate but sometimes not) or simply data that is incorrect (a touch sensor might fail to be triggered).

The first two of these problems can be minimized with careful design, but the third category, sensor unreliability, is worth looking at more closely. Let's delve into a brief analysis of the sensor problem and then return to discuss control ideas.

### 6.3.1 Sensor Unreliability

A variety of problems afflict typical robot sensors:

**Spurious Sensor Data.** Most sensors will occasionally generate noise in their output. For example, an infrared sensor might indicate the presence of infrared light when there actually is no light present. Or a bend sensor might give a spurious reading.

If the noise is predictable enough, it can be filtered out in software. The noisy IR sensor might not be trusted until it gives some number of consecutive readings in agreement with one another.

However, if the noise problem is very bad, a sensor might be rendered useless, or worse, dangerous, if the program running the robot imbues too much trust in the sensor reading.

**Missed Sensor Data.** Affiliated with the problem of noisy data is missed data. Suppose, for either electrical or software reasons, a sensor reading is not detected. A light sensor changes state twice before the software can count it. Or a touch sensor jams and fails to trigger.

**Corrupted Sensor Data.** As discussed in the previous section on calibration, sensor data can be adversely affected by ambient environmental conditions or battery strength.

To some extent, unruly sensor data can be filtered or otherwise processed “at the source,” that is, before higher-level control routines see it. We'll use the `wall_distance()` introduced at the beginning of the chapter as an example.

The wall distance routine got its data directly from the bend sensor and then outputted an interpretation of that data. The routine did not process the sensor data in any way—it did not check it for unreasonable data samples, for example. Suppose that the bend sensor should never report a value above 250, and for some reason, a bogus value is detected. This probably indicates some type of sensor failure—maybe the sensor is unplugged. It would make sense to intercept this failure locally, where the sensor data is first entered into the software system.

In a similar way, sensor data could be averaged, smoothed, or otherwise processed before interpretation. It is logical to assign individual routines to perform this activity for variety of sensor, which might need to be dealt with in a particular way. Using the multi-tasking capabilities of IC, each sensor or sensor sub-system could be assigned its own C process to perform this activity.

### 6.3.2 Task-Oriented Control

With all of these problems facing a robot, how can it get anything done? Usually—and this is not a tart response—by assuming that they don't exist. The insidious part is that most of the time, ignoring the failure modes will work. However, when the failures do occur, they will return to inflict crippling damage to a robot's performance.

Let's return again to the wall-following example, as implemented by the function `follow_wall()` (see Figure 6.3). In a worst-case scenario, what could happen while a robot was merrily running along, following a wall?

1. The robot could run into an object or a corner, properly triggering a touch sensor.
2. The robot could run into an object or corner, *not* triggering a touch sensor.
3. The robot could wander off away from the wall.
4. The robot could slam into the wall, get stuck, and conditionally trigger a touch sensor.
5. The bend sensor could fall off its mount, causing a series of incorrect sensor readings.
6. Other failures of a catastrophic nature.

Ideally, control software should *expect* occurrences of cases like those numbered #1 through #4 and be able to *detect* case #5. Probably there isn't much that can be done about case #6, catastrophic failure.

Suppose we treat the wall-following activity as a discrete *robot task*, with initial conditions, an activity to perform (perhaps repetitively), exit conditions, and a return

value. If we organize our thinking about robot activity with these categories, we would analyze the wall-following function as follows:

**Task Analysis of Simple Wall Follow Function.**

**Initial conditions.** Robot is in close proximity to a wall near its bend sensor.

**Activity.** Feedback loop with three states based on wall sensor reading.

**Exit conditions.** None; never exits.

**Return value.** None, even if it did exit.

### Exit Conditions

Using this framework, we can to extend the simple wall-following function so that it could deal with several of the problems it might face while following a wall. Some of these “problems” actually *must* be dealt with; if a robot doesn’t run into an obstacle sooner or later, either something is wrong, unless the robot is following a very long (circular?) wall!

```
int NORMAL_EXIT= 0;      /* exit condition code */

int follow_wall2()
{
    while (!robot_stuck()) {
        int state= wall_distance();

        if (state == TOO_CLOSE) veer_away_from_wall();
        else if (state == TOO_FAR) veer_toward_wall();
        else drive_straight();
    }

    return NORMAL_EXIT;
}
```

Figure 6.6: Wall Following Function with Exit Condition

By adding a test for touch sensors inside the loop code of `follow_wall()`, we can create a function that will detect collisions and exit. Our new function, `follow_wall2()`, is shown in Figure 6.6. Note the new sensor function `robot_stuck()`, which is expected to return a boolean true if it believes that the robot is stuck. This function can use any of the sensors available to the robot—including touch sensors, shaft encoders, and motor force sensing—to determine if this is the case.

## Timeouts

Detecting collisions can only be as good as one's collision sensors. Based on the premise that such sensors are not perfect, it is a good idea to add some kind of timer-based exit condition. This will prevent the all-too-frequent case of a robot that is stuck, but does not have its touch sensor depressed. Many a robot simply does not “believe” that it is stuck—its program stays stuck in some loop, taking no compensatory action.

Time-outs can solve this problems, and provide other information as well. In a typical application, the *maximum time* that the robot is allowed to take in performing a particular task would be determined. When the function to perform the task is invoked, it would be given this maximum time. If (continuing the example) the wall-following task failed to exit before the time limit had expired, the timeout would trigger, and cause the function to exit (with an appropriate error return value).

Further, the timing information could be used to *verify* that the task had exited normally. Suppose a robot should take six seconds to get from the start of the wall-follow to running into a particular wall. If it ended up taking only three seconds, then probably an obstacle caused the premature exit.

Figure 6.7 is a listing of our third wall-following function, with timeout capability. Note that the timing variables are defined in long integers (with timing units of milliseconds). Floating point variables could also have been used (with the more intuitive units of seconds), but long integers are significantly more efficient.

If we do a task analysis of the `follow_wall13()` function, we will see a much better set of specifications, as shown in Figure 6.8.

## Monitoring State Transitions inside a Feedback Loop

With the third version of the wall-following function, we can be sure that the robot will not wedge and get stuck forever. However, with a little more analysis, we can create a program to detect failure situations in advance of the overall task timeout.

The key is in the guts of the feedback loop, associated with the functions `veer_away_from_wall()`, `veer_toward_wall()`, and `drive_straight()`. When the robot is following a wall normally, these functions should alternate control, each being operative for only a short period of time. Said another way: the robot will not simply drive straight for a long time: it will veer into the wall for a bit, veer away from the wall for a bit, drive straight for a bit, etc. Contrarily, if the robot wandered away from the wall, then the `veer_toward_wall()` output would be continuously asserted.

By monitoring for normal exchange of control amongst these feedback outputs, we can detect normal operation of the feedback loop. Conversely, by looking for abnormal exchange of control—in particular, one output being asserted for too long a period of time—we can detect failure conditions.

```
int NORMAL_EXIT= 0;    /* exit condition codes */
int TIMEOUT_EXIT= 1;
int PREMATURE_EXIT= 2;

/*
arguments to function are:
    timeout value
    maximum normal exit time
*/

int follow_wall3(long timeout, long normal_exit_max)
{
    long entry_time= mseconds();
    long exit_timeout, normal_exit_timeout, exit_time;

    exit_timeout= entry_time + timeout;
    normal_exit_timeout= entry_time + normal_exit_max;

    while (!robot_stuck() && (mseconds() < exit_timeout)) {
        int state= wall_distance();

        if (state == TOO_CLOSE) veer_away_from_wall();
        else if (state == TOO_FAR) veer_toward_wall();
        else drive_straight();
    }
    exit_time= mseconds();

    if (exit_time > exit_timeout) return TIMEOUT_EXIT;
    if (exit_time < normal_exit_timeout) return PREMATURE_EXIT;

    return NORMAL_EXIT;
}
```

Figure 6.7: Wall Following Function with Exit and Timeout Conditions

<p><b>Initial conditions.</b> Robot is in close proximity to a wall near its bend sensor.</p> <p><b>Input values.</b> Maximum length of time to perform wall-follow, normal length of time for wall-follow.</p> <p><b>Activity.</b> Feedback loop with three states based on wall sensor reading.</p> <p><b>Exit conditions.</b> “Robot stuck” function. Timeout.</p> <p><b>Return value.</b> Normal exit, timeout error exit, premature error exit.</p>
--

Figure 6.8: Task Analysis of Wall Follow Function with Sensing and Timeout Capability.

The code to implement this idea works as follows: each time a new feedback output is selected, a timer is reset. The timer keeps track of how much time is spent in consecutive selections of the feedback output. If the same feedback output is selected repeatedly for too long a period of time, an exit error condition is generated.

Several constants are used to adjust the parameters of the timeout: `DRIVE-STRAIGHT_MAXTIME`, `VEER_IN_MAXTIME`, and `VEER_OUT_MAXTIME`. Three new exit error conditions are used to report which part of the feedback loop failed. The final program, `follow_wall4()`, is shown in Figures 6.9 and 6.10.

There is an annoying possibility when using feedback state monitoring that should be noted. It is conceivable that for some duration of time that it would be *correct* for the loop to stay in one state for an unusually long period of time, in which case method would incorrectly cause a premature exit. In our wall-following example, perhaps if the robot were to drive very straight and were oriented exactly parallel to the wall, it would be proper to stay in the “drive straight” state for a long while.

Upping the timeout values of the constants would minimize the potential of this problem, but would also diminish the effectiveness of the method. It is probably best to deal with these circumstances on a case-by-case basis. One possibility is to deliberately handicap the feedback control so that it oscillates a bit; clearly this has disadvantages too.

### 6.3.3 Coordination of Tasks

The robot task model just presented should prove useful as a way to make a robot’s behavior more reliable. Yet, a further question arises: How should the selection and execution of different tasks be done?

This question is often asked by contemporary robot researchers. In addition to a

```

int NORMAL_EXIT=      0;      /* exit condition code */
int TIMEOUT_EXIT=    1;
int PREMATURE_EXIT=  2;
int DRIVE_STRAIGHT_EXIT= 3;
int VEER_IN_STUCK_EXIT= 4;
int VEER_OUT_STUCK_EXIT= 5;

long DRIVE_STRAIGHT_MAXTIME= 3000L;      /* values in milliseconds */
long VEER_IN_MAXTIME=      3000L;
long VEER_OUT_MAXTIME=    3000L;

/*
arguments to function are:
timeout value
maximum normal exit time
*/

int follow_wall4(long timeout, long normal_exit_max)
{
long entry_time= mseconds();
long exit_timeout, normal_exit_timeout, exit_time;

long state_start;      /* keeps track of when a state was
                        first selected */
int feedback_state= 0; /* keeps track of which state is being
                        timed */

exit_timeout= entry_time + timeout;
normal_exit_timeout= entry_time + normal_exit_max;

while (!robot_stuck() && (mseconds() < exit_timeout)) {
int state= wall_distance();

if (state == TOO_CLOSE) {

/* check for timeout */
if (feedback_state == VEER_IN_EXIT) /* reuse error codes
as state codes */
if (mseconds() - state_start > VEER_IN_MAXTIME)
return feedback_state; /* exit task with error code */
else {
/* set state, state_start variables */
feedback_state= VEER_IN_EXIT;
state_start= mseconds();
}

/* generate movement command */
veer_away_from_wall();
}
}
}

```

Figure 6.9: Wall Following Function with Exit, Timeout, and Feedback Monitoring Conditions, Page 1

```

else if (state == TOO_FAR) {
    if (feedback_state == VEER_OUT_EXIT)
        if (mseconds() - state_start > VEER_OUT_MAXTIME)
            return feedback_state;
        else {
            feedback_state= VEER_OUT_EXIT;
            state_start= mseconds();
        }
    veer_toward_wall();
}
else {
    if (feedback_state == DRIVE_STRAIGHT_EXIT)
        if (mseconds() - state_start > DRIVE_STRAIGHT_MAXTIME)
            return feedback_state;
        else {
            feedback_state= DRIVE_STRAIGHT_EXIT;
            state_start= mseconds();
        }
    drive_straight();
}
}

exit_time= mseconds();
if (exit_time > exit_timeout) return TIMEOUT_EXIT;
if (exit_time > normal_exit_timeout) return PREMATURE_EXIT;
return NORMAL_EXIT;
}

```

Figure 6.10: Wall Following Function with Exit, Timeout, and Feedback Monitoring Conditions, Page 2

variety of ways of thinking about robot tasks, there are many different approaches to organizing the higher-level control of mobile robots.

6.270 robots have special requirements in comparison to robots used more generally in research. 6.270 robots must be fast; many research robots can sit and compute for a while. 6.270 robots must be reliable; for other demonstrations, a robot might be videotaped until it performs as desired. 6.270 robots have only a few chances to perform correctly for the contest; in some research experiments, software robots are “evolved” through many generations until they behave appropriately.

Still, some of the ideas from the research field may be helpful. To extend from the task model developed in this chapter, here are several different approaches that could be used to coordinate and control task execution.

### **Task Sequencing**

In this model, only one task executes at a time. A “task manager” is responsible for selecting tasks based upon a predetermined sequence of tasks, with alternative sequences to deal with exceptional circumstances. This can be visualized as a connected graph of tasks, with the path of traversal determined by the task manager.

A simple example of task sequencing would be a program to make a robot follow the inside wall of a rectangular area. The task manager would alternately invoke “follow wall” and “negotiate corner” tasks, assuming there were no errors.

### **Concurrent and Non-Competing Tasks**

This model builds on the task sequencing model by allowing concurrent execution of tasks that are essentially non-interfering. For example, a task to control a “radar dish” sensor (that locates sources of infrared light) can be operated independently from a task that drives the robot. There may be some communication between the tasks (the radar dish task may wish to know that the robot’s base is moving), but there is no direct interference or need for coordination between the tasks.

### **Concurrent and Competing Tasks**

In the most general situation, concurrent tasks might interfere with each other, or compete for resources on the robot (such as control of drive motors or control of an active sensor). In this case, some method, either explicit or implicit, must be devised to resolve resource conflicts.

One method is to provide each task with a priority level; if two or more tasks are competing for the same resource, the task with the highest priority would win. A method for dealing with ties would be needed as well.

### Robot Metacognition

A sophisticated task manager might have a separate module that acts as its “overseer.” To use Marvin Minsky’s idea and terminology from his *Society of Mind*, the main part of our brains (the “A brain”) might be observed by a separate part of the brain (the “B brain”).

The B brain, or overseer, checks the A-brain (which is mostly in control) for things like non-productive loops and other wedged conditions. If it detects one of these undesirable states, it makes an intervention that will provoke a different response from the A brain.

Here’s an example to bring this metaphor back to our robots. Suppose a task manager (of the sequencer variety) is trying to drive the robot around the inside of a rectangle, as suggested earlier. It is alternating between two tasks: a “follow wall” task and a “negotiate corner” task. But suppose the corner routine is failing, and unbeknownst to the sequencer, the robot is stuck in the same corner.

The robot’s “B-brain controller” might notice an especially tight loop between the execution of the two tasks (much the same as the wall-follower would notice trouble in shifting between the feedback outputs). The B-brain would conclude that something had gone wrong, and execute an emergency “get unwedged” task.

### Control of a 6.270 Robot

In the past, most 6.270 robots have been designed with the “task sequencer” model in mind. There have been some cases of concurrent, non-competing tasks, but few or no examples of concurrent and competing tasks.

Rather than advocating a specific method, it is left for the reader to think about these issues and decide what is best for his or her own robot. Some of these methods were developed to make robots more “creature-like,” which is not necessarily a desirable characteristic for a 6.270 robot. It would not be ideal for a robot to suddenly decide that it did not really want to follow that wall or whatever. A task sequencing method, perhaps with a few checks for unproductive loops, will probably be more than adequate for most robot designs.

# Chapter 7

## IC Manual

Interactive C (IC for short) is a C language consisting of a compiler (with interactive command-line compilation and debugging) and a run-time machine language module. IC implements a subset of C including control structures (`for`, `while`, `if`, `else`), local and global variables, arrays, pointers, 16-bit and 32-bit integers, and 32-bit floating point numbers.

IC works by compiling into pseudo-code for a custom stack machine, rather than compiling directly into native code for a particular processor. This pseudo-code (or *p-code*) is then interpreted by the run-time machine language program. This unusual approach to compiler design allows IC to offer the following design tradeoffs:

- **Interpreted execution** that allows run-time error checking and prevents crashing. For example, IC does array bounds checking at run-time to protect against programming errors.
- **Ease of design.** Writing a compiler for a stack machine is significantly easier than writing one for a typical processor. Since IC's p-code is machine-independent, porting IC to another processor entails rewriting the p-code interpreter, rather than changing the compiler.
- **Small object code.** Stack machine code tends to be smaller than a native code representation.
- **Multi-tasking.** Because the pseudo-code is fully stack-based, a process's state is defined solely by its stack and its program counter. It is thus easy to task-switch simply by loading a new stack pointer and program counter. This task-switching is handled by the run-time module, not by the compiler.

Since IC's ultimate performance is limited by the fact that its output p-code is interpreted, these advantages are taken at the expense of raw execution speed. Still, IC is no slouch.

*IC was designed and implemented by Randy Sargent with the assistance of Fred Martin.*

## 7.1 Getting Started

This section describes how to boot IC on the 6.270 board using the MIT Athena computer network. Commands that are typed to the computer are shown underlined for visibility.

1. **Add the 6.270 directory to the execution path.** Type the following command at the Unix prompt:

add 6.270

2. **Plug the board into the computer.** Using the modular phone cable, plug the modified end into the DEC or VAX's printer port (indicated by an icon of a computer printer). *Make sure that the exposed wires of the plug are facing up.* If using the VAX2000 computer, use the 9-pin plug adapter.

Plug the other end into the modular jack on the 6811 board. *Before the board is turned on*, check that the board's green LED (labelled SER RCV) is lit. If it is not lit, there is a problem with the connection.

3. **Initialize the board.** The first step in using IC is to load the run-time module (called the "p-code program") into the board. If the p-code is already loaded, this step may be skipped. If not:

- Switch the board on. *Do not* hit the reset button at this time. When the board is switched on, the yellow LED (labelled SER XMIT) should flash briefly and then stay off. *If the yellow LED is lit, the board is not ready to be initialized. Turn the board off and on and try again.*

- From the Unix prompt, type

init\_bd

A process should begin that downloads the p-code program to the board. This will take about 15 to 30 seconds to complete. If the program exits with an error message, check the connection and try again.

4. **Reset the board.** Press the reset button on the board to reset it. The following should happen:

- (a) The board will emit a brief beep;
- (b) A version message will be printed on the LCD screen (e.g., "IC vX.XX");

(c) The yellow LED will turn on brightly.

If these things do not happen, repeat step 3 to initialize the board.

5. **Begin IC.** From the Unix prompt, type:

IC

At this point, IC will boot, ready to load a C program or evaluate expressions typed to the IC prompt.

## 7.2 Using IC

IC is started from the Unix shell by typing `ic` at the prompt. Some Unix systems (in particular, MIT Athena DECstations) have an unrelated application named `ic`. If this application is first in the execution path, it will be invoked rather than the IC compiler. This situation may be remedied by reordering the execution path to include the path to the IC compiler first, or by using the program name `icc`, which will also invoke IC.

IC can be started with the name (or names) of a C file to compile.

When running and attached to a 6811 system, C expressions, function calls, and IC commands may be typed at the “C>” prompt.

All C expressions must be ended with a semicolon. For example, to evaluate the arithmetic expression  $1 + 2$ , type the following:

```
C> 1 + 2;
```

When this expression is typed, it is compiled by the console computer and then downloaded to the 6811 system for evaluation. The 6811 then evaluates the compiled form and returns the result, which is printed on the console computer’s screen.

To evaluate a series of expressions, create a C block by beginning with an open curly brace “{” and ending with a close curly brace “}”. The following example creates a local variable `i` and prints the sum `i+7` to the 6811’s LCD screen:

```
C> {int i=3; printf("%d", i+7);}
```

### 7.2.1 IC Commands

IC responds to the following commands:

- **Load file.** The command `load <filename>` compiles and loads the named file. The board must be attached for this to work. IC looks first in the local directory and then in the IC library path for files.  
Several files may be loaded into IC at once, allowing programs to be defined in multiple files.
- **Unload file.** The command `unload < filename >` unloads the named file, and re-downloads remaining files.
- **List files, functions, or globals.** The command `list files` displays the names of all files presently loaded into IC. The command `list functions` displays the names of presently defined C functions. The command `list globals` displays the names of all currently defined global variables.
- **Kill all processes.** The command `kill_all` kills all currently running processes.
- **Print process status.** The command `ps` prints the status of currently running processes.
- **Edit a file.** The command `edit <filename>` brings up a system editor to allow editing of a file. This command is most useful on single-tasking operating systems, like MS-DOS.
- **Run an inferior shell.** If IC is running on a MS-DOS system, this command opens a shell to execute MS-DOS functions.
- **Help.** The command `help` displays a help screen of IC commands.
- **Quit.** The command `quit` exits IC. CTRL-C can also be used.

### 7.2.2 Line Editing

IC has a built-in line editor and command history, allowing editing and re-use of previously typed statements and commands. The mnemonics for these functions are based on standard Emacs control key assignments.

To scan forward and backward in the command history, type CTRL-P or ↑ for backward, and CTRL-N or ↓ for forward.

An earlier line in the command history can be retrieved by typing the exclamation point followed by the first few characters of the line to retrieve, and then the space bar.

Figure 7.1 shows the keystroke mappings understood by IC.

IC does parenthesis-balance-highlighting as expressions are typed.

Keystroke	Function
<b>DEL</b>	backward-delete-char
CTRL-A	beginning-of-line
CTRL-B	backward-char
<b>←</b>	backward-char
CTRL-D	delete-char
CTRL-E	end-of-line
CTRL-F	forward-char
<b>→</b>	forward-char
CTRL-K	kill-line
CTRL-U	universal-argument
ESC D	kill-word
ESC <b>DEL</b>	backward-kill-word

Figure 7.1: IC Command-Line Keystroke Mappings

### 7.2.3 The `main()` Function

After functions have been downloaded to the board, they can be invoked from the IC prompt. If one of the functions is named `main()`, it will automatically be run when the board is reset.

To reset with board *without* running the `main()` function (for instance, when hooking the board back to the computer), hold down one of the two user input buttons on the board while pressing reset. The board will reset without running `main()`.

## 7.3 A Quick C Tutorial

Most C programs consist of function definitions and data structures. Here is a simple C program that defines a single function, called `main`.

```
void main()
{
    printf("Hello, world!\n");
}
```

All functions must have a return value; that is, the value that they return when they finish execution. `main` has a return value type of `void`, which is the “null” type.

Other types include integers (`int`) and floating point numbers (`float`). This *function declaration* information must precede each function definition.

Immediately following the function declaration is the function's name (in this case, `main`). Next, in parentheses, are any arguments (or inputs) to the function. `main` has none, but a empty set of parentheses is still required.

After the function arguments is an open curly-brace “{”. This signifies the start of the actual function code. Curly-braces signify program *blocks*, or chunks of code.

Next comes a series of C *statements*. Statements demand that some action be taken. Our demonstration program has a single statement, a `printf` (formatted print). This will print the message “Hello, world!” to the LCD display. The `\n` indicates end-of-line.

The `printf` statement ends with a semicolon (“;”). All C statements must be ended by a semicolon. Beginning C programmers commonly make the error of omitting the semicolon that is required at the end of each statement.

The `main` function is ended by the close curly-brace “}”.

Let's look at an another example to learn some more features of C. The following code defines the function *square*, which returns the mathematical square of a number.

```
int square(int n)
{
    return n * n;
}
```

The function is declared as type `int`, which means that it will return an integer value. Next comes the function name `square`, followed by its argument list in parenthesis. `square` has one argument, `n`, which is an integer. Notice how declaring the type of the argument is done similarly to declaring the type of the function.

When a function has arguments declared, those argument variables are valid within the “scope” of the function (i.e., they only have meaning within the function's own code). Other functions may use the same variable names independently.

The code for `square` is contained within the set of curly braces. In fact, it consists of a single statement: the `return` statement. The `return` statement exits the function and returns the value of the C *expression* that follows it (in this case “`n * n`”).

Expressions are evaluated according set of precedence rules depending on the various operations within the expression. In this case, there is only one operation (multiplication), signified by the “\*”, so precedence is not an issue.

Let's look at an example of a function that performs a function call to the `square` program.

```
float hypotenuse(int a, int b)
{
```

```
float h;
h = sqrt((float)(square(a) + square(b)));
return h;
}
```

This code demonstrates several more features of C. First, notice that the floating point variable `h` is defined at the beginning of the `hypotenuse` function. In general, whenever a new program block (indicated by a set of curly braces) is begun, new local variables may be defined.

The value of `h` is set to the result of a call to the `sqrt` function. It turns out that `sqrt` is a built-in function that takes a floating point number as its argument.

We want to use the `square` function we defined earlier, which returns its result as an integer. But the `sqrt` function requires a floating point argument. We get around this type incompatibility by *coercing* the integer sum (`square(a) + square(b)`) into a float by preceding it with the desired type, in parentheses. Thus, the integer sum is made into a floating point number and passed along to `sqrt`.

The `hypotenuse` function finishes by returning the value of `h`.

This concludes the brief C tutorial.

## 7.4 Data Types, Operations, and Expressions

Variables and constants are the basic data objects in a C program. Declarations list the variables to be used, state what type they are, and may set their initial value. Operators specify what is to be done to them. Expressions combine variables and constants to create new values.

### 7.4.1 Variable Names

Variable names are case-sensitive. The underscore character is allowed and is often used to enhance the readability of long variable names. C keywords like `if`, `while`, etc. may not be used as variable names.

Global variables and functions may not have the same name. In addition, local variables named the same as functions prevent the use of that function within the scope of the local variable.

### 7.4.2 Data Types

IC supports the following data types:

**16-bit Integers** 16-bit integers are signified by the type indicator `int`. They are signed integers, and may be valued from  $-32,768$  to  $+32,767$  decimal.

**32-bit Integers** 32-bit integers are signified by the type indicator `long`. They are signed integers, and may be valued from  $-2,147,483,648$  to  $+2,147,483,647$  decimal.

**32-bit Floating Point Numbers** Floating point numbers are signified by the type indicator `float`. They have approximately seven decimal digits of precision and are valued from about  $10^{-38}$  to  $10^{38}$ .

**8-bit Characters** Characters are an 8-bit number signified by the type indicator `char`. A character's value typically represents a printable symbol using the standard ASCII character code.

Arrays of characters (character strings) are supported, but individual characters are not.

### 7.4.3 Local and Global Variables

If a variable is declared within a function, or as an argument to a function, its binding is *local*, meaning that the variable has existence only that function definition.

If a variable is declared outside of a function, it is a global variable. It is defined for all functions, including functions that are defined in files other than the one in which the global variable was declared.

#### Variable Initialization

Local and global variables can be initialized when they are declared. If no initialization value is given, the variable is initialized to zero.

```
int foo()
{
    int x;          /* create local variable x
                   with initial value 0    */
    int y= 7;      /* create local variable y
                   with initial value 7   */
    ...
}

float z=3.0;      /* create global variable z
                  with initial value 3.0 */
```

Local variables are initialized whenever the function containing them runs.

Global variables are initialized whenever a reset condition occurs. Reset conditions occur when:

1. New code is downloaded;
2. The `main()` procedure is run;
3. System hardware reset occurs.

### Persistent Global Variables

A special *uninitialized* form of global variable, called the “persistent” type, has been implemented for IC. A persistent global is *not* initialized upon the conditions listed for normal global variables.

To make a persistent global variable, prefix the type specifier with the key word `persistent`. For example, the statement

```
persistent int i;
```

creates a global integer called `i`. The initial value for a persistent variable is arbitrary; it depends on the contents of RAM that were assigned to it. Initial values for persistent variables cannot be specified in their declaration statement.

Persistent variables keep their state when the robot is turned off and on, when `main` is run, and when system reset occurs. Persistent variables, in general, will lose their state when a new program is downloaded. However, it is possible to prevent this from occurring. If persistent variables are declared at the beginning of the code, before any function or non-persistent globals, they will be re-assigned to the same location in memory when the code is re-compiled, and thus their values will be preserved over multiple downloads.

If the program is divided into multiple files and it is desired to preserve the values of persistent variables, then all of the persistent variables should be declared in one particular file and that file should be placed first in the load ordering of the files.

Persistent variables were created with two applications in mind:

- Calibration and configuration values that do not need to be re-calculated on every reset condition.
- Robot learning algorithms that might occur over a period when the robot is turned on and off.

#### 7.4.4 Constants

##### Integers

Integers may be defined in decimal integer format (e.g., `4053` or `-1`), hexadecimal format using the “`0x`” prefix (e.g., `0x1fff`), and a non-standard but useful binary format using the “`0b`” prefix (e.g., `0b1001001`). Octal constants using the zero prefix are not supported.

## Long Integers

Long integer constants are created by appending the suffix “`l`” or “`L`” (upper- or lower-case alphabetic `L`) to a decimal integer. For example, `0L` is the long zero. Either the upper or lower-case “`L`” may be used, but upper-case is the convention for readability.

## Floating Point Numbers

Floating point numbers may use exponential notation (e.g., “`10e3`” or “`10E3`”) or must contain the decimal period. For example, the floating point zero can be given as “`0.`”, “`0.0`”, or “`0E1`”, but not as just “`0`”.

## Characters and Character Strings

Quoted characters return their ASCII value (e.g., ‘`x`’).

Character strings are defined with quotation marks, e.g., “`This is a character string.`”.

### 7.4.5 Operators

Each of the data types has its own set of operators that determine which operations may be performed on them.

## Integers

The following operations are supported on integers:

- **Arithmetic.** addition `+`, subtraction `-`, multiplication `*`, division `/`.
- **Comparison.** greater-than `>`, less-than `<`, equality `==`, greater-than-equal `>=`, less-than-equal `<=`.
- **Bitwise Arithmetic.** bitwise-OR `|`, bitwise-AND `&`, bitwise-exclusive-OR `^`, bitwise-NOT `~`.
- **Boolean Arithmetic.** logical-OR `||`, logical-AND `&&`, logical-NOT `!`.

When a C statement uses a boolean value (for example, `if`), it takes the integer zero as meaning false, and any integer other than zero as meaning true. The boolean operators return zero for false and one for true.

Boolean operators `&&` and `||` stop executing as soon as the truth of the final expression is determined. For example, in the expression `a && b`, if `a` is false, then `b` does not need to be evaluated because the result must be false. The `&&` operator “knows this” and does not evaluate `b`.

### Long Integers

A subset of the operations implemented for integers are implemented for long integers: arithmetic addition `+`, subtraction `-`, and multiplication `*`, and the integer comparison operations. Bitwise and boolean operations and division are not supported.

### Floating Point Numbers

IC uses a package of public-domain floating point routines distributed by Motorola. This package includes arithmetic, trigonometric, and logarithmic functions.

The following operations are supported on floating point numbers:

- **Arithmetic.** addition `+`, subtraction `-`, multiplication `*`, division `/`.
- **Comparison.** greater-than `>`, less-than `<`, equality `==`, greater-than-equal `>=`, less-than-equal `<=`.
- **Built-in Math Functions.** A set of trigonometric, logarithmic, and exponential functions is supported, as discussed in Section 7.10 of this document.

### Characters

Characters are only allowed in character arrays. When a cell of the array is referenced, it is automatically coerced into an integer representation for manipulation by the integer operations. When a value is stored into a character array, it is coerced from a standard 16-bit integer into an 8-bit character (by truncating the upper eight bits).

## 7.4.6 Assignment Operators and Expressions

The basic assignment operator is `=`. The following statement adds 2 to the value of `a`.

```
a = a + 2;
```

The abbreviated form

```
a += 2;
```

could also be used to perform the same operation.

All of the following binary operators can be used in this fashion:

```
+ - * / % << >> & ^ |
```

### 7.4.7 Increment and Decrement Operators

The increment operator “++” increments the named variable. For example, the statement “a++” is equivalent to “a= a+1” or “a+= 1”.

A statement that uses an increment operator has a value. For example, the statement

```
a= 3;
printf("a=%d a+1=%d\n", a, ++a);
```

will display the text “a=3 a+1=4.”

If the increment operator comes after the named variable, then the value of the statement is calculated *after* the increment occurs. So the statement

```
a= 3;
printf("a=%d a+1=%d\n", a, a++);
```

would display “a=3 a+1=3” but would finish with a set to 4.

The decrement operator “--” is used in the same fashion as the increment operator.

### 7.4.8 Precedence and Order of Evaluation

The following table summarizes the rules for precedence and associativity for the C operators. Operators listed earlier in the table have higher precedence; operators on the same line of the table have equal precedence.

Operator	Associativity
() []	left to right
! ~ ++ -- - ( <i>type</i> )	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
	left to right
&&	left to right
	right to left
= += -= etc.	right to left
,	left to right

## 7.5 Control Flow

IC supports most of the standard C control structures. One notable exception is the `case` and `switch` statement, which is not supported.

### 7.5.1 Statements and Blocks

A single C statement is ended by a semicolon. A series of statements may be grouped together into a *block* using curly braces. Inside a block, local variables may be defined.

There is never a semicolon after a right brace that ends a block.

### 7.5.2 If-Else

The `if else` statement is used to make decisions. The syntax is:

```
if ( expression )
    statement-1
else
    statement-2
```

*expression* is evaluated; if it is not equal to zero (e.g., logic true), then *statement-1* is executed.

The `else` clause is optional. If the `if` part of the statement did not execute, and the `else` is present, then *statement-2* executes.

### 7.5.3 While

The syntax of a `while` loop is the following:

```
while ( expression )
    statement
```

`while` begins by evaluating *expression*. If it is false, then *statement* is skipped. If it is true, then *statement* is evaluated. Then the expression is evaluated again, and the same check is performed. The loop exits when *expression* becomes zero.

One can easily create an infinite loop in C using the `while` statement:

```
while (1)
    statement
```

### 7.5.4 For

The syntax of a `for` loop is the following:

```
for ( expr-1 ; expr-2 ; expr-3 )  
    statement
```

This is equivalent to the following construct using `while`:

```
expr-1 ;  
while ( expr-2 ) {  
    statement  
    expr-3 ;  
}
```

Typically, *expr-1* is an assignment, *expr-2* is a relational expression, and *expr-3* is an increment or decrement of some manner. For example, the following code counts from 0 to 99, printing each number along the way:

```
int i;  
for (i= 0; i < 100; i++)  
    printf("%d\n", i);
```

### 7.5.5 Break

Use of the `break` provides an early exit from a `while` or a `for` loop.

## 7.6 LCD Screen Printing

IC has a version of the C function `printf` for formatted printing to the LCD screen.

The syntax of `printf` is the following:

```
printf( format-string , [ arg-1 ] , ... , [ arg-N ] )
```

This is best illustrated by some examples.

### 7.6.1 Printing Examples

**Example 1: Printing a message.** The following statement prints a text string to the screen.

```
printf("Hello, world!\n");
```

In this example, the format string is simply printed to the screen.

The character “\n” at the end of the string signifies *end-of-line*. When an end-of-line character is printed, the LCD screen will be cleared when a subsequent character is printed. Thus, most `printf` statements are terminated by a `\n`.

**Example 2: Printing a number.** The following statement prints the value of the integer variable `x` with a brief message.

```
printf("Value is %d\n", x);
```

The special form `%d` is used to format the printing of an integer in decimal format.

**Example 3: Printing a number in binary.** The following statement prints the value of the integer variable `x` as a binary number.

```
printf("Value is %b\n", x);
```

The special form `%b` is used to format the printing of an integer in binary format. Only the *low byte* of the number is printed.

**Example 4: Printing a floating point number.** The following statement prints the value of the floating point variable `n` as a floating point number.

```
printf("Value is %f\n", n);
```

The special form `%f` is used to format the printing of floating point number.

**Example 5: Printing two numbers in hexadecimal format.**

```
printf("A=%x B=%x\n", a, b);
```

The form `%x` formats an integer to print in hexadecimal.

### 7.6.2 Formatting Command Summary

Format Command	Data Type	Description
<code>%d</code>	<code>int</code>	decimal number
<code>%x</code>	<code>int</code>	hexadecimal number
<code>%b</code>	<code>int</code>	low byte as binary number
<code>%c</code>	<code>int</code>	low byte as ASCII character
<code>%f</code>	<code>float</code>	floating point number
<code>%s</code>	<code>char array</code>	char array (string)

### 7.6.3 Special Notes

- The final character position of the LCD screen is used as a system “heartbeat.” This character continuously blinks back and forth when the board is operating properly. If the character stops blinking, the board has failed.
- Characters that would be printed beyond the final character position are truncated.
- When using a two-line display, the `printf()` command treats the display as a single longer line.
- Printing of long integers is not presently supported.

## 7.7 Arrays and Pointers

IC supports one-dimensional arrays of characters, integers, long integers, and floating-point numbers. Pointers to data items and arrays are supported.

### 7.7.1 Declaring and Initializing Arrays

Arrays are declared using the square brackets. The following statement declares an array of ten integers:

```
int foo[10];
```

In this array, elements are numbered from 0 to 9. Elements are accessed by enclosing the index number within square brackets: `foo[4]` denotes the fifth element of the array `foo` (since counting begins at zero).

Arrays are initialized by default to contain all zero values; arrays may also be initialized at declaration by specifying the array elements, separated by commas, within curly braces. Using this syntax, the size of the array would not be specified within the square braces; it is determined by the number of elements given in the declaration. For example,

```
int foo[] = {0, 4, 5, -8, 17, 301};
```

creates an array of six integers, with `foo[0]` equalling 0, `foo[1]` equalling 4, etc.

Character arrays are typically text strings. There is a special syntax for initializing arrays of characters. The character values of the array are enclosed in quotation marks:

```
char string[] = "Hello there";
```

This form creates a character array called `string` with the ASCII values of the specified characters. In addition, the character array is terminated by a zero. Because of this zero-termination, the character array can be treated as a string for purposes of printing (for example). Character arrays can be initialized using the curly braces syntax, but they will not be automatically null-terminated in that case. In general, printing of character arrays that are *not* null-terminated will cause problems.

### 7.7.2 Passing Arrays as Arguments

When an array is passed to a function as an argument, the array's pointer is actually passed, rather than the elements of the array. If the function modifies the array values, the array will be modified, since there is only one copy of the array in memory.

In normal C, there are two ways of declaring an array argument: as an array or as a pointer. IC only allows declaring array arguments as arrays.

As an example, the following function takes an index and an array, and returns the array element specified by the index:

```
int retrieve_element(int index, int array[])
{
    return array[index];
}
```

Notice the use of the square brackets to declare the argument `array` as an array of integers.

When passing an array variable to a function, use of the square brackets is not needed:

```
{
int array[10];

retrieve_element(3, array);
}
```

### 7.7.3 Declaring Pointer Variables

Pointers can be passed to functions which then go on to modify the value of the variable being pointed to. This is useful because the same function can be called to modify different variables, just by giving it a different pointer.

Pointers are declared with the use of the asterisk (\*). In the example

```
int *foo;
float *bar;
```

`foo` is declared as a pointer to an integer, and `bar` is declared as a pointer to a floating point number.

To make a pointer variable point at some other variable, the ampersand operator is used. The ampersand operator returns the *address* of a variable's value; that is, the place in memory where the variable's value is stored. Thus:

```
int *foo;
int x= 5;

foo= &x;
```

makes the pointer `foo` “point at” the value of `x` (which happens to be 5).

This pointer can now be used to retrieve the value of `x` using the asterisk operator. This process is called *de-referencing*. The pointer, or reference to a value, is used to fetch the value being pointed at. Thus:

```
int y;

y= *foo;
```

sets `y` equal to the value pointed at by `foo`. In the previous example, `foo` was set to point at `x`, which had the value 5. Thus, the result of dereferencing `foo` yields 5, and `y` will be set to 5.

### 7.7.4 Passing Pointers as Arguments

Pointers can be passed to functions; then, functions can change the values of the variables that are pointed at. This is termed *call-by-reference*; the reference, or pointer, to the variable is given to the function that is being called. This is in contrast to *call-by-value*, the standard way that functions are called, in which the value of a variable is given to the function being called.

The following example defines an `average_sensor` function which takes a port number and a pointer to an integer variable. The function will average the sensor and store the result in the variable pointed at by `result`.

In the code, the function argument is specified as a pointer using the asterisk:

```
void average_sensor(int port, int *result)
{
    int sum= 0;
    int i;

    for (i= 0; i< 10; i++) sum += analog(port);

    *result= sum/10;
}
```

Notice that the function itself is declared as a `void`. It does not need to return anything, because it instead stores its answer in the pointer variable that is passed to it.

The pointer variable is used in the last line of the function. In this statement, the answer `sum/10` is stored at the location pointed at by `result`. Notice that the asterisk is used to get the *location* pointed by `result`.

## 7.8 The IC Library File

Library files provide standard C functions for interfacing with hardware on the robot controller board. These functions are written either in C or as assembly language drivers. Library files provide functions to do things like control motors, make tones, and input sensors values.

IC automatically loads the library file every time it is invoked. Depending on which 6811 board is being used, a different library file will be required. IC may be configured to load different library files as its default; for the purpose of the 6.270 contest, the on-line version of IC will be configured appropriately for the board that is in use.

*As of this writing, there are three related 6811 systems in use: the 1991 6.270 Board (the “Revision 2” board), the 1991 Sensor Robot, and the 1992 6.270 Board (the “Revision 2.1” board). This writing covers the 1992 board only; documentation for the other two systems is available elsewhere.*

On the MIT Athena system, IC library files are located in the directory `/mit/-6.270/lib/ic`. (To understand better how the library functions work, study of the library file source code is recommended.) The main library file for the 1992 6.270 Board is named `lib_r21.lis`.

### 7.8.1 Output Control

#### DC Motors

Motor ports are numbered from 0 to 5; ports for motors 0 to 3 are located on the Microprocessor Board while motors 4 and 5 are located on the Expansion Board.

Motor may be set in a “forward” direction (corresponding to the green motor LED being lit) and a “backward” direction (corresponding to the motor red LED being lit).

The functions `fd(int m)` and `bk(int m)` turn motor `m` on or off, respectively, at full power. The function `off(int m)` turns motor `m` off.

The power level of motors may also be controlled. This is done in software by a motor on and off rapidly (a technique called *pulse-width modulation*). The `motor(int m, int p)` function allows control of a motor's power level. Powers range from 100 (full on in the forward direction) to -100 (full on the the backward direction). The system software actually only controls motors to seven degrees of power, but argument bounds of -100 and +100 are used.

```
void fd(int m)
```

Turns motor `m` on in the forward direction. Example: `fd(3);`

```
void bk(int m)
```

Turns motor `m` on in the backward direction. Example: `bk(1);`

```
void off(int m)
```

Turns off motor `m`. Example: `off(1);`

```
void alloff()
```

```
void ao()
```

Turns off all motors. `ao` is a short form for `alloff`.

```
void motor(int m, int p)
```

Turns on motor `m` at power level `p`. Power levels range from 100 for full on forward to -100 for full on backward.

## Servo Motor

A library routine allows control of a single servo motor. The servo motor has a three-wire connection: power, ground, and control. The power and ground of the servo motor connect to the power port labelled PWR OUT on the Microprocessor Board; the control wire plugs into the pin labelled D2 of the Port D I/O connector.

The position of the servo motor shaft is controlled by a rectangular waveform that is generated on the D2 pin. The duration of the positive pulse of the waveform determines the position of the shaft. This pulse repeats every 20 milliseconds.

The length of the pulse is determined by the value of a library variable.

```
int servo_pulse_wavetime
```

Library variable. Value is the time in half-microseconds of the positive portion of a rectangular wave that is generated on the Port D2 pin for use in controlling a servo motor.

Initial value is 2000, yielding a pulse time of one millisecond. Values between 1400 and 4000 (.7 to 2 milliseconds) are boundary settings for typical servo motors.

### Unidirectional Drivers

**LED Drivers** There are two output ports located on the Expansion Board that are suitable for driving LEDs or other small loads. These ports draw their power from the motor battery and hence will only work when that battery is connected.

The following commands are used to control the LED ports:

```
void led_out0(int s)
```

Turns on LED0 port if *s* is non-zero; turns it off otherwise.

```
void led_out1(int s)
```

Turns on LED1 port if *s* is non-zero; turns it off otherwise.

**Expansion Board Motor Ports** Motor ports 4 and 5, located on the Expansion Board, may also be used to control unidirectional devices, such as a solenoid, lamp, or a motor that needs to be driven in one direction only. Each of the two motor ports, when used in this fashion, can independently control two such devices.

To use the ports unidirectionally, the two-pin header directly beneath the motor 4 and 5 LEDs is used.

```
void motor4_left(int s)
```

Turns on left side of motor 4 port if *s* is non-zero; turns it off otherwise.

```
void motor4_right(int s)
```

Turns on right side of motor 4 port if *s* is non-zero; turns it off otherwise.

```
void motor5_left(int s)
```

Turns on left side of motor 5 port if *s* is non-zero; turns it off otherwise.

```
void motor5_right(int s)
```

Turns on right side of motor 5 port if *s* is non-zero; turns it off otherwise.

### 7.8.2 Sensor Input

```
int digital(int p)
```

Returns the value of the sensor in sensor port *p*, as a true/false value (1 for true and 0 for false).

Sensors are expected to be *active low*, meaning that they are valued at zero volts in the active, or true, state. Thus the library function returns the inverse of the actual reading from the digital hardware: if the reading is zero volts or logic zero, the `digital()` function will return true.

If the `digital()` function is applied to port that is implemented in hardware as an analog input, the result is true if the analog measurement is less than 127, and false if the reading is greater than or equal to 127.

Ports are numbered as marked on the Microprocessor Board and Expansion Board.

```
int analog(int p)
```

Returns value of sensor port numbered `p`. Result is integer between 0 and 255.

If the `analog()` function is applied to a port that is implemented digitally in hardware, then the value 0 is returned if the digital reading is 0, and the value 255 is returned if the digital reading is 1.

Ports are numbered as marked on the Microprocessor Board and Expansion Board.

```
int motor_force(int m)
```

Returns value of analog input sensing current level through motor `m`. Result is integer between 0 and 255, but typical readings range from about 40 (low force) to 100 (high force).

The force-sensing circuitry functions properly only when motors are operated at full speed. The circuit returns invalid results when motors are pulse-width modulated because of spikes that occur in the feedback path.

The force-sensing circuitry is implemented for motors 0 through 3.

```
int dip_switch(int sw)
```

Returns value of DIP switch `sw` on interface board. Switches are numbered from 1 to 4 as per labelling on actual switch. Result is 1 if the switch is in the position labelled “on,” and 0 if not.

```
int dip_switches()
```

Returns value on DIP switches as a four-bit binary number. Left-most switch is most significant binary digit. “On” position is binary one.

```
int choose_button()
```

Returns value of button labelled `CHOOSE`: 1 if pressed and 0 if released.  
Example:

```
/* wait until choose button pressed */
while (!left_button()) {}
```

```
int escape_button()
```

Returns value of button labelled ESCAPE.

Example:

```
/* wait for button to be pressed; then
   wait for it to be released so that
   button press is debounced          */
while (!escape_button()) {}
while (escape_button()) {}
```

### Infrared Subsystem

The infrared subsystem is composed of two parts: an infrared transmitter, and infrared receivers. Software is provided to control transmission frequency and detection of infrared light at two frequencies.

### Infrared Transmission

```
void ir_transmit_on()
```

Enables transmission of infrared light through IR OUT port.

```
void ir_transmit_off()
```

Disables transmission of infrared light through IR OUT port.

```
void set_ir_transmit_frequency(int period)
```

Sets infrared transmission frequency. `period` determines the delay in half-microseconds between transitions of the infrared waveform. If `period` is set to 10,000, a frequency of 100 Hz. will be generated. If `period` is set to 8,000, a frequency of 125 Hz. will be generated. The decoding software is capable of detecting transmissions on either of these two frequencies only.

Upon a reset condition, the infrared transmission frequency is set for 100 Hz. and is disabled.

**Infrared Reception** In a typical 6.270 application, one robot will be broadcasting infrared at 100 Hz. and will set its detection system for 125 Hz. The other robot will do the opposite. Each robot must physically shield its IR sensors from its own light; then each robot can detect the emissions of the other.

The infrared reception software employs a *phase-locked loop* to detect infrared signals modulated at a particular frequency. This program generates an internal squarewave at the desired reception frequency and attempts to lock this squarewave into synchronization with a waveform received by an infrared sensor. If the error

between the internal wave and the external wave is below some threshold, the external wave is considered “detected.” The software returns as a result the number of consecutive detections for each of the infrared sensor inputs.

Up to four infrared sensors may be used. These are plugged into positions 0 through 3 of the digital input port. These ports and the remainder of the digital input port may be used without conflict for standard digital input while the infrared detection software is operating.

The following library functions control the infrared detection system:

```
void ir_receive_on()
```

Enables the infrared reception software. The default is disabled. When the software is enabled, between 20% and 30% of the 6811 processor time will be spent performing the detection function; therefore it should only be enabled if it is being used.

```
void ir_receive_off()
```

Disables the infrared reception software.

```
void set_ir_receive_frequency(int f)
```

Sets the operating frequency for the infrared reception software. `f` should be 100 for 100 Hz. or 125 for 125 Hz. Default is 100.

```
int ir_counts(int p)
```

Returns number of consecutive squarewaves at operating frequency detected from port `p` of the digital input port. Result is number from 0 to 255. `p` must be 0, 1, 2, or 3.

Random noise can cause spurious readings of 1 or 2 detections. The return value of `ir_counts()` should be greater than three before it is considered the result of a valid detection.

## Shaft Encoders

Machine language drivers are provided to keep count of rapid transitions, as might occur on a shaft encoder sensor. Two types of shaft encoders are *optical*, in which a slotted wheel or black-and-white disk provides visual cues to an optosensor, and *magnetic*, in which a small magnet rotates past a magnetic sensor.

In either case, the task of the software consists of counting pulses. To count accurately, the software uses different thresholds for the rising and falling edge of a pulse. Hence the signal must rise above an upper threshold before being detecting as a valid “logic high” of the pulse, and must fall beneath a lower threshold before

being detected as a logic low. This method prevents the possibility that the signal might oscillate rapidly about a single threshold point. (The thresholds will probably need to be calibrated for the particular performance of each sensor arrangement.)

The software returns a total count of pulses, which may be reset by the user, and a velocity, consisting of the number of pulses recorded in the most recent 64 milliseconds. The software does *not* keep track of the direction of rotation of the shaft.

The software samples the sensor at the rate of 1000 Hz. Therefore the software cannot detect pulses more rapid than that frequency.

**Software Driver Files** The library functions for shaft encoders differ from most other library functions discussed in that they are *not* automatically loaded with the system library. These functions are stored in distinct files and must be explicitly loaded by the user when needed.

A separate driver program is implemented for ports 12 through 17 of the Expansion Board. The files containing the drivers are named `sencdr12.icb` through `sencdr17.icb` (for ports 12 through 17, respectively) and are located in the IC system library. These `icb` files must be loaded at the IC command line or from within a C program's `lis` file (as explained in Section 7.14) when needed.

Although the software drivers run in the background, each uses processing time, and only those which are needed should be loaded. For example, if shaft encoders are to be used on ports 12 and 13, then the files `sencdr12.icb` and `sencdr13.icb` only should be loaded.

**Shaft Encoder Routines** The operation of the shaft encoders is controlled mostly through global variables that are defined when the `icb` files are loaded. Instead of calling a library *function*, getting a measurement from a shaft encoder consists simply of examining a library *variable*.

The library routines and variables are named according to the port number they are designed for and are located in the correspondingly named file. The following explanation assumes the file `sencdr12.icb` has been loaded.

```
int encoder12_low_threshold
```

Library variable. Its value determines the logic low level of the pulsetrain being measured. Should be calibrated for the performance of actual sensor that is used. Default value is 10; must be between 0 and 255.

```
int encoder12_high_threshold
```

Library variable. Its value determines the logic high level of the pulsetrain being

measured. Should be calibrated for the performance of actual sensor that is used. Default value is 240; must be between 0 and 255.

Example:

```
/* set threshold points for shaft encoder
   on port 12 */
encoder12_low_threshold= 20;
encoder12_high_threshold= 40;
```

`int encoder12_counts`

Library variable. Continuously updated count of pulses. Each transition from high to low and low to high is registered as one count.

May be reset to zero simply by setting the variable to zero:

```
/* reset port 12 encoder counts to zero */
encoder12_counts= 0;
```

`int encoder12_velocity`

Library variable. Updated every 64 milliseconds as the number of counts recorded over the last 64 millisecond period.

### 7.8.3 Time Commands

System code keeps track of time passage in milliseconds. The time variables are implemented using the long integer data type. Standard functions allow use floating point variables when using the timing functions.

`void reset_system_time()`

Resets the count of system time to zero milliseconds.

`long mseconds()`

Returns the count of system time in milliseconds. Time count is reset by hardware reset (i.e., pressing reset switch on board) or the function `reset_system_time()`. `mseconds()` is implemented as a C primitive (not as a library function).

`float seconds()`

Returns the count of system time in seconds, as a floating point number. Resolution is one millisecond.

```
void sleep(float sec)
```

Waits for an amount of time equal to or slightly greater than `sec` seconds. `sec` is a floating point number.

Example:

```
/* wait for 1.5 seconds */  
sleep(1.5);
```

```
void msleep(long msec)
```

Waits for an amount of time equal to or greater than `msec` milliseconds. `msec` is a long integer.

Example:

```
/* wait for 1.5 seconds */  
msleep(1500L);
```

### 7.8.4 Tone Functions

Several commands are provided for producing tones on the standard beeper.

```
void beep()
```

Produces a tone of 500 Hertz for a period of 0.3 seconds.

```
void tone(float frequency, float length)
```

Produces a tone at pitch `frequency` Hertz for `length` seconds. Both `frequency` and `length` are floats.

```
void set_beeper_pitch(float frequency)
```

Sets the beeper tone to be `frequency` Hz. The subsequent function is then used to turn the beeper on.

```
void beeper_on()
```

Turns on the beeper at last frequency selected by the former function.

```
void beeper_off()
```

Turns off the beeper.

## 7.9 Multi-Tasking

### 7.9.1 Overview

One of the most powerful features of IC is its multi-tasking facility. Processes can be created and destroyed dynamically during run-time.

Any C function can be spawned as a separate task. Multiple tasks running the same code, but with their own local variables, can be created.

Processes communicate through global variables: one process can set a global to some value, and another process can read the value of that global.

Each time a process runs, it executes for a certain number of *ticks*, defined in milliseconds. This value is determined for each process at the time it is created. The default number of ticks is five; therefore, a default process will run for 5 milliseconds until its “turn” ends and the next process is run. All processes are kept track of in a *process table*; each time through the table, each process runs once (for an amount of time equal to its number of ticks).

Each process has its own *program stack*. The stack is used to pass arguments for function calls, store local variables, and store return addresses from function calls. The size of this stack is defined at the time a process is created. The default size of a process stack is 256 bytes.

Processes that make extensive use of recursion or use large local arrays will probably require a stack size larger than the default. Each function call requires two stack bytes (for the return address) plus the number of argument bytes; if the function that is called creates local variables, then they also use up stack space. In addition, C expressions create intermediate values that are stored on the stack.

It is up to the programmer to determine if a particular process requires a stack size larger than the default. A process may also be created with a stack size *smaller* than the default, in order to save stack memory space, if it is known that the process will not require the full default amount.

When a process is created, it is assigned a unique *process identification number* or *pid*. This number can be used to kill a process.

## 7.9.2 Creating New Processes

The function to create a new process is `start_process`. `start_process` takes one mandatory argument—the function call to be started as a process. There are two optional arguments: the process’s number of ticks and stack size. (If only one optional argument is given, it is assumed to be the ticks number, and the default stack size is used.)

`start_process` has the following syntax:

```
int start_process( function-call( ... ) , [TICKS] , [STACK-SIZE] )
```

`start_process` returns an integer, which is the process ID assigned to the new process.

The function call may be any valid call of the function used. The following code shows the function `main` creating a process:

```

void check_sensor(int n)
{
    while (1)
        printf("Sensor %d is %d\n", n, digital(n));
}

void main()
{
    start_process(check_sensor(2));
}

```

Normally when a C function ends, it exits with a return value or the “void” value. If a function invoked as a process ends, it “dies,” letting its return value (if there was one) disappear. (This is okay, because processes communicate results by storing them in globals, not by returning them as return values.) Hence in the above example, the `check_sensor` function is defined as an infinite loop, so as to run forever (until the board is reset or a `kill_process` is executed).

Creating a process with a non-default number of ticks or a non-default stack size is simply a matter of using `start_process` with optional arguments; e.g.

```
start_process(check_sensor(2), 1, 50);
```

will create a `check_sensor` process that runs for 1 milliseconds per invocation and has a stack size of 50 bytes (for the given definition of `check_sensor`, a small stack space would be sufficient).

### 7.9.3 Destroying Processes

The `kill_process` function is used to destroy processes. Processes are destroyed by passing their process ID number to `kill_process`, according to the following syntax:

```
int kill_process(int pid)
```

`kill_process` returns a value indicating if the operation was successful. If the return value is 0, then the process was destroyed. If the return value is 1, then the process was not found.

The following code shows the `main` process creating a `check_sensor` process, and then destroying it one second later:

```

void main()
{
    int pid;

    pid= start_process(check_sensor(2));
    sleep(1.0);
    kill_process(pid);
}

```

### 7.9.4 Process Management Commands

IC has two commands to help with process management. The commands only work when used at the IC command line. They are not C functions that can be used in code.

`kill_all`

kills all currently running processes.

`ps`

prints out a list of the process status.

The following information is presented: process ID, status code, program counter, stack pointer, stack pointer origin, number of ticks, and name of function that is currently executing.

### 7.9.5 Process Management Library Functions

The following functions are implemented in the standard C library.

`void hog_processor()`

Allocates an additional 256 milliseconds of execution to the currently running process. If this function is called repeatedly, the system will wedge and only execute the process that is calling `hog_processor()`. Only a system reset will unwedge from this state. Needless to say, this function should be used with extreme care, and should not be placed in a loop, unless wedging the machine is the desired outcome.

`void defer()`

Makes a process swap out immediately after the function is called. Useful if a process knows that it will not need to do any work until the next time around the scheduler loop. `defer()` is implemented as a C built-in function.

## 7.10 Floating Point Functions

In addition to basic floating point arithmetic (addition, subtraction, multiplication, and division) and floating point comparisons, a number of exponential and transcendental functions are built in to IC:

`float sin(float angle)`

Returns sine of `angle`. Angle is specified in radians; result is in radians.

`float cos(float angle)`

Returns cosine of `angle`. Angle is specified in radians; result is in radians.

`float tan(float angle)`

Returns tangent of `angle`. Angle is specified in radians; result is in radians.

`float atan(float angle)`

Returns arc tangent of `angle`. Angle is specified in radians; result is in radians.

`float sqrt(float num)`

Returns square root of `num`.

`float log10(float num)`

Returns logarithm of `num` to the base 10.

`float log(float num)`

Returns natural logarithm of `num`.

`float exp10(float num)`

Returns 10 to the `num` power.

`float exp(float num)`

Returns  $e$  to the `num` power.

`(float) a ^ (float) b`

Returns `a` to the `b` power.

## 7.11 Memory Access Functions

IC has primitives for directly examining and modifying memory contents. These should be used with care as it would be easy to corrupt memory and crash the system using these functions.

There should be little need to use these functions. Most interaction with system memory should be done with arrays and/or globals.

`int peek(int loc)`

Returns the byte located at address `loc`.

`int peekword(int loc)`

Returns the 16-bit value located at address `loc` and `loc+1`. `loc` has the most significant byte, as per the 6811 16-bit addressing standard.

```
void poke(int loc, int byte)
```

Stores the 8-bit value `byte` at memory address `loc`.

```
void pokeword(int loc, int word)
```

Stores the 16-bit value `word` at memory addresses `loc` and `loc+1`.

```
void bit_set(int loc, int mask)
```

Sets bits that are set in `mask` at memory address `loc`.

```
void bit_clear(int loc, int mask)
```

Clears bits that are set in `mask` at memory address `loc`.

## 7.12 Error Handling

There are two types of errors that can happen when working with IC: *compile-time* errors and *run-time* errors.

Compile-time errors occur during the compilation of the source file. They are indicative of mistakes in the C source code. Typical compile-time errors result from incorrect syntax or mis-matching of data types.

Run-time errors occur while a program is running on the board. They indicate problems with a valid C form when it is running. A simple example would be a divide-by-zero error. Another example might be running out of stack space, if a recursive procedure goes too deep in recursion.

These types of errors are handled differently, as is explained below.

### 7.12.1 Compile-Time Errors

When compiler errors occur, an error message is printed to the screen. All compile-time errors must be fixed before a file can be downloaded to the board.

### 7.12.2 Run-Time Errors

When a run-time error occurs, an error message is displayed on the LCD screen indicating the error number. If the board is hooked up to IC when the error occurs, a more verbose error message is printed on the terminal.

Here is a list of the run-time error codes:

Error Code	Description
1	no stack space for <code>start_process()</code>
2	no process slots remaining
3	array reference out of bounds
4	stack overflow error in running process
5	operation with invalid pointer
6	floating point underflow
7	floating point overflow
8	floating point divide-by-zero
9	number too small or large to convert to integer
10	tried to take square root of negative number
11	tangent of 90 degrees attempted
12	log or ln of negative number or zero
15	floating point format error in printf
16	integer divide-by-zero

## 7.13 Binary Programs

With the use of a customized 6811 assembler program, IC allows the use of machine language programs within the C environment. There are two ways that machine language programs may be incorporated:

1. Programs may be called from C as if they were C functions.
2. Programs may install themselves into the interrupt structure of the 6811, running repetitiously or when invoked due to a hardware or software interrupt.

When operating as a function, the interface between C and a binary program is limited: a binary program must be given one integer as an argument, and will return an integer as its return value. However, programs in a binary file can declare any number of global integer variables in the C environment. Also, the binary program can use its argument as a pointer to a C data structure.

### 7.13.1 The Binary Source File

Special keywords in the source assembly language file (or module) are used to establish the following features of the binary program:

**Entry point.** The entry point for calls to each program defined in the binary file.

**Initialization entry point.** Each file may have one routine that is called automatically upon a reset condition. (The reset conditions are explained in Section 7.4.3, which discusses global variable initialization.) This initialization routine particularly useful for programs which will function as interrupt routines.

**C variable definitions.** Any number of two-byte C integer variables may be declared within a binary file. When the module is loaded into IC, these variables become defined as globals in C.

To explain how these features work, let's look at a sample IC binary source program, listed in Figure 7.2.

```

/* Sample icb file */

/* origin for module and variables */
    ORG     MAIN_START

/* program to return twice the argument passed to us */
subroutine_double:
    ASLD
    RTS

/* declaration for the variable "foo" */
variable_foo:
    FDB     55

/* program to set the C variable "foo" */
subroutine_set_foo:
    STD     variable_foo
    RTS

/* program to retrieve the variable "foo" */
subroutine_get_foo:
    LDD     variable_foo
    RTS

/* code that runs on reset conditions */
subroutine_initialize_module:
    LDD     #69
    STD     variable_foo
    RTS

```

Figure 7.2: Sample IC Binary Source File: `testicb.asm`

The first statement of the file (“`ORG MAIN_START`”) declares the start of the binary programs. This line must precede the code itself itself.

The entry point for a program to be called from C is declared with a special form beginning with the text `subroutine_`. In this case, the name of the binary program

is `double`, so the label is named `subroutine_double`. As the comment indicates, this is a program that will double the value of the argument passed to it.

When the binary program is called from C, it is passed one integer argument. This argument is placed in the 6811's D register (also known as the "Double Accumulator") before the binary code is called.

The `double` program doubles the number in the D register. The `ASLD` instruction ( "Arithmetic Shift Left Double [Accumulator]") is equivalent to multiplying by 2; hence this doubles the number in the D register.

The `RTS` instruction is "Return from Subroutine." All binary programs must exit using this instruction. When a binary program exits, the value in the D register is the return value to C. Thus, the `double` program doubles its C argument and returns it to C.

### Declaring Variables in Binary Files

The label `variable_foo` is an example of a special form to declare the name and location of a variable accessible from C. The special label prefix "`variable_`" is followed the name of the variable, in this case, "`foo`."

This label must be immediately followed by the statement `FDB <number>`. This is an assembler directive that creates a two-byte value (which is the initial value of the variable).

Variables used by binary programs must be declared in the binary file. These variables then become C globals when the binary file is loaded into C.

The next binary program in the file is named "`set_foo`." It performs the action of setting the value of the variable `foo`, which is defined later in the file. It does this by storing the D register into the memory contents reserved for `foo`, and then returning.

The next binary program is named "`get_foo`." It loads the D register from the memory reserved for `foo` and then returns.

### Declaring an Initialization Program

The label `subroutine_initialize_module` is a special form used to indicate the entry point for code that should be run to initialize the binary programs. This code is run upon standard reset conditions: program download, hardware reset, or running of the `main()` function.

In the example shown, the initialization code stores the value 69 into the location reserved for the variable `foo`. This then overwrites the 55 which would otherwise be the default value for that variable.

Initialization of global variables defined in a binary module is done differently than globals defined in C. In a binary module, the globals are initialized to the value

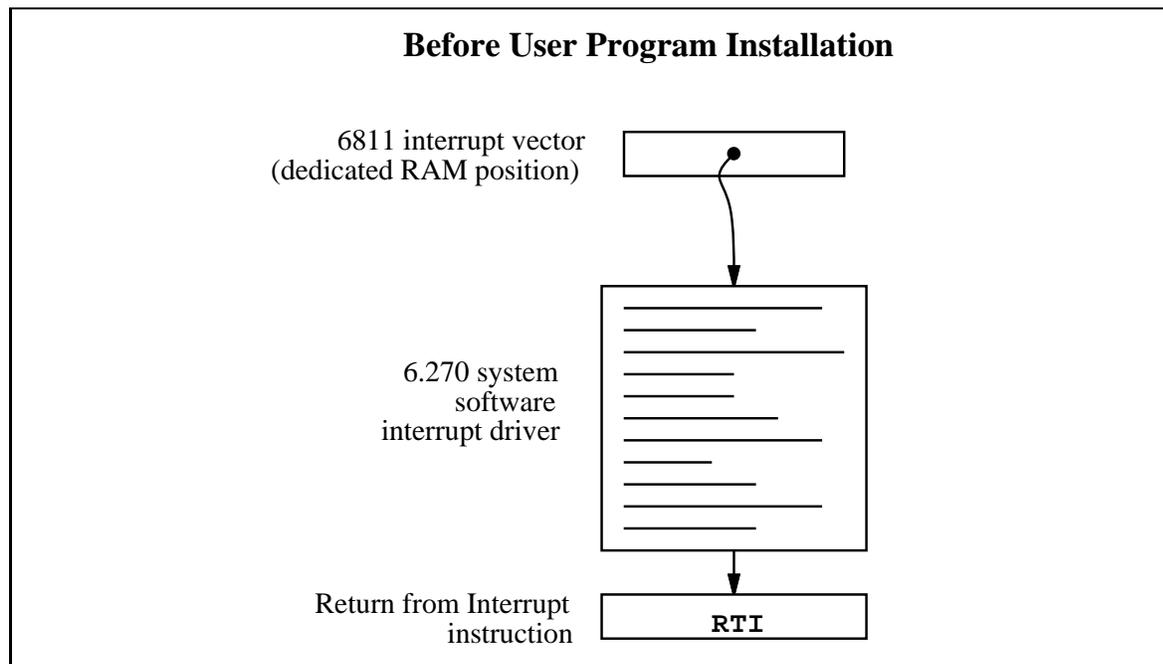


Figure 7.3: Interrupt Structure Before User Program Installation

declared by the `FDB` statement only when the code is downloaded to the 6811 board (not upon reset or running of main, like normal globals).

However, the initialization routine is run upon standard reset conditions, and can be used to initialize globals, as this example has illustrated.

### 7.13.2 Interrupt-Driven Binary Programs

Interrupt-driven binary programs use the initialization sequence of the binary module to install a piece of code into the interrupt structure of the 6811.

The 6811 has a number of different interrupts, mostly dealing with its on-chip hardware such as timers and counters. One of these interrupts is used by the 6.270 software to implement time-keeping and other periodic functions (such as LCD screen management). This interrupt, dubbed the “System Interrupt,” runs at 1000 Hertz.

Instead of using another 6811 interrupt to run user binary programs, additional programs (that need to run at 1000 Hz. or less) may install themselves into the System Interrupt. User programs would then become part of the 1000 Hz interrupt sequence.

This is accomplished by having the user program “intercept” the original 6811 interrupt vector that points to 6.270 interrupt code. This vector is made to point at the user program. When user program finishes, it jumps to the start of the 6.270 interrupt code.

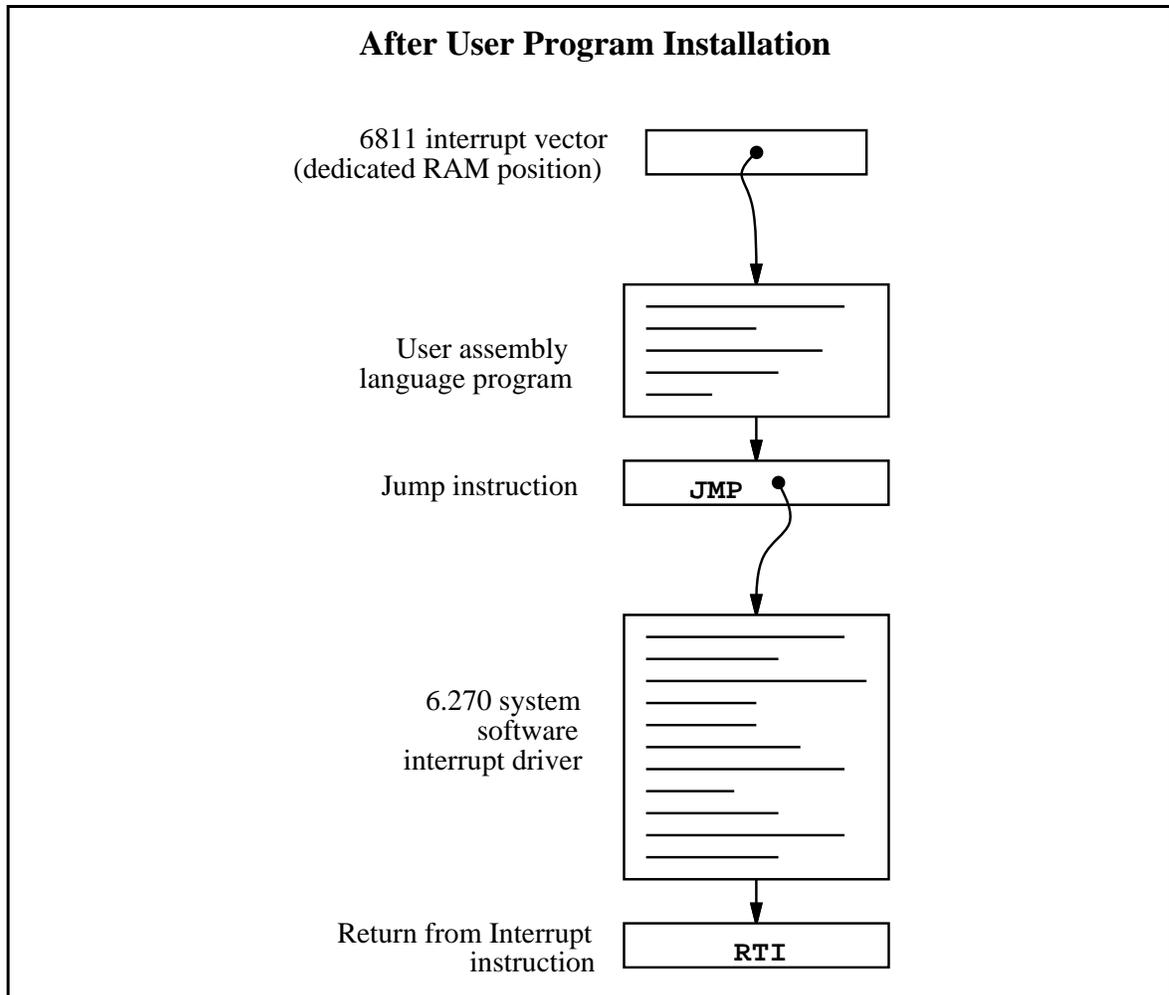


Figure 7.4: Interrupt Structure After User Program Installation

Figure 7.3 depicts the interrupt structure before user program installation. The 6811 vector location points to system software code, which terminates in a “return from interrupt” instruction.

Figure 7.4 illustrates the result after the user program is installed. The 6811 vector points to the user program, which exits by jumping to the system software driver. This driver exits as before, with the RTI instruction.

Multiple user programs could be installed in this fashion. Each one would install itself ahead of the previous one. Some standard 6.270 library functions, such as the shaft encoder software, is implemented in this fashion.

Figure 7.5 shows an example program that installs itself into the System Interrupt. This program toggles the signal line controlling the piezo beeper every time it is run; since the System Interrupt runs at 1000 Hz., this program will create a continuous tone of 500 Hz.

The first line after the comment header includes a file named “6811regs.asm”. This file contains equates for all 6811 registers and interrupt vectors; most binary programs will need at least a few of these. It is simplest to keep them all in one file that can be easily included. (This and other files included by the `as11` assembler are located in the assembler’s default library directory, which is `/mit/6.270/lib/as11/` on the MIT Athena system.)

The `subroutine_initialize_module` declaration begins the initialization portion of the program. The file “`ldxibase.asm`” is then included. This file contains a few lines of 6811 assembler code that perform the function of determining the base pointer to the 6811 interrupt vector area, and loading this pointer into the 6811 X register.

The following four lines of code install the interrupt program (beginning with the label `interrupt_code_start`) according to the method that was illustrated in Figure 7.4.

First, the existing interrupt pointer is fetched. As indicated by the comment, the 6811’s TOC4 timer is used to implement the System Interrupt. The vector is poked into the JMP instruction that will conclude the interrupt code itself.

Next, the 6811 interrupt pointer is replaced with a pointer to the new code. These two steps complete the initialization sequence.

The actual interrupt code is quite short. It toggles bit 3 of the 6811’s PORTA register. The PORTA register controls the eight pins of Port A that connect to external hardware; bit 3 is connected to the piezo beeper.

The interrupt code exits with a jump instruction. The argument for this jump is poked in by the initialization program.

The method allows any number of programs located in separate files to attach themselves to the System Interrupt. Because these files can be loaded from the C environment, this system affords maximal flexibility to the user, with small overhead in terms of code efficiency.

```

* icb file: "sysibEEP.asm"

*
* example of code installing itself into
* SystemInt 1000 Hz interrupt
*
* Fred Martin
* Thu Oct 10 21:12:13 1991
*

#include <6811regs.asm>

        ORG     MAIN_START

subroutine_initialize_module:

#include <ldxibase.asm>
* X now has base pointer to interrupt vectors ($FF00 or $BF00)

* get current vector; poke such that when we finish, we go there
        LDD     TOC4INT,X          ; SystemInt on TOC4
        STD     interrupt_code_exit+1

* install ourself as new vector
        LDD     #interrupt_code_start
        STD     TOC4INT,X

        RTS

* interrupt program begins here
interrupt_code_start:
* frob the beeper every time called
        LDAA    PORTA
        EORA    #%00001000        ; beeper bit
        STAA    PORTA

interrupt_code_exit:
        JMP     $0000        ; this value poked in by init routine

```

Figure 7.5: sysibEEP.asm: Binary Program that Installs into System Interrupt

### 7.13.3 The Binary Object File

The source file for a binary program must be named with the `.asm` suffix. Once the `.asm` file is created, a special version of the 6811 assembler program is used to construct the binary object code. This program creates a file containing the assembled machine code plus label definitions of entry points and C variables.

```
S116802005390037FD802239FC802239CC0045FD8022393C
S9030000FC
S116872B05390037FD872D39FC872D39CC0045FD872D39F4
S9030000FC
6811 assembler version 2.1 10-Aug-91
  please send bugs to Randy Sargent (rsargent@athena.mit.edu)
  original program by Motorola.
subroutine_double 872b *0007
subroutine_get_foo 8733 *0021
subroutine_initialize_module 8737 *0026
subroutine_set_foo 872f *0016
variable_foo 872d *0012 0017 0022 0028
```

Figure 7.6: Sample IC Binary Object File: `testicb.icb`

The program `as11_ic` is used to assemble the source code and create a binary object file. It is given the filename of the source file as an argument. The resulting object file is automatically given the suffix `.icb` (for IC Binary). Figure 7.6 shows the binary object file that is created from the `testicb.asm` example file.

### 7.13.4 Loading an `icb` File

Once the `.icb` file is created, it can be loaded into IC just like any other C file. If there are C functions that are to be used in conjunction with the binary programs, it is customary to put them into a file with the same name as the `.icb` file, and then use a `.lis` file to load the two files together.

### 7.13.5 Passing Array Pointers to a Binary Program

A pointer to an array is a 16-bit integer address. To coerce an array pointer to an integer, use the following form:

```
array_ptr= (int) array;
```

where `array_ptr` is an integer and `array` is an array.

When compiling code that performs this type of pointer conversion, IC must be used in a special mode. Normally, IC does not allow certain types of pointer

manipulation that may crash the system. To compile this type of code, use the following invocation:

```
ic -wizard
```

Arrays are internally represented with a two-byte length value followed by the array contents.

## 7.14 IC File Formats and Management

This section explains how IC deals with multiple source files.

### 7.14.1 C Programs

All files containing C code must be named with the “.c” suffix.

Loading functions from more than one C file can be done by issuing commands at the IC prompt to load each of the files. For example, to load the C files named `foo.c` and `bar.c`:

```
C> load foo.c
C> load bar.c
```

Alternatively, the files could be loaded with a single command:

```
C> load foo.c bar.c
```

If the files to be loaded contain dependencies (for example, if one file has a function that references a variable or function defined in the other file), then the second method (multiple file names to one load command) or the following approach must be used.

### 7.14.2 List Files

If the program is separated into multiple files that are always loaded together, a “list file” may be created. This file tells IC to load a set of named files. Continuing the previous example, a file called `gnu.lis` can be created:

Listing of `gnu.lis`:

```
foo.c
bar.c
```

Then typing the command `load gnu.lis` from the C prompt would cause both `foo.c` and `bar.c` to be loaded.

### 7.14.3 File and Function Management

#### Unloading Files

When files are loaded into IC, they stay loaded until they are explicitly unloaded. This is usually the functionality that is desired. If one of the program files is being worked on, the other ones will remain in memory so that they don't have to be explicitly re-loaded each time the one undergoing development is reloaded.

However, suppose the file `foo.c` is loaded, which contains a definition for the function `main`. Then the file `bar.c` is loaded, which happens to also contain a definition for `main`. There will be an error message, because both files contain a `main`. IC will unload `bar.c`, due to the error, and re-download `foo.c` and any other files that are presently loaded.

The solution is to first unload the file containing the `main` that is not desired, and then load the file that contains the new `main`:

```
C> unload foo.c
C> load bar.c
```

## 7.15 Configuring IC

IC has a multitude of command-line switches that allow control of a number of things. Explanations for these switches can be gotten by issuing the command “`ic -help`”.

IC stores the search path for and name of the library files internally; these may be changed by executing the command “`ic -config`”. When this command is run, IC will prompt for a new path and library file name, and will create a new executable copy of itself with these changes.

# Appendix A

## Introduction to 6.270

6.270 is a hands-on, learn-by-doing class in which participants design and build a robot that will play in a competition at the end of the class. From the student’s perspective, the goal of the class is to design a robotic machine that will be able to navigate its way around the playing surface, and successfully interact with game objects, including the opposing machine.

The machines in MIT’s Introduction to Design class (2.70) are controlled by humans wielding joysticks. They are not robots—they are machines.

The machines built by students of 6.270 *are* robots. They are fully autonomous entities, operating under their own battery power with a microprocessor in control. Once a 6.270 contest round begins, there is no human intervention. The design of a 6.270 robot is a complex task, because an entry must be designed intelligently with respect to mechanical, electronic, and control factors.

From the perspective of its creators, the goal of 6.270 is to teach students about engineering and design by giving them the hardware, software, and information they need to design and implement an operational robot.

The course includes concepts and applications that are related to various MIT engineering classes, but 6.270 has no formal curriculum, no material that must be “covered.” This allows the class to have a great deal of flexibility, which it hands back to its students. We hope is that the course inspires students to learn whatever material is most exciting to them and challenges them at whatever level is most appropriate. If one student focuses on hardware issues while another tackles puzzling software problems, the course is a success to both of them.

There are no formal prerequisites for 6.270. We’ve found that people can learn everything they need to know by working with each other, being introduced to some material in class, and mostly, by hacking on their robots. All members of the MIT community, from frosh to grad students, staff, or even professors, are encouraged to register and take the class.

6.270 does require that its students be prepared to put forth a real effort! We

expect most students to spend about eighty hours over the month of IAP building their robots. We've also noticed that people who make a real commitment to the class are more confident, feel more involved, and have a lot more fun. So: if you going to take 6.270, be ready for a month-long immersion into robotics.

## A.1 Registration Policy

All entrants will be organized into teams. There are a couple of reasons for this. First, we find that people learn a lot in the close and intense relationship of a small team. Second, we think the class would be too much work for one person to handle by his or her self.

You are encouraged to form a team of two to four people and register together. You may also register alone, in which case we will find you a team with two other people.

The lottery for this year's class was held on October 28th, 1991. Between the opening of enrollment in late September and that date, one hundred and seventeen (117) teams consisting of three hundred and forty-one (341) individuals had registered for the class.

Registration in the class was limited to fifty (50) teams. We would have accepted more students if resources permitted us but they do not.

## A.2 Kit Fee and Toolkit Fee

Your 6.270 kit, which is yours to keep at the end of the contest, will be valued at about \$500. The class is mostly financed by our commercial sponsors (namely Microsoft, Motorola, LEGO, and Polaroid) and the Course Six department, but part of the budget is derived from the entry fee.

This year the entry fee is \$50 per team, the same as last year. *You may have heard or read differently due to a change from earlier versions of our registration materials, but this figure is the correct one!* We have managed to keep student costs down as a result of the strong financial commitment that has been given to the class from the Course Six department.

We encourage you to start thinking now about the fate of your kit after the contest. In the past, we have observed some teams having great difficulty in deciding which team members should have stewardship of the kit or various parts of the kit. We don't have any easy answers for this hard problem, but we do suggest that you open the question earlier rather than later.

Separate from the 6.270 kit, a complete set of electronic hand tools will be reserved for purchase by your team. This kit will include a soldering iron, diagonal cutters, long nose pliers, wire strippers, a multimeter, and several other useful implements.

The 6.270 tool kit will have a retail value between \$75 and \$100; we expect to sell the kits for between \$40 and \$50 (we can give you these prices due to the quantity discounts we get in purchasing for the class). *You will be expected to either provide your own electronic assembly tools or purchase the standard tool kit.*

A final word about contest costs: if it is difficult for you to afford the contest costs, both the 6.270 kit and the toolkit are returnable (if in good condition) for a refund. If you would like to take the class, but you cannot afford to put up the money to register for the class and buy the toolkit, come talk to the organizers. We can probably work something out.

## A.3 Team Organization

In many ways, the quality of the interactions of the members of your team will profoundly affect the quality of your overall experience in 6.270. Over the past years of 6.270, we have seen many different students and many different team dynamics.

There are important questions to be answered when thinking about the organization of your 6.270 team:

- Will each participant be responsible for one portion of the design? Conversely, will all participants take part in all activities?
- How will major design decisions, like the overall machine strategy or physical configuration, be made? Will there need to be a consensus amongst all of the team members before these types of decisions will be made?
- How will disputes be resolved?
- Which team members will keep which parts of the 6.270 kit at the end of the course?

These are tough questions that have no pat answers. It is recommended that questions like these be taken seriously, and that each team spend time discussing these issues at the start of the course. Each team should come up with a solution that works best for them.

As mentioned earlier, teams may have two, three, or four students. However, we *strongly recommend* teams of two or three students. It has been our observation that most teams of four tend to end up having a real “working team” of two or three persons. We would rather that you form a team of two or three people who are really committed to working together than you form a team of four with a less strong commitment.

If you are taking the class for credit, we will require that this suggestion be followed: persons who are taking the class for credit must be members of teams of two or three people.

## A.4 6.270 Alumni

If you have taken 6.270 before and you are taking it again, you will be expected to assist by working as a laboratory TA. Alums will be expected to work about four hours *per week* for the class. Alternate duties to working as a lab TA include: helping to build the game board, and other general support activities. If you're an alum, we will expect you to have a genuine and positive attitude about this service work.

Alumni had the same chance in the lottery as other registrants.

## A.5 Credit Guidelines

6.270 is offered as MIT course 6.190 for six units of P/F credit. Taking the class for credit is optional. The class will entail a lot of work whether or not it is taken for credit; signing up for credit will give you official recognition for taking the class. If you sign up for credit but then do not complete the requirement, your registration will be dropped; it will be as if you never signed up in the first place.

Our job as instructors is to ensure that credit is properly awarded to students deserving of it. Our basic assumption is that anyone who is in the class is going to be doing a lot of work; the guidelines should add only a little bit of overhead to you in reporting your work to us. Hopefully, you may even learn a little more by going through the process, which includes reporting your progress through the course.

As mentioned in the registration guidelines, team size for students registering for credit will be limited to two or three members. This is because of our (mostly true) observation that it's very hard for a team of four members to fairly distribute work amongst themselves.

The following requirements for credit have been established:

- **Individual Written Reports.**

Each individual desiring credit must turn in a series of three weekly written reports. These will be due at the end of each week.

The written design report must state explicitly what *you* have contributed to the progress of your robot. This should include:

- actual construction work, programming, or other tangible results that you have done;
- ideas you have contributed to the design of your robot (whether they have been implemented or not);
- plans for the next week of work.

The purpose of the individual report is to get a sense of what each person on a team is contributing to the design, so it's important to make sure we know what you've done.

- **Team Video Reports.**

In addition to the individual reports, a team video report will be done once per week.

A video station will be set up in the 6.270 lab area. To make your report, you and your team can simply go to the camera and make a brief presentation on the status of your robot. This presentation should focus on issues that the team has worked on together, such as the current state of the robot, the strategy of the robot, and how the team arrived at consensus (or not!) on particular issues.

Hopefully, the video station concept will make the design reporting a fun and painless process. Any ideas presented to the camera *will remain confidential* for the duration of the contest.

- **Team Interview.** During the final week of IAP, your team will have a 20 minute meeting with the contest organizers to discuss your participation in the class: what you learned, what you didn't learn, what you liked about it, what you didn't like.

The purpose of the interview is two-fold. Most importantly, it serves as an opportunity for you to give us (the organizers) feedback about the course and your learning experience so that we can continue to improve the class format. Secondly, it helps us make sure that people who are receiving credit actually did some work.

The interview is required for all individuals who will receive credit, but we hope that people who aren't registered for credit will participate as well.

- **Completed Robot.** Your team must "show" a robot the day of the contest. Its functionality (or lack of) has no affect on your receiving credit for the work you have done; the combination of the design reports and the team interview will be the main indicators of your participation.
- **Program Listing.** You must turn in a copy of the program that your robot uses in the contest.

These course requirements are meant to be useful to both you, the course participant, and the instructors, who will be authorizing credit. You should have no problem at all receiving credit if all of the requirements are satisfied. If you have any questions about your standing in the course at any time, feel free to ask any of the instructors for feedback.

Please note that there is no leeway on any of the due dates, due to the scheduling constraints of the Registrar and the sanity of the organizers. Please do not ask for extensions.

## A.6 Schedule

The schedule of activities between the start of IAP and the eve of the contest is very tight. You will have to work steadily and with determination to produce a working machine by the end of the course. In no fashion do we, the contest organizers, say that this course is not time consuming! However, since it is IAP, you can assume it is the main time-sink you've signed up for.

There will be about 150 students taking the 1992 6.270 course, making it the largest course taught during IAP. Since much of the learning we believe occurs with a hands-on interactions, the class will be too large as a whole to teach on this basis. Therefore we have several class meeting formats, including lectures, recitations, and lab.

We recommend that you attend all of the lectures and recitations (for the section you are in). We will deal with administrative and "bug fix" matters at the beginning of each meeting, so if are not finding these sessions helpful, you can leave after this practical information is disbursed.

- **General Lecture** The function of the general lecture to teach the basics of what is going on during that week of the course. The lecture will be held on **Tuesday** nights in room 34-101 from 7-10 pm. We do not expect all the lectures to take the entire three hours, but some may.
- **Catch-Up Help Session** This informal class will be designed for those people who may feel that they are not up to speed in a certain area. For example, we may go over general microprocessor architecture, programming, fundamentals behind sensors. It is not necessary to go to these lectures, but they will be fun and you may even learn something. These lectures will be held on **Sunday** at 7:00 pm, in Room 34-101.
- **Recitations** Some material will be presented in recitations rather than in lectures, to encourage a more interactive format. There will be three recitation sections, meeting on Wednesday and Thursday evenings.

The class will be divided into three groups to approximately sort people of comparable background levels. **Section B** (beginning) will meet **Wednesday** nights from 7:00 pm to 8:30 pm. **Section I** (intermediate) will meet **Thursday** nights from 7:00 pm to 8:30 pm. **Section A** (advanced) will meet **Thursday** nights from 8:30 to 10:00 pm. Sections will be selected to match contestants'

choice as best as possible. Please do not change recitation sections unless you are trading places with someone else in the class.

- **Laboratory Time**

This is supervised time for building your robot. Lab time will be most critical when working on your microprocessor and other circuit boards. After that, building motors and sensors will be important. During the final week, testing machines on the table will be the focus of lab activity.

Labs will be held on the 6th floor of building 38 during the day time and in the Chu Lounge (38-201) when at night.

### A.6.1 Important Dates

Before reading the listing the full month of meetings, please note following *very important* meetings:

**Parts-Sorting Session.** Attendance to this session is mandatory: all teams must provide four person-hours of manual labor helping to sort out the kit parts. Usually this session is a lot of fun as you get introduced to other people in the class and see all of the electronic goodies.

*Date, Time, and Place:* Monday, January 6th, 10:00 am to 4:00 pm, Building 38.

**Official Orientation Meeting.** Attendance to this session is mandatory: each team must have at least 50% of its members in attendance. In this session, we go over the contest rules, organization of the class, and hand out the kits.

*Date, Time, and Place:* Tuesday, January 7th, 2:00 pm to 5:00 pm, Room 34-101.

**The Contest, First Round.** Your machine must compete in the first round to qualify for the second round.

*Date, Time, and Place:* Saturday, February 1st, 6:00 pm, Room 34-101.

**Robot Impounding and Party.** All work on robots will cease at 7 o'clock P.M., Sunday the Second of February. Robots will be impounded at *The Chu Lounge, 38-201*. A party will ensue.

**The Contest, Second Round.** Robots will be released from impoundment at 5:00 pm, Monday February 3rd. You must check your robot into 26-100 by 6:00 pm. Good luck!

*Date, Time, and Place:* Monday, February 3rd, 7:00 pm, Room 26-100.

Figure A.1 illustrates the month's schedule in pictorial form.

# 6.270 IAP '92

Figure A.1: Pictorial Schedule of the Month's Activities

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday
<b>Jan 5</b>	<b>6</b> <b>Parts-Sorting.</b> Bldg. 38 10am-5pm	<b>7</b> <b>1st Meeting!</b> 34-101 5-8pm	<b>8</b> <b>Supervised Lab</b> 10am-4pm, 38-600 <b>B Recitation</b> 34-101, 7-8:30pm	<b>9</b> <b>Supervised Lab</b> 10am-4pm, 38-600 <b>I Recitation</b> 34-101, 7-8:30pm <b>A Recitation</b> 34-101, 8:30-10pm	<b>10</b> <b>Supervised Lab</b> 10am-4pm, 38-600
<b>12</b> <b>Intro to 6811 'C'</b> 34-101, 7-9pm	<b>13</b> <b>Supervised Lab</b> 12n-4pm, 38-600	<b>14</b> <b>Supervised Lab</b> 12n-4pm, 38-600 <b>LEGO Lecture.</b> 34-101, 7pm	<b>15</b> <b>Supervised Lab</b> 12n-4pm, 38-600 <b>B Recitation</b> 34-101, 7-8:30pm	<b>16</b> <b>Supervised Lab</b> 12n-4pm, 38-600 <b>I Recitation</b> 34-101, 7-8:30pm <b>A Recitation</b> 34-101, 8:30-10pm	<b>17</b> <b>Supervised Lab</b> 12n-4pm, 38-600
<b>19</b>	<b>20</b> <b>Martin Luther King Day</b> <i>(No scheduled activities.)</i>	<b>21</b> <b>Supervised Lab</b> 12n-4pm, 38-600 <b>Robust Programming.</b> 34-101, 7pm	<b>22</b> <b>Supervised Lab</b> 12n-4pm, 38-600 <b>B Recitation</b> 34-101, 7-8:30pm	<b>23</b> <b>Supervised Lab</b> 12n-4pm, 38-600 <b>I Recitation</b> 34-101, 7-8:30pm <b>A Recitation</b> 34-101, 8:30-10pm	<b>24</b> <b>Supervised Lab</b> 12n-4pm, 38-600
<b>26</b>	<b>27</b> <b>Supervised Lab</b> 12n-4pm, 38-600	<b>28</b> <b>Team Interviews</b> 34-30x, 7-10pm	<b>29</b> <i>(Last Day of IAP.)</i> <b>Team Interviews</b> 34-30x, 7-10pm	<b>30</b> <i>(Reg Day.)</i> <b>Team Interviews</b> 34-30x, 7-10pm	<b>31</b> <i>(First Day of Classes.)</i>
<b>Feb 1</b> <b>Contest Round 1</b> 34-101 6pm	<b>Feb 2</b> <b>Robot Impounding</b> Chu Lounge 6pm	<b>3</b> <b>The Contest!</b> 26-100 6pm	<b>4</b>	<b>5</b> <i>(resume life)</i>	<b>6</b>

## A.6.2 Detailed Schedule of the Month's Activities.

**Monday, January 6. Part-Sorting Meeting.** *10:00a to 4:00p, Building 38.* Sort kits. Mandatory attendance as mentioned above.

**Tuesday, January 7.**

- **First meeting.** *2:00p to 5:00p, 34-101.* Hand out kits, go over month's schedule, go over rules, get psyched.

**Wednesday, January 8.**

- **Lab.** *10:00a to 4:00p, 6th floor of bldg. 38.* Supervised lab time. Begin building microprocessor board! Soldering demonstrations every 1 hour.
- **Section B, Meeting 1.** *7:00p to 8:30p, 34-101.* Introduction to microprocessor board, demonstration of sample robots, brainstorming of contest design solutions, Q & A.

**Thursday, January 9.**

- **Lab.** *10:00a to 4:00p, 6th floor of bldg. 38.* Supervised lab time. Continue building microprocessor board. Soldering demonstrations every 2 hours.
- **Section I, Meeting 1.** *7:00p to 8:30p, 34-101.* See description of Meeting 1, above.
- **Section A, Meeting 1.** *8:30p to 10:00p, 34-101.* See description of Meeting 1, above.

**Friday, January 10.**

- **Lab.** *10:00a to 4:00p, 6th floor of bldg. 38.* Supervised lab time. Finish microprocessor board; begin testing it.
- **VIDEO DESIGN REPORT #1 DUE.** Complete in lab.

**Saturday, January 11.**

- **Lab.** *10:00a to 4:00p, 6th floor of bldg. 38.* Supervised lab time. Continue testing microprocessor board.

**Sunday, January 12.**

- **Lab.** *10:00a to 4:00p, 6th floor of bldg. 38.* Build battery charger board and LEGO-ize some motors.

- **DESIGN REPORT #1 DUE.**
- **Special Lecture: Introduction to C Programming.** 7:00p to 8:30p, 34-101. This lecture will introduce the ‘C’ language to people who have never used it before.

**Monday, January 13.**

- **Lab.** 12:00n to 4:00p, 6th floor of bldg. 38. Build light sensor, touch sensor, reflectance sensor.

**Tuesday, January 14.**

- **Lab.** 12:00n to 4:00p, 6th floor of bldg. 38. Build Motor Switch Board. Start Expansion Board, IR transmitter board, and wire IR receivers.
- **Lecture: The Art and Science of LEGO.** 7:00p to 10:00p, 34-101. Learn about designing sturdy structures and effective gearboxes with LEGO Technics parts. Short lecture plus lots of workshop time.

**Wednesday, January 15.**

- **Lab.** 12:00n to 4:00p, 6th floor of bldg. 38. Continue Expansion Board, wiring sensors. Begin building robot.
- **Section B, Meeting 2.** 7:00p to 8:30p, 34-101. Introduction to 6811 C environment. Simple control of motors and sensors. Printing to the LCD screen. 6.270 ‘C’ libraries.

**Thursday, January 16.**

- **Lab.** 12:00n to 4:00p, 6th floor of bldg. 38.
- **Section I, Meeting 2.** 7:00p to 8:30p, 34-101. See description of Meeting 2, above.
- **Section A, Meeting 2.** 8:30p to 10:00p, 34-101. See description of Meeting 2, above.

**Friday, January 17.**

- **Lab.** 12:00n to 4:00p, 6th floor of bldg. 38. Finish Expansion Board. Test it. Continue building robot.
- **VIDEO DESIGN REPORT #2 DUE.**

**Saturday, January 18.**

- *No scheduled activities.*

**Sunday, January 19.**

- **WRITTEN DESIGN REPORT #2 DUE.**
- **Catch-Up Help Session.** 7:00p to 10:00p, Chu Lounge (38-201).

**Monday, January 20.**

- **Martin Luther King Day.** *No scheduled activities.*

**Tuesday, January 21.**

- **Lab.** 12:00n to 4:00p, 6th floor of bldg. 38.
- **Lecture: Robust Robot Programming.** 7:00p to 10:00p, 34-101. How to program your robot so that it deals well with a dynamic, unpredictable environment.

**Wednesday, January 22.**

- **Lab.** 12:00n to 4:00p, 6th floor of bldg. 38. Things are done electronics-wise. You are finishing mechanical design.
- **Section B, Meeting 3.** 7:00p to 8:30p, 34-101. Multi-tasking in the 6811 C environment. Performing sensor filtering techniques with processes.

**Thursday, January 23.**

- **Lab.** 12:00n to 4:00p, 6th floor of bldg. 38.
- **Section I, Meeting 3.** 7:00p to 8:30p, 34-101. See description of Meeting 3, above.
- **Section A, Meeting 3.** 8:30p to 10:00p, 34-101. See description of Meeting 3, above.

**Friday, January 24.**

- **Lab.** 12:00n to 4:00p, 6th floor of bldg. 38. Robot is fully constructed; you are working on final programming.
- **VIDEO DESIGN REPORT #3 DUE.**

**Saturday, January 25.**

- *No scheduled activities.*

**Sunday, January 26.**

- **WRITTEN DESIGN REPORT #3 DUE.**
- **Catch-Up Help Session.** 7:00p to 10:00p, *Chu Lounge (38-201).*

No more evening class meetings.

**Monday, January 27.**

- **Lab.** 12:00n to 4:00p, *6th floor of bldg. 38.* Robot is beginning to work as an integrated unit.

**Tuesday, January 28.**

- **Team Interviews, Day 1.** 7:00p to 10:00p, *locations to be announced.*  
Twenty minute interview per team, one of three days only.

**Wednesday, January 29.**

- **Last day of IAP.**
- **Team Interviews, Day 2.** 7:00p to 10:00p, *locations to be announced.*  
Twenty minute interview per team.

**Thursday, January 30.**

- **Registration day.**
- **No Lab.**
- **Team Interviews, Day 3.** 7:00p to 10:00p, *locations to be announced.*  
Twenty minute interview per team.

**Friday, January 31.**

- **First day of classes.**
- **Evening Lab.** 7:00p to 12:00m, *Chu Lounge (38-201).*

**Saturday, February 1.**

- **Lab.** 10:00a to 5:00p, *Chu Lounge (38-201).*

- **First Round of the Contest.** 6:00p, Room 34-101.

**Sunday, February 2.**

- **Last-Chance Lab.** 12:01a to 7:00p, Chu Lounge (38-201) and surrounding areas.
- **Robot Impounding.** 7:00p to 8:00p, Chu Lounge (38-201). **All robots that will compete in the contest must be turned in at this time.**
- **Party.** 8:00p, location TBA.

**Monday, February 3.**

- **The 6.270 Contest, Round 2.** 6:00p to 10:00p, 26-100. Good luck, everyone!

## A.7 Robo-Cup Contest Rules

This year's contest is "Robo-Cup Soccer," a game played by two autonomous robots.

### A.7.1 Object

Have your robot place more balls into its goal within a 45 second period.

### A.7.2 Balls

- The balls will be practice golf balls. These are plastic balls, similar in weight to ping-pong balls, but slightly larger and with less bounce.
- One point will be awarded for each ball in your robot's goal when the contest ends. It does not matter which robot caused a ball to enter your robot's goal.
- The balls will be dispensed six inches away from the wall at designated drop points.  
The balls will be dispensed when the corresponding touch sensor has been activated. The dispenser will allow one ball to be dispensed once every five seconds.
- You may place one ball into your robot before the start of the contest round.
- The balls are inert and all have identical properties.
- Robots may gather balls "into" their body.
- The balls may not be altered or destroyed in any way.

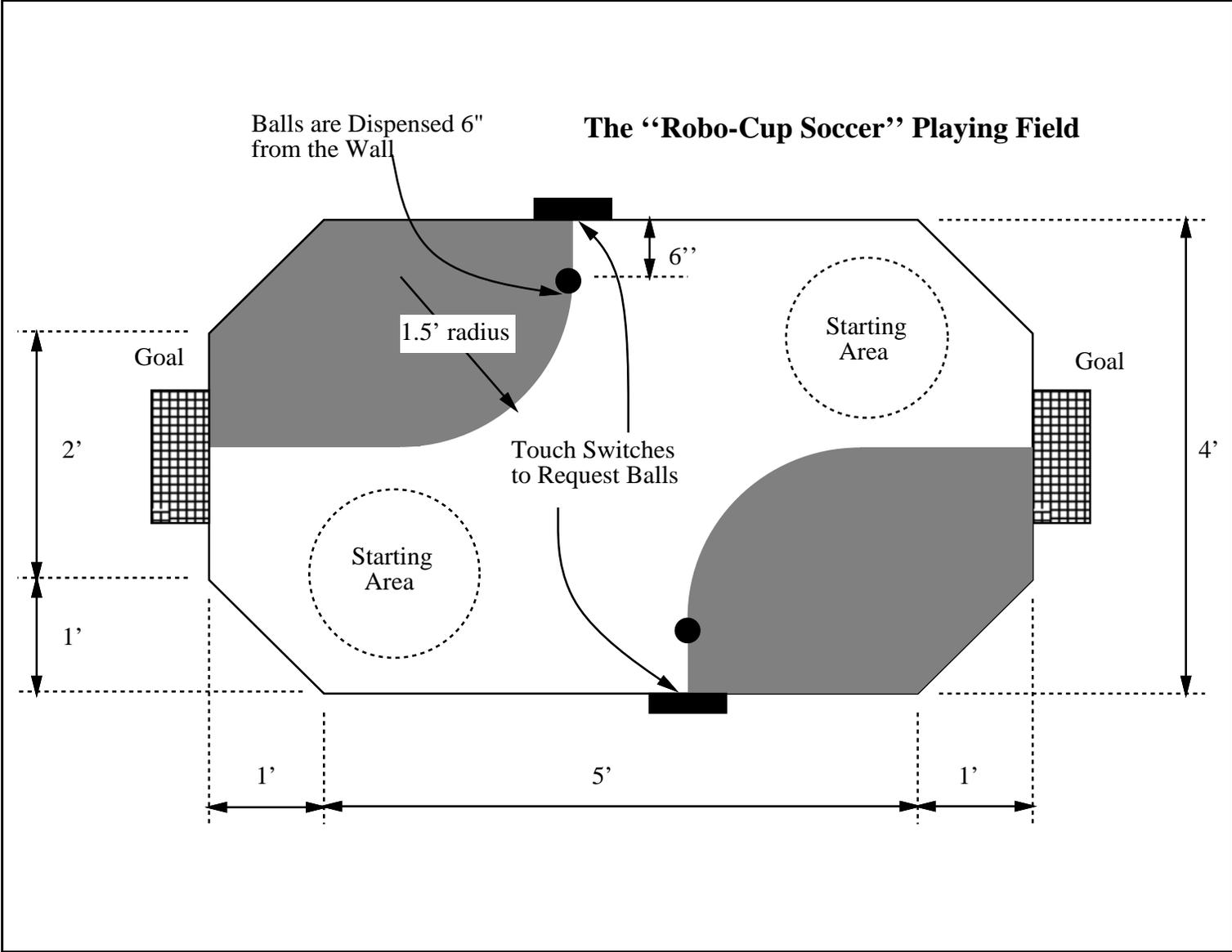


Figure A.2: Robo-Cup '92 Contest Playing Field Specification

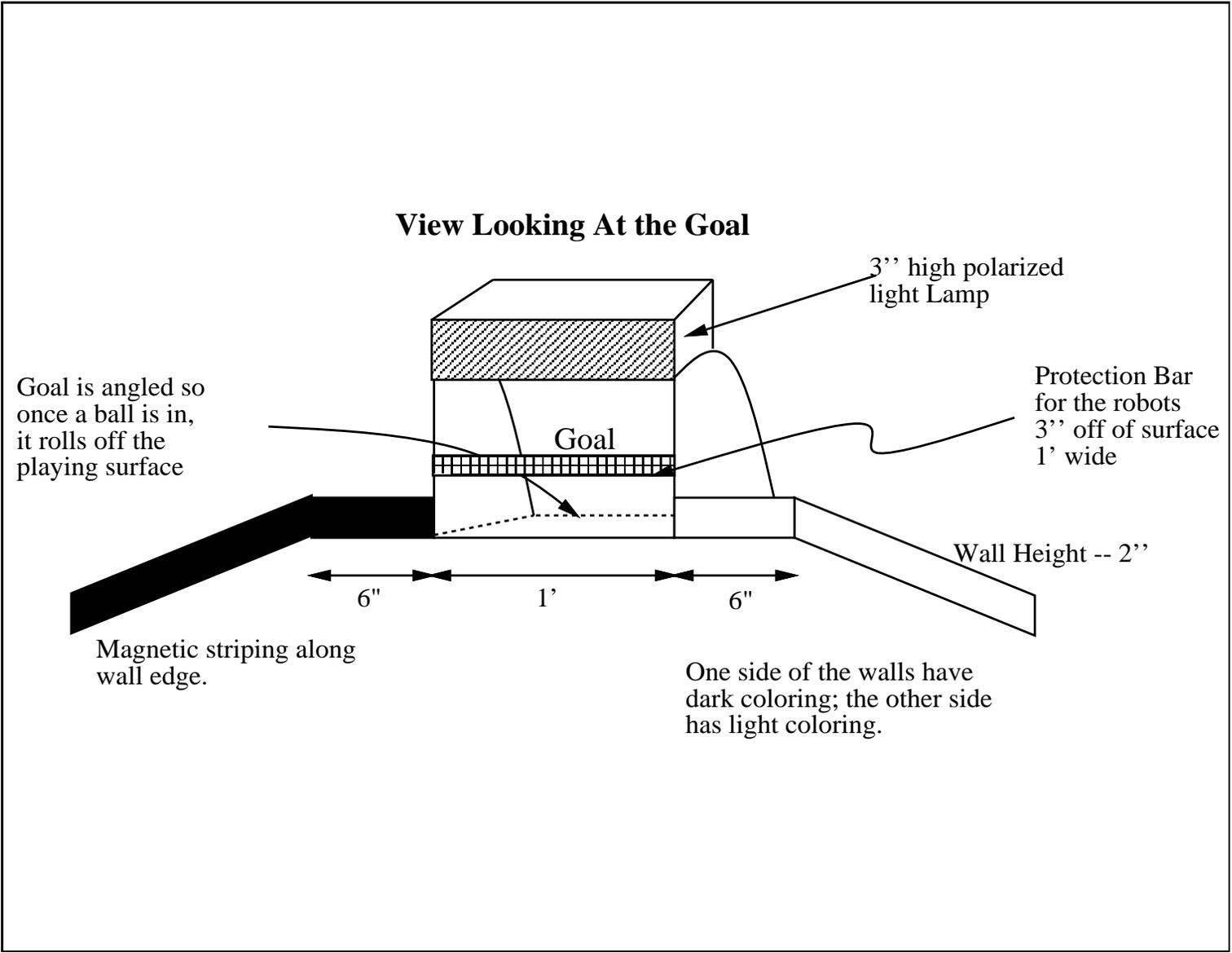


Figure A.3: Robo-Cup '92 Contest Goal Specification

### A.7.3 The Goal

- Your robot's goal is defined as the goal which is furthest from the robot's starting position.
- The goal is one foot wide and eight inches high with a barrier post that is one inch wide at a height of three inches above the surface.

See Figure A.3 for an illustration of the goal.

- Your robot may not advertently place anything inside either of the goals except the balls.
- The surface inside of the goal will be sloped so that the balls will roll into the goal when they cross the goal line.

### A.7.4 Period of Play

- The powered portion of a round will last 45 seconds. After the machines are started, they will have 45 seconds to apply battery power to their actuators.
- The round ends when all machines and balls come to rest.
- The round will be started by the judges turning on the starting lights, located underneath the table in the center of each robot's starting circle, for the first one second of the round.
- The contestants will place the machines on the playing field within the designated starting circles. The robots may be placed in any orientation within the starting area.
- The contestants will have 30 seconds to place their machines on the field from the time the judges call them.
- The contestants must stand a given distance away from the playing field. Any contestant who touches their machine during the round of play will automatically be disqualified their robot from the round.
- The machines must have their own internal clock (software will be provided to do this) that cuts off power to the motors at the end of 45 seconds. Any machines that continue to supply actuator power after 45 seconds will be disqualified.
- The contest will be an double elimination competition held over two nights. Machines must qualify for the second night of competition, as follows:

- *Night 1.* All machines will play one round. If a machine loses its round against an opponent, it will run against an inert placebo. If it cannot win against the inert placebo after two tries, it will not qualify for the second night of play.
  - *Night 2.* The main competition. Machines will play until they lose twice. Loss against opponent from the first night is included.
- All rounds will have two robot players.

### A.7.5 Control

- All entries must be solely controlled by their onboard computer. There can be no human intervention once the round begins.
- After the start of the contest, there can be no change to the robot's program or configuration switches made by the contestants.
- No parts or substances may be deliberately dumped, deposited, or otherwise left to remain on the playing surface. A machine that appears to have been designed to perform such a function will be disqualified.
- Machines are not allowed to destroy their opponent's microprocessor board.
- Machines cannot try to destroy other machines' broadcast or detection beacons.

### A.7.6 Infrared Beacon

All robots are required to carry an infrared transmitter. This transmitter acts as a beacon so that robots can locate each other on the playing field. The following rules describe the functionality of the infrared beacon.

- All entries must carry an infrared beacon that is capable of broadcasting infrared (IR) light at modulated at either 100 Hertz or 125 Hertz with a 40,000 Hertz carrier (hardware is provided to do this).
- Machines failing to meet the infrared transmission specification, or in any way modifying or jamming their transmission frequency during the round of play will be disqualified.
- Judges will assign frequencies for IR emitters to the machines in the beginning of each round by setting the robot's DIP switch 1. If the switch is one, the robot must broadcast 100 Hertz infrared light. If the switch is zero, the robot must broadcast 125 Hertz infrared light. Software will be provided to do this.

- The IR broadcasting beacon must be located at exactly one foot (12 inches) above the surface of the playing field when mounted on the robot.
- The beacon must be located so that its center is never more than four inches (measured horizontally) from the geometric center of the microprocessor board.
- The beacon cannot be deliberately obstructed, or be designed in such a way that “accidental” obstructions are probable.

### A.7.7 The Contest Playing Table

Figure A.2 illustrates the Robo-Cup playing field.

#### Polarized Light Goal Lamps

- Goals will have panels that emit polarized light at one of two orientations: a +45 degree and -45 degree (with respect to the vertical) polarization (see Figure A.3).
- The setting of DIP switch 1 will indicate which polarization angle is emitted by the robot's goal. If the switch is one, the robot's goal will be the one with the positive polarization. If the switch is zero, the robot's goal will be the one with negative polarization.

#### Wall Striping

- The two sides of the playing field will be coded with light or dark visible striping on the lower inside edge of the playing field wall.
- All wall edges will also be coded with magnetic strips.

#### Ball Dispensers

- The ball dispensers will drop balls at a distance of 15 inches above the designated drop points.
- Robots may obtain balls from either ball dispenser.
- Ball dispensers will dispense balls at a rate not greater than one ball per five seconds.
- Depressing the touch bumper near to the ball dispenser causes a ball to be dispensed as soon as is possible given the dispensing rate mentioned above.
- The touch bumper must be released before a subsequent ball can be dispensed.

### A.7.8 Structure

- All kits contain exactly the same components, with the exception that some LEGO parts may be colored differently in different kits.
- Only LEGO parts and connectors may be used as robot structure. LEGO rubber bands are counted as LEGO parts; therefore, LEGO rubber bands *may* be used to provide structural support to your machine.
- LEGO pieces may not be glued together.
- LEGO pieces may not be altered in any way, with the following exceptions:
  1. The grey LEGO baseplate may be altered freely.
  2. LEGO pieces may be modified to facilitate the mounting of sensors and actuators.
  3. LEGO pieces may be modified to perform a function directly related to the operation of a sensor. An example: Holes may be drilled into a LEGO wheel to help make an optical shaft encoder.
- String may not be used for structural purposes.
- The wooden dowel may be used only as a tower to mount the infrared transmitters and any receivers.
- A non-LEGO part may be attached to at most five LEGO parts via glue.
- Cardboard, other paper products, and tape may be used for the purpose of creating optical shields for light sensors.
- Wire may only be used for electrical purposes, and not structural.
- Rubber bands may be glued to LEGO wheels or gears to increase the coefficient of friction.
- Only the LEGO rubber bands and thin rubber bands may be used to provide stored energy.
- Contestants may not alter the structure of their entry once the contest has begun, but may repair broken components between rounds if time permits.
- The dimension of the machine may not exceed an imaginary 1 foot cube at the start of each round. Only the IR transmitting and receiving beacons and the bend sensors may protrude outside this volume. Entries may however expand once the round has begun.

- Entries may not drag wires between two or more structurally separate parts of their robot.

One portion of the robot is considered structurally separate from another if when the machine is lifted from a supporting surface and held from the other portion, the two portions are supported mainly by wire.

- No lubricants may be used.
- Cable ties may not be used for structural purposes.
- Some parts in the 6.270 kit are considered tools and may not be used on the robot. Examples are: the red plastic parts container; the small rectangular parts container; the soldering iron sponge. If there is any question about whether an object is a “kit part” or a “tool part,” ask the organizers.
- Any machine that appears to be a safety hazard will be disqualified from the competition.

### A.7.9 The \$10 Electronics Rule

To encourage creativity, contestants may spend up to \$10 of their own funds for the purchase of additional electronic components used in their design. Other than this rule, robots must be designed completely from standard kit parts. The following conditions apply to all non-kit-standard electronic additions:

- The following components, categories of components, or varieties of circuitry are *disallowed*:
  - Batteries of any variety.
  - Motor driver circuitry, including relays, power transistors, or any other replacements or modifications to the standard motor driver circuitry.
- No single part may cost more than \$2.
- Resistors rated less than 1 watt and capacitors valued less than 100  $\mu\text{F}$  may be used freely, without accounting toward the \$10 total.
- Contestants who add *any* non-kit parts to their project must turn in a design report that includes: description of the modification, schematic of all added circuitry, and store receipts for parts purchases. This design report must be turned in to the organizers by 5:00 pm, Friday, January 31, 1992. *Any machines found with added circuitry that has not been documented in this fashion will be disqualified.*

- If a contestant wishes to use an electronic part which has been obtained through other means than retail purchase, an equivalent cost value to the part will be assigned by the organizers. Contestants must obtain this cost estimate *in writing* from the organizers and include it in the design report mentioned above.

### A.7.10 Scoring

- Each ball entering the upper goal area will be awarded three points. Each ball entering the lower goal area will be awarded two points. The winner will be the machine with points at the end of the round.
- In rounds containing a placebo, the contestant's robot must score at least one ball into its goal in order to be declared the winner of that round.
- If no goals are scored a double loss will be awarded by the judges.
- If there is a tie at the end of the round, the win will be determined as the robot that has more balls on the half of the playing field nearer to its goal.
- A double win may be awarded at the judges' discretion.
- The judges will decide any discrepancies in the contest play.

### A.7.11 Organizers

- Contestants may approach the organizers in privacy to consult about possible designs that may be questionable under the rules listed above. These designs will not be divulged to any of the other contestants.
- Final arbitration of any rule disputes before the day of the contest (February 3rd) will be decided by the contest organizers—Fred Martin, Pankaj Oberoi, and Randy Sargent.

## A.8 Parts List

This section lists all of the parts in the 1992 6.270 kit, including pricing and source information.

The prices indicated for most parts represent large quantity discounts of up to 50% off of the single-quantity price.

Other prices are estimates for parts that were donated to the contest: The following companies donated parts at no charge to the contest: Motorola, Gates, 3M, Methode, and AGE.

LEGO Systems, Inc. provided parts to the contest at approximately a 50% to 70% discount off the retail price. Some of these parts were obtained packaged from the USA division; for these parts, stock numbers are indicated in the parts listing. Other LEGO parts were hand-picked for the 6.270 contest from the LEGO factories in their home country of Denmark.

Overall, the LEGO component of the 6.270 kit would be valued between \$150 and \$200 retail.

#PER KIT PART REF	DESCRIPTION	COST EA	COST KIT	SUPPLIER	PART NUM
INTEGRATED CIRCUITS					
1 U1	68HC11A0 microprocessor	\$7.00	\$7.00	Motorola	industry
1 U2	62256LP 32K static RAM	\$7.00	\$7.00	Motorola	industry
1 U3	74HC373 transparent latch	\$0.50	\$0.50	Motorola	industry
1 U4	74HC138 3-to-8 decoder	\$0.30	\$0.30	Motorola	industry
2 U5, U17	74HC374 8 bit latch	\$0.40	\$0.80	Motorola	industry
1 U6	74HC244 8 bit bus driver	\$0.40	\$0.40	Motorola	industry
1 U7	74HC132 quad schmitt trigger	\$0.30	\$0.30	Motorola	industry
1 U8	74HC4053 triple SPDT switch	\$0.50	\$0.50	Motorola	industry
1 U9	74HC10 triple 3-input NAND	\$0.30	\$0.30	Motorola	industry
1 U10	74HC390 dual decade counter	\$0.50	\$0.50	Motorola	industry
1 U11	LM386 power op-amp	\$0.70	\$0.70	Motorola	industry
1 U12	74HC04 hex inverter	\$0.30	\$0.30	Motorola	industry
3 U18,U19,U20	74HC4051 8PST analog switch	\$0.50	\$1.50	Motorola	industry
3 U13,U15,U21	SGS-Thomson L293D motor driver IC	\$1.95	\$5.85	Active	L293D
2 U14,U16	SGS-Thomson L293B motor driver IC	\$1.90	\$3.80	Nu Horizons	L293B
TRANSISTORS					
2 Q1,Q2	MPS2222A general-purpose transistor	\$0.07	\$0.14	Motorola	
DIODES AND LEDs					
5 D1,7,8,9,10	1N4001 power diode	\$0.05	\$0.25	Motorola	
5 D2,D3,D4, D5,D6	1N4148 signal diode	\$0.01	\$0.07	EBC	51-04148
1 BR1	bridge rectifier	\$0.25	\$0.25	All	FWB-201
14 LED1,2,3,4,5,6,13 15,17,18,21,22	HLMP1700 hi efficiency red LED	\$0.10	\$1.40	R&D	393
11 LED7,8,9,10,11,14, 16,19,20	HLMP1790 hi efficiency green LED	\$0.23	\$2.53	Active	HLMP1790
1 LED12	HLMP1719 hi efficiency yellow LED	\$0.23	\$0.23	Active	HLMP1719
8 LED23-30	clear lens mini red LED	\$0.06	\$0.52	EBC	55-32010
8 LED31-38	Motorola MLED71 infrared LED	\$0.22	\$1.76	Motorola	MLED71
INDUCTOR					
1 L1	1uH high current inductor	\$0.5355	\$0.54	Digikey	M7010
RESISTIVE DEVICES					
2 R14,15	1K resistor, 1/8 watt	\$0.0230	\$0.05	Digikey	1.0KE
3 R6,16,17	2.2K resistor, 1/8 watt	\$0.0230	\$0.07	Digikey	2.2KE
1 R5	3.3K resistor, 1/8 watt	\$0.0230	\$0.02	Digikey	3.3KE
3 R4,8,10	10K resistor, 1/8 watt	\$0.0230	\$0.07	Digikey	10KE
6 R1,2,7,9,12,13	47K resistor, 1/8 watt	\$0.0184	\$0.11	Digikey	47KE
1 R3,	100K resistor, 1/8 watt	\$0.0230	\$0.02	Digikey	100KE
1 R11	2.2M resistor, 1/8 watt	\$0.0230	\$0.02	Digikey	2.2ME
2 R18,R19	7.5 ohm, 5 watt power resistor	\$0.272	\$0.54	Digikey	7.5W-5
2 R20,R21	15 ohm, 2 watt power resistor	\$0.108	\$0.22	Digikey	15W-2
2 RP1,10	47K x 9 common (V) resistor pack	\$0.15	\$0.30	EBC	20-19547
1 RP2	47K x 4 common (V) resistor pack	\$0.07	\$0.07	EBC	20-14547
1 RP3	1K x 3 isolated (E) resistor pack	\$0.09	\$0.09	EBC	20-23410
1 RP4	1K x 5 common (V) resistor pack	\$0.15	\$0.15	EBC	20-15410
1 RP5	22K x 3 isolated (E) resistor pack	\$0.09	\$0.09	EBC	20-23522
1 RP6	1K x 7 common (V) resistor pack	\$0.12	\$0.12	EBC	20-17410
1 RP7	1.2K x 4 isolated (E) resistor pack	\$0.12	\$0.12	EBC	20-24410
2 RP8,9	47 ohm x 4 common (V) resistor pack	\$0.07	\$0.15	EBC	20-14247
1 RP11	47K x 7 common (V) resistor pack	\$0.15	\$0.30	EBC	20-17547
2 VR1,2	47K trimmer potentiometer	\$0.32	\$0.65	EBC	21-21550

Figure A.4: 6.270 Parts Listing, Page One of Four

CAPACITORS						
1	C3	4700pF monolithic capacitor	\$0.10	\$0.10	Active	31066
9	C4,7,8,10,12,16, 17,18,19	0.1uF monolithic capacitor	\$0.11	\$0.99	Active	SR205C104KAA
1	C14	1uF tantalum capacitor	\$0.20	\$0.20	Sprague	
1	C6	2.2uF tantalum capacitor	\$0.20	\$0.20	Sprague	
2	C9,11	4.7uF tantalum capacitor	\$0.20	\$0.40	Sprague	
2	C1,C2	10uF tantalum capacitor	\$0.20	\$0.40	Sprague	
1	C5	47uF electrolytic capacitor	\$0.09	\$0.09	EBC	65-24746
1	C15	220uF electrolytic capacitor	\$0.16	\$0.16	EBC	67-22256
1	C13	470uF electrolytic capacitor	\$0.24	\$0.24	EBC	67-24756
OSCILLATION						
1	XTAL	8.000 MHz ceramic resonator	\$0.7091	\$0.71	Digikey	PX800
1	PIEZO	beeper element	\$0.70	\$0.70	Time Elex	PKM17EPP-4001
SWITCHES						
1	SW1	CKC5100 DPDT slide switch	\$0.928	\$0.93	Digikey	CKC5100-ND
1	SW2	red pushbutton switch	\$0.83	\$0.83	Mouser	10KB-011
2	SW3,4	micromini pushbutton switch	\$0.33	\$0.66	Mouser	10KB-010
1	SW5	4 position DIP switch	\$0.40	\$0.40	MPJA	SW-1355
2	SW6,7	mini slide switch	\$0.26	\$0.52	Mouser	10SP001
4	SW8,9,10,11	3 position slide switch	\$0.33	\$1.33	All	SSW-11
CONNECTORS						
4	J1,3,4,5	2.1mm DC power jack, PC mount	\$0.34	\$1.36	Mouser	16PJ031
1	J2	4-pin modular jack, PC mount	\$0.45	\$0.45	All	MT-4J
2		2.1mm DC power plug	\$0.30	\$0.60	All	DCSID
4		36x1 female strip header	\$2.00	\$8.00	3M	
4		36x1 male strip header	\$1.25	\$5.00	3M	
1		4-pin modular plug	\$0.18	\$0.18	All	CMP-4
1		DEC-style modular plug	\$0.35	\$0.35	Altex	MP-6D
1		modular-to-spade-connector cord	\$0.55	\$0.55	Mendelson	550-5729F
1		DB-9 female connector	\$0.30	\$0.30	Mendelson	240-1100F
1		DB-9 connector hood & hardware	\$0.55	\$0.55	Mendelson	240-1091F
INTEGRATED CIRCUIT SOCKETS						
1	PLCC1	52 pin PLCC socket	\$2.75	\$2.75	Method	
1	DIP11	8-pin DIP socket	\$0.03	\$0.03	EBC	33-13084
3	DIP7,9,12,	14-pin DIP socket	\$0.06	\$0.18	EBC	33-13144
6	DIP4,8,10,18,19,20	16-pin DIP socket	\$0.07	\$0.41	EBC	33-13164
3	DIP13,15,21	16-pin machined DIP socket (gold)	\$0.556	\$1.67	Digikey	ED3316
3	DIP5,6,17	20-pin DIP socket	\$0.09	\$0.26	EBC	33-13204
1	DIP2	28-pin DIP socket	\$0.12	\$0.12	EBC	33-13284
BATTERY & CHARGER						
6		2 volt, 2.5 A/hr lead acid cell	\$3.00	\$18.00	Gates	
1		12v, 1.2A DC adapter	\$3.25	\$3.25	Edlie	TC626-8
SENSORS (and supporting electronics)						
4		Sharp GPIU52X sensor	\$1.30	\$5.20	Time	GPIU52X
8		resistive bend sensor	\$0.25	\$2.00	AGE	
5		super mini switch	\$0.15	\$0.75	MPJA	SW-1409
2		Omron D2F-L mini switch	\$0.65	\$1.30	All	SMS-282
5		lever micro switch	\$0.15	\$0.75	MPJA	3182-SW
4		125K pot	\$0.19	\$0.76	Elex Goldmine	G923
2		10K linear pot	\$0.25	\$0.50	All	LSP-10K
4		digital hall effect switches	\$0.39	\$1.56	R&D	#239
4		break-beam optical switch	\$0.50	\$2.00	Motorola?	
10		330 ohm, 1/8 watt resistors	\$0.0184	\$0.18	Digikey	330KE
4		super bright clear lens red LED	\$0.25	\$1.00	Elex Goldmine	A1012
4		super bright green LED	\$0.25	\$1.00	Elex Goldmine	A1018
2		mini high brightness lamp (3 to 6v)	\$0.22	\$0.44	Elex Goldmine	A1036
6		CdS photocell	\$0.22	\$1.32	Elex Goldmine	G990
6		reflective IR sensor	\$0.50	\$3.00	All	OSR-4
2		small magnet	\$0.33	\$0.67	All	MAG-3
MOTORS AND SOLENOIDS						
6		Polaroid 5v motor	\$5.00	\$30.00	Polaroid	

Figure A.5: 6.270 Parts Listing, Page Two of Four

MISCELLANEOUS				
1	4-AA cell battery holder	\$0.80	\$0.80	All BH-4AF
1	16x2 LCD display (non-backlit)	\$8.00	\$8.00	Timeline 16x2 LCD
3	heat sink	\$0.393	\$1.18	Digikey HS125
0.033	heat shrink tubing, 1/8" (60' spool)	\$21.95	\$0.73	MCM 95-310
0.05	heat shrink tubing, 1/4" (40' spool)	\$21.95	\$1.10	MCM 95-320
1	17-bin parts caddy	\$2.75	\$2.75	MCM 21-1665
2	red cylindrical parts caddy	\$0.40	\$0.80	Elex Goldmine G727
1	miniature parts case	\$0.39	\$0.39	Edlie TD2110-16
3	hex nylon spacers, 1/2" x 6-32	\$0.17	\$0.51	Mouser 561-L6.50
6	nylon screws, 1/4" x 6-32	\$0.05	\$0.30	Mouser 561-J632.25
2	nylon screws, 1/2" x 4-40	\$0.05	\$0.10	Mouser 561-J440.5
2	nylon nuts, 4-40	\$0.08	\$0.16	Mouser 561-H440
1	5-minute epoxy	\$2.33	\$2.33	Radio Shack 64-2313
1	super glue	\$1.15	\$1.15	MCM 20-350
1	6.270 kit box	\$2.00	\$2.00	Staples?
0.1	releasable head cable ties (per 100)	\$8.90	\$0.89	MCM 21-1415
8	4-conductor modular phone cable	\$0.05	\$0.36	All 4C/F
8	9-conductor ribbon cable (feet)	\$0.14	\$1.14	Digikey HC09M-ND
0.01	2-conductor power cable (feet)	\$0.03	\$0.00	R&D #B102
TOOLKIT				
1	Tenna grounded soldering iron	\$6.25	\$6.25	MCM 21-940
1	extra tip for soldering iron	\$1.15	\$1.15	MCM 21-435
1	soldering iron stand	\$4.10	\$4.10	MCM 21-180
1	"nippy" cutters	\$4.95	\$4.95	MCM 22-1050
1	wire strippers	\$3.10	\$3.10	MCM 22-555
1	desoldering pump	\$4.75	\$4.75	MCM 21-590
1	helping hands unit	\$4.00	\$4.00	MCM 21-425
1	mini needle nose pliers	\$3.90	\$3.90	MCM 22-575
1	soldering iron tip cleaner sponge	\$0.33	\$0.33	Elex Goldmine S2035
1	orange utility knife	\$1.25	\$1.25	Radio Shack 64-1868
2	safety goggles	\$2.00	\$4.00	MCM 21-855
LEGO PARTS				
1	small pulley, stop bush			LEGO USA 1314
1	piston rod			LEGO USA 1315
0.5	connector peg			LEGO USA 1316
1	chain link			LEGO USA 1317
1	8-, 16-, 24-tooth gears			LEGO USA 1318
2	24-, 40-tooth gears			LEGO USA 1319
0.5	bevel gears			LEGO USA 1320
1	worm gear, gear rack			LEGO USA 1321
1	O-ring, pulley wheels			LEGO USA 1322
1	large tire, small tire, hubs			LEGO USA 1323
1	rubber bands			LEGO USA 1324
1	4-, 6-, 8-long axles			LEGO USA 1325
1	10-, 12-long axles			LEGO USA 1326
1	1x3, 1x4 plates			LEGO USA 1327
1	1x6, 1x8 plates			LEGO USA 1328
1	2x3, 2x4 plates			LEGO USA 1329
1	2x6, 2x8 plates			LEGO USA 1330
1	1x2, 1x4 Technic beams			LEGO USA 1331
2	1x6, 1x8 Technic beams			LEGO USA 1332
2	1x12, 1x16 Technic beams			LEGO USA 1333
1	u-joint, piston, round plate			LEGO USA 1339
1	angle plate, hinge, turntable			LEGO USA 1338
2	misc. Technic parts			LEGO USA 1345
0.33	24x24 base plate (three units)			LEGO USA 1348

Figure A.6: 6.270 Parts Listing, Page Three of Four

2	81.6x15 tires	LEGO Denmark	factory
2	hub for 81.6x15 tire	LEGO Denmark	factory
4	hinged plates	LEGO Denmark	factory
4	1x5 plate with end holes	LEGO Denmark	factory
4	20x30mm tires	LEGO Denmark	factory
4	hubs for 20x30 tires	LEGO Denmark	factory
12	1x2 plate (black)	LEGO Denmark	factory
12	2x2 plate (black)	LEGO Denmark	factory
12	2x2 plate (black), flat top	LEGO Denmark	factory
4	optosensor encoder wheels	LEGO Denmark	factory
6	4x6 plate (assorted colors)	LEGO Denmark	factory
6	4x12 plate (assorted colors)	LEGO Denmark	factory
6	6x6 plate (assorted colors)	LEGO Denmark	factory
6	6x12 plate (assorted colors)	LEGO Denmark	factory
4	1x4 flat tile (black)	LEGO Denmark	factory
6	1x8 flat tile (black)	LEGO Denmark	factory
1	4x4 turntable (black)	LEGO Denmark	factory
12	toggle joint with tooth	LEGO Denmark	factory
20	2x2 brick, assorted colors	LEGO Denmark	factory
20	2x3 brick, assorted colors	LEGO Denmark	factory
60	2x4 brick, assorted colors	LEGO Denmark	factory

TYPE	INDEX	COMPANY INFORMATION
manufact	3M	3M
retail	Active	Active Electronics orders: (617) 932-4616
surplus	All	All Electronics Corp. P.O. Box 567, Van Nuys, CA 91408 orders: (800) 826-5432 info: (818) 904-0524 fax: (818) 781-2653
retail	Altex	Altex Electronics, Inc. 11342 IH-32 North, San Antonio, TX 78233 orders: (800) 531-5369 info: (512) 637-3200 fax: (512) 344-2985
retail	EBC	Electronic Buyer's Club
surplus	Edlie	Edlie Electronics 2700 Hempstead Turnpike, Levittown, NY 11756-1443 orders: (800) 645-4722 info: (516) 735-3330 fax: (516) 731-5125
surplus	Elex Goldmine	The Electronic Goldmine
manufact	Gates	Gates Energy Products, Inc.
retail	MCM	MCM Electronics 650 Congress Park Dr., Centerville, OH 45459-4072 orders: (800) 543-4330 info: (513) 434-0031 fax: (513) 434-6959
surplus	Mendelson	Mendelson Electronics Co, Inc. 340 East 1st St., Dayton, OH 45402 orders: (800) 422-3525 info: (513) 461-3525 fax: (513) 461-3391
manufact	Methode	Methode, Inc.
manufact	Motorola	Motorola Semiconductor, Inc.
surplus	MPJA	Marlin P. Jones & Assoc., P.O. Box 12685, Lake Park, FL 33403-0685 orders and info: (407) 844-8236 fax: (407) 844-8764
manufact	Polaroid	Polaroid, Inc.
surplus	R&D	R&D Electronics 1224 Prospect Avenue, Cleveland, OH 44115 orders: (800) 642-1123 info: (216) 621-1121 fax: (216) 621-8628
retail	Radio Shack	Radio Shack 178 Alewife Brook Pkway, Cambridge MA 02138 (617) 491-2925 attn: Don
distr'er	Time Elex	Time Electronics
surplus	Timeline	Timeline Inc. 1490 W. Artesia Blvd., Gardena, CA 90247 orders: (800) 872-8878 info: (213) 217-8912 fax: (213) 532-6304

Figure A.7: 6.270 Parts Listing, Page Four of Four

# Appendix B

## 6.270 Hardware

This chapter is partly tutorial and partly technical reference: in addition to documenting the 6.270 hardware, it explains the design in a way that would be understandable to the beginner. The discussion does however assume familiarity with some ideas of digital electronics.

The information presented here should be considered optional, as it is not strictly necessary to know it to build a robot. Hopefully though, this chapter will satisfy most readers' curiosity about how the 6.270 hardware works.

### B.1 The Microprocessor and Memory

At the most primitive level, a computer consists of a microprocessor, which executes instructions, and a memory, in which those instructions (and other data) is stored.

Figure B.1 shows a block diagram of these two components. The diagram shows four types of wires that connect the microprocessor and the memory:

**Address Bus.** These wires are controlled by the microprocessor to select a particular location in memory for reading or writing.

The 6.270 board uses a memory chip that has 15 address wires. Since each wire has two states (it can be a digital one or a zero), 2 to the 15th power locations are possible.  $2^{15}$  is precisely 32,768 locations; thus, the system has 32K of memory.

**Data Bus.** These wires are used to pass data between the microprocessor and the memory. When data is written to the memory, the microprocessor drives these wires; when data is read from the memory, the memory drives the wires.

In our example (and in the 6.270 board), there are eight data wires (or bits). These wires can transfer  $2^8$  or 256 different values per transaction. This data word of 8 bits is commonly referred to as a *byte*.

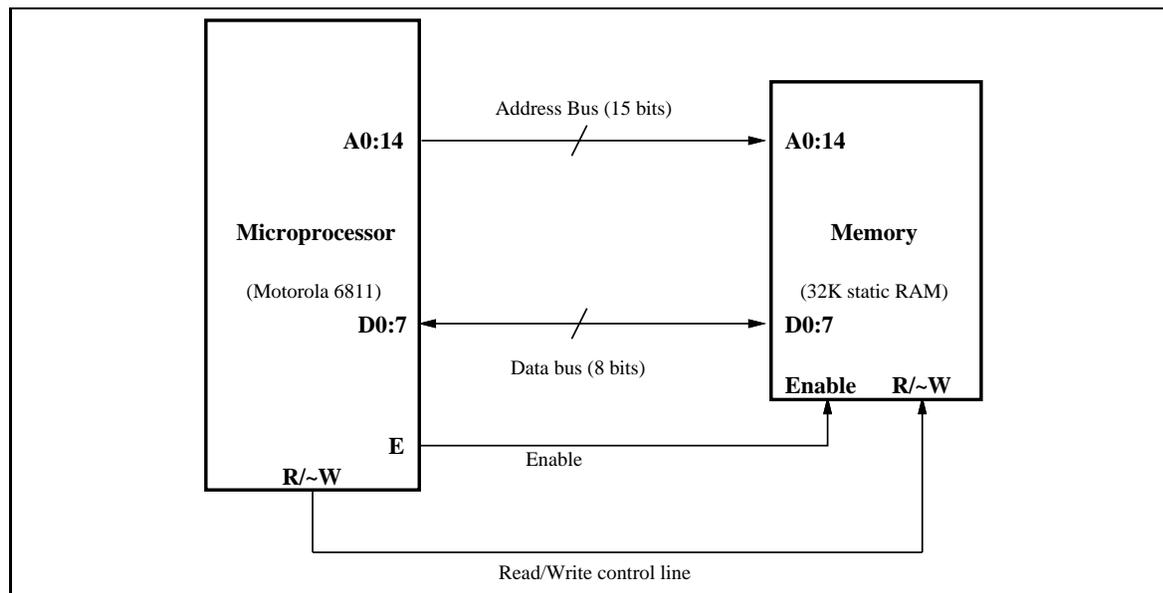


Figure B.1: Block Diagram of Microprocessor and Memory

**Read/Write Control Line.** This single wire is driven by the microprocessor to control the function of the memory. If the wire is logic true, then the memory performs a “read” operation. If the wire is logic zero, then the memory performs a “write operation.”

**Memory Enable Control Line.** This wire, also called the *E clock*, connects to the enable circuitry of the memory. When the memory is enabled, it performs either a read or write operation as determined by the read/write line.

### B.1.1 Multiplexing Data and Address Signals

Things are a little more complex with the particular microprocessor that is used in the 6.270 board, the Motorola 6811. On the 6811, The eight data bus wires take turns functioning as address wires as well.

When a memory location is needed (for reading or writing), first the data wires function as address wires, transmitting the eight lower-order bits of the address. Then they function as data wires, either transmitting a data byte (for a write cycle) or receiving a data byte (for a read cycle). All this happens very fast; 2 million times per second to be exact.

The memory needs to help to deal with the split-personality data/address bus. This help comes in the form of an *8-bit latch*. This chip (the 74HC373) performs the function of latching, or storing, the 8 address values so that the memory will have the full 15-bit address available for reading or writing data.

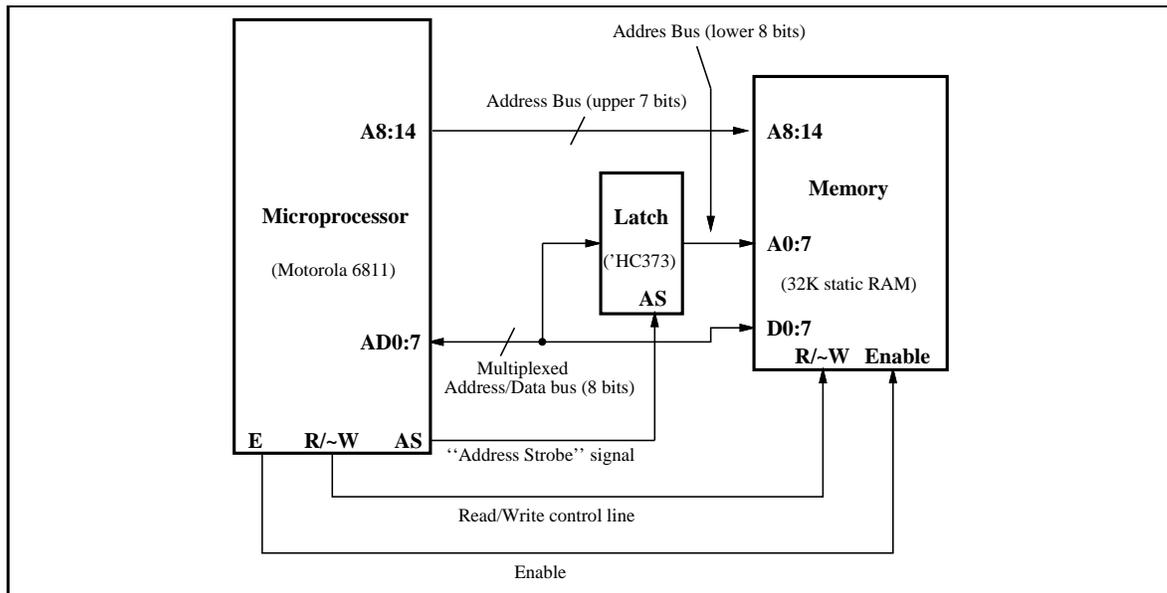


Figure B.2: Block Diagram of Microprocessor and Memory with Latch

Figure B.2 shows how the latch is wired. The upper 7 address bits are normal, and run directly from the microprocessor to the memory. The lower 8 bits are the split-personality, or, more technically, *multiplexed* address and data bus. These wires connect to the inputs of the latch and also to the data inputs of the memory.

An additional signal, the *Address Strobe* output of the microprocessor, tells the latch when to grab hold of the address values from the address/data bus.

When the full 15-bit address is available to the memory (7 bits direct from the microprocessor and 8 bits from the latch), the read or write transaction can occur. Because the address/data bus is also wired directly to the memory, data can flow in either direction between the memory and the microprocessor.

This whole process—the transmitting of the lower address bits, the latching of these bits, and then a read or write transaction with the memory—is orchestrated by the microprocessor. The E clock, the Read/Write line, and the Address Strobe line perform in tight synchronization to make sure these operations happen in the correct sequence and within the timing capacities of the actual chip hardware.

## B.2 Memory Mapping

So far we have seen how a memory can be connected to the address space of a microprocessor. In a circuit like the one of the 6.270 board, the microprocessor must interact with other devices than the memory—for example, motors and sensors.

A typical solution uses 8-bit latches for input and output. These latches are

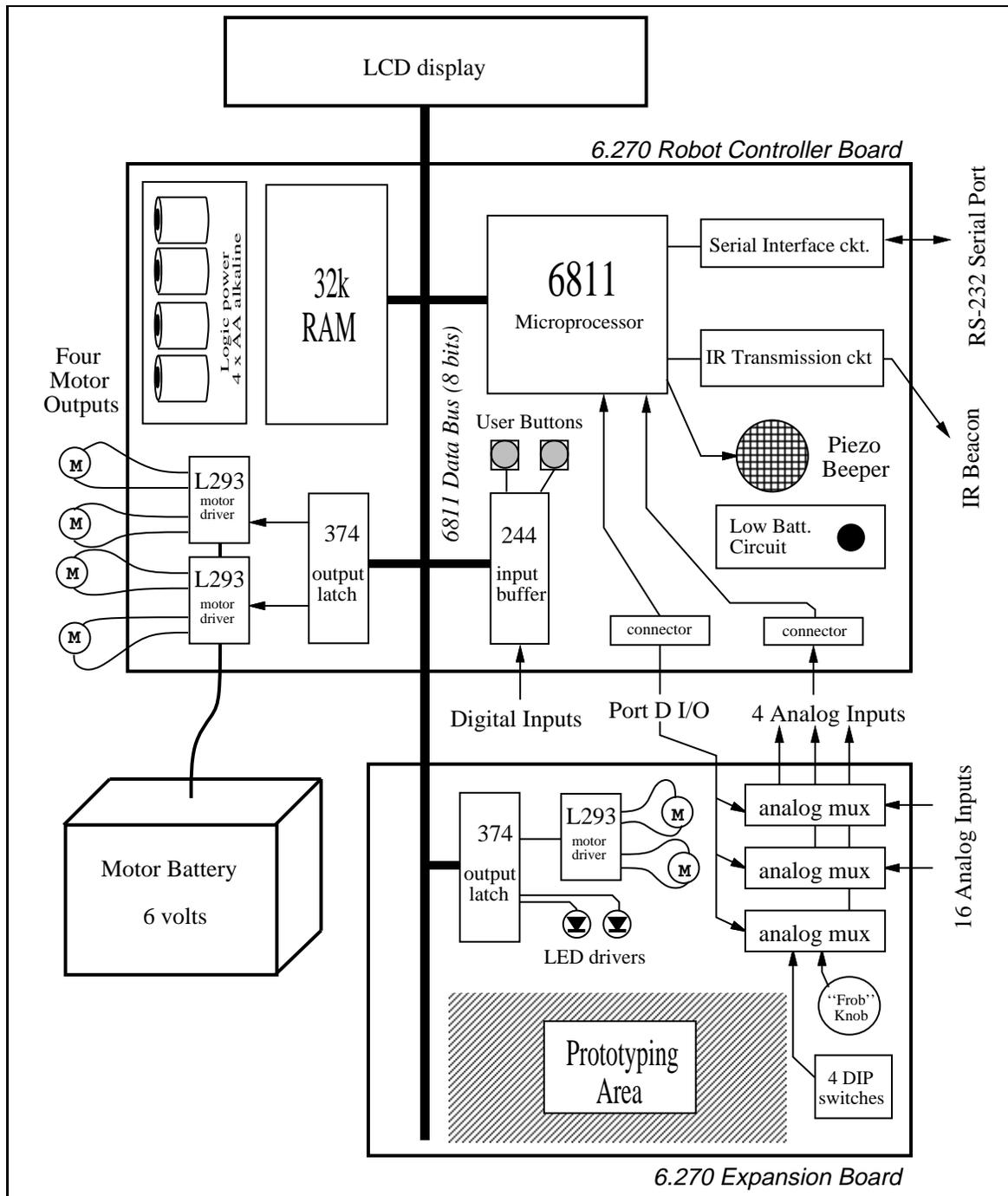


Figure B.3: 6.270 System Block Diagram

connected to the data bus of the microprocessor so that they appear like a location in memory. Then, the act of reading or writing from one of these memory locations causes data to be read from or written to a latch—to which the external devices are connected.

Figure B.3 is a block diagram of the 6.270 Robot Controller Board system. Following the present discussion that concerns how the motors and sensors are addressed by the microprocessor, notice that a chip labelled “374” is connected to the data bus. The ’374 has outputs that control the motors (through chips labelled “L293,” which will be discussed later). The digital sensors are driven into the data bus by a chip labelled “244.” On the expansion board, another 374 chip is used for eight bits of digital output.

These interface latch chips are used in a technique called *memory mapping*. The chips are “mapped” to a particular address in the microprocessor’s memory.

The following discussion will show how both the 32k RAM memory and the digital input and output latch chips share the address space of the microprocessor.

### B.2.1 Memory-Mapping the RAM

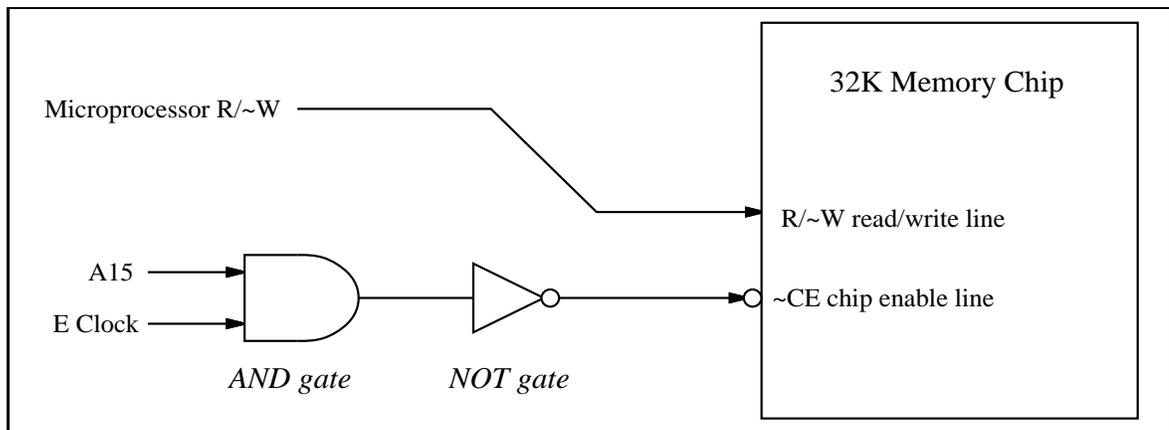


Figure B.4: Enabling the Memory

The 6811 has a total of 16 address bits, yielding 64K bytes of addressable locations (65536, to be exact). Half of this space will be taken up by the 32K memory chip (also known as a *RAM* chip, for “random access memory”).

The 6811 has a bank of *interrupt vectors*, which are hardware-defined locations in the address space that the microprocessor expects to find pointers to driver routines. When the microprocessor is reset, it finds the reset vector to determine where it should begin running a program.

These vectors are located in the upper 32K of the address space. Thus, it is logical to map the RAM into this upper block, so that the RAM may be used to store these vectors.

The technique used to map the memory to the upper 32K block is fairly simple. Whenever the 6811's A15 (the highest-order address bit) is logic one, an address in the upper 32K is being selected. The other fifteen address bits (A0 through A14) determine that address.

A logic gate is used to enable the memory when A15 is logic one *and* when the E clock is high (since the E clock must control the timing of the enable). Figure B.4 shows a block diagram of this circuit. (The actual circuit to enable the RAM, shown in Figure B.9, is slightly more complex due to considerations of battery-protecting the memory, as explained later.)

Memory chips are part of a class of chips that have *negative true* enable inputs. This means that they are enabled when the enable input is logic zero, not logic one.

There are two methods for denoting an input that is negative true. As shown in Figure B.4, the chip enable input is shown with connecting to a circle. This circle indicates a negative true input. Also, the name for the signal, *CE* is prefixed with a  $\sim$  symbol.

The function of the NOT gate shown in the diagram is to convert the positive-true enable produced by the AND gate into the negative-true signal required by the  $\sim$ CE input. (Often these two gates are collapsed into a single NAND gate.)

## B.2.2 Memory-Mapping with the 74HC138 Chip

Figure B.5 shows the 74HC138 chip, which is commonly used in circuits that map devices onto an address space. This chip is a 3-to-8 decoder: a binary number of three digits (the *select inputs*) causes one of eight possible outputs to be selected (the *control outputs*). The chip also has three *enable inputs*, all of which must be enabled to make the chip become active.

The outputs of the '138 chip control the input and output latches shown in the system block diagram. The '138 determines when these latches are activated, either to read data from the data bus (in the case of the '374 output latch), or to write data onto the data bus (in the case of the '244 input latch).

### Enable Inputs

The enable inputs of the '138 determine *when the chip will become active*, and thereby turn on one of the input or output latches. These enables inputs are critical because the '138 must not become active at the same time as the RAM chip. In it did, then two devices (the RAM and perhaps a '244) would attempt to drive the data bus simultaneously, causing a problematic situation called *bus contention*.

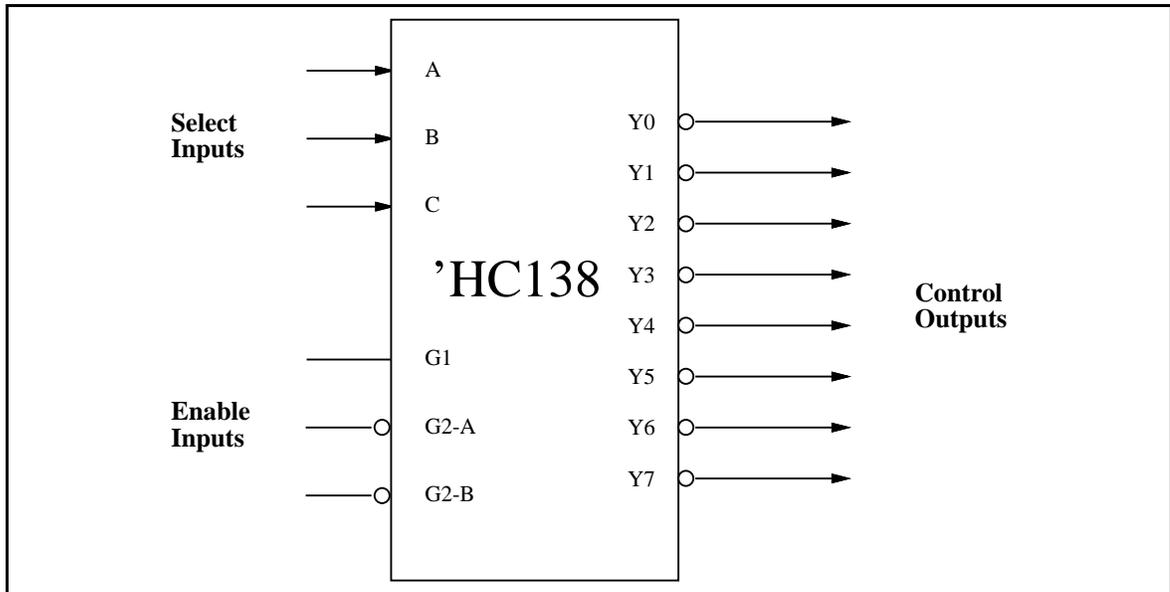


Figure B.5: The 'HC138 Address Decoder

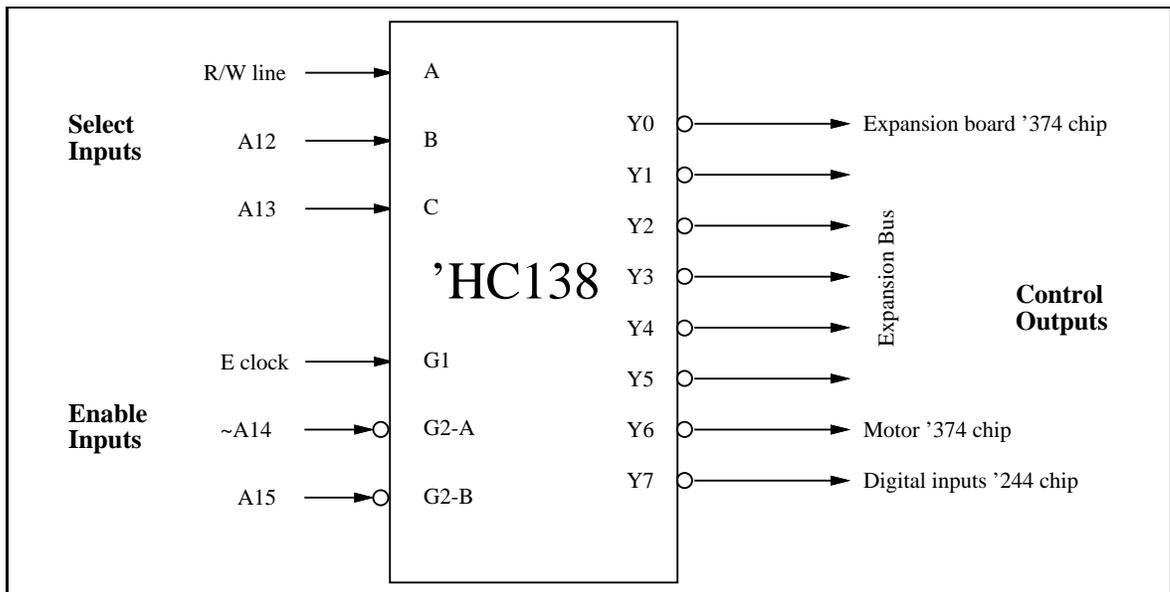


Figure B.6: Wiring the 'HC138 Address Decoder

As shown in Figure B.6, A15, the highest order address bit, is connected to a *negative enable* of the '138. Thus A15 must be *zero* to enable the chip. Since the RAM is enabled only when A15 is one (as was explained earlier), there is no chance that the '138 and the RAM could be active at the same time.

$\sim$ A14, which is the logical inverse of A14, is connected to a second negative enable of the '138. Thus when A14 is one,  $\sim$ A14 is zero, and the G2-A enable is true. So A14 must be one in order to active the '138.

The final enable input is positive true, and is connected to the 6811 E clock. When A15 is zero and A14 is one, the E clock will turn on the '138 at the appropriate time for standard 6811 read/write cycles.

### Select Inputs

Given that the '138 is enabled, the A, B, and C inputs determine which device connected to its outputs will be activated. A, B, and C form a binary number (C is the most significant bit) to determine the selected output.

The A13 and A12 address bits and the 6811 read/write line make the selection. Suppose A13 and A12 are one. The read/write line makes the final choice. This line is one for a read and zero for a write. If a read operation is in progress, then the ABC inputs will form the number 7, and the Y7 output will be activated. As shown in Figure B.6, this output connects to the digital input '244 chip. So, the '244 chip will turn on and will drive a byte onto the data bus. The read operation will complete with this byte having been read from the location in 6811 address space that was selected.

Notice that address bits A0 through A11 have no effect on the operation just described. As long as A15 is zero, A14, A13, and A12 are one, a read operation will cause the '138 to turn on the digital input '244 chip to write a byte onto the data bus. Thus, the digital input chip is selected by a read from any address from \$7000 to \$7FFF<sup>1</sup>. This is fairly wasteful of the address space of the 6811, but keep in mind that the only circuitry required to arrange this solution was the '138 chip.

Suppose a write operation were to occur in that same range of memory. The relevant upper four address bits would have the same values, but the read/write line would be zero (indicating the write operation). Thus the '138 ABC inputs would form the number 6, and output Y6 would be activated. Y6 is connected to the '374 chip that controls the motors; thus, the '374 would latch the value present on the data bus during the write operation.

As shown in Figure B.6, most of the '138 outputs are still available for future expansion. The 6.270 Expansion Board includes a circuit with one '374 chip, connected to the Y0 output. Outputs Y1 through Y5 are left free for further expansion use.

---

<sup>1</sup>These numbers are expressed in the hexadecimal numbering system, in which each digit represents a four-bit value from zero (0) to fifteen (F)

### B.2.3 System Memory Map

Figure B.7 summarizes the memory map solution that has been implemented for the 6.270 Board.

The 32K RAM takes up half of the total address space of the microprocessor. As indicated in the map, it is located in the upper 32K of the microprocessor's memory, from addresses \$8000 to \$FFFF.

The four digital input and output ports are mapped at locations starting at \$4000, \$5000, \$6000, and \$7000.

There is small area of memory that is internal to the 6811 chip itself. This memory consists of 256 bytes located at the start of the address space, from locations \$00 to \$FF.

The 6811 also has a bank of 64 internal *special function registers*, located at addresses \$1000 to \$103F. These registers control various hardware features of the 6811 (the analog inputs and serial communications are two examples).

The remainder of this section presents details on the digital input and output circuit wiring.

### B.2.4 Digital Inputs

Figure B.8 shows the digital input circuitry. U6, a 74HC244 chip, is used to latch an eight-bit word of sensor inputs and drive the 6811 data bus with that value when the chip is selected.

The '244 chip has two halves which may be separately enabled. The Y7 select is connected to both enable inputs, so that both halves of the chip are always selected simultaneously.

The lower two bits of the '244 are connected to the two user buttons (which have been dubbed CHOOSE and ESCAPE). The upper six bits are connected to the digital input header.

The lower two bits of the input header are connected to two timer inputs inputs of the 6811. These inputs can be used to precisely measure waveforms, or can simply be used for digital input. The library functions written to perform digital inputs insulate the user from the fact that the eight pins on the input header are not mapped contiguously to one location in memory.

RP1, a 47K resistor pack, acts as pull-up resistors to the inputs of the '244 chip, making the default values of the inputs one.

### B.2.5 Digital Outputs

Figure B.14 shows the complete schematic for the '374 output latch controlling the motors. For the purpose of the discussion to this point, notice that the data inputs

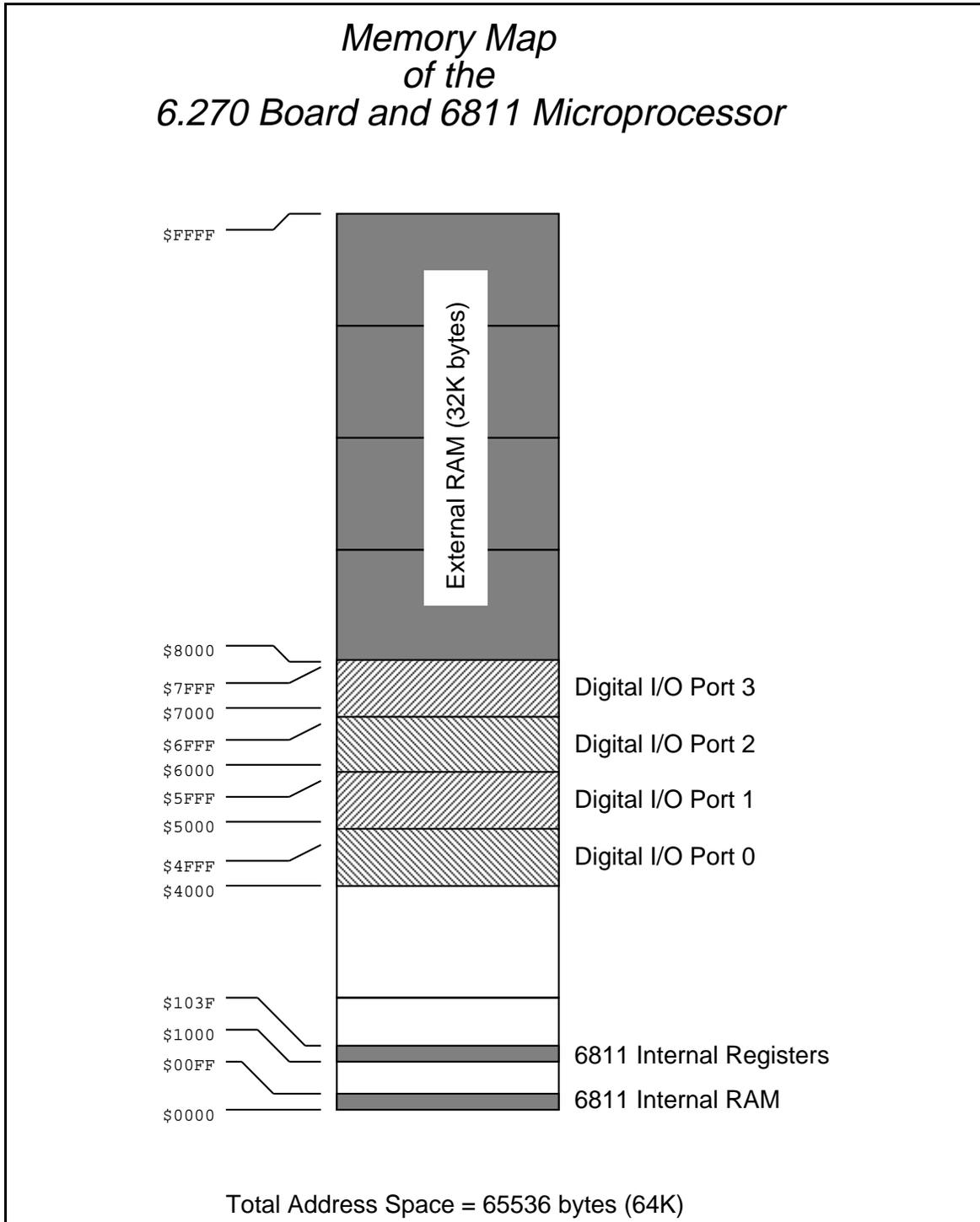


Figure B.7: 6811 System Memory Map

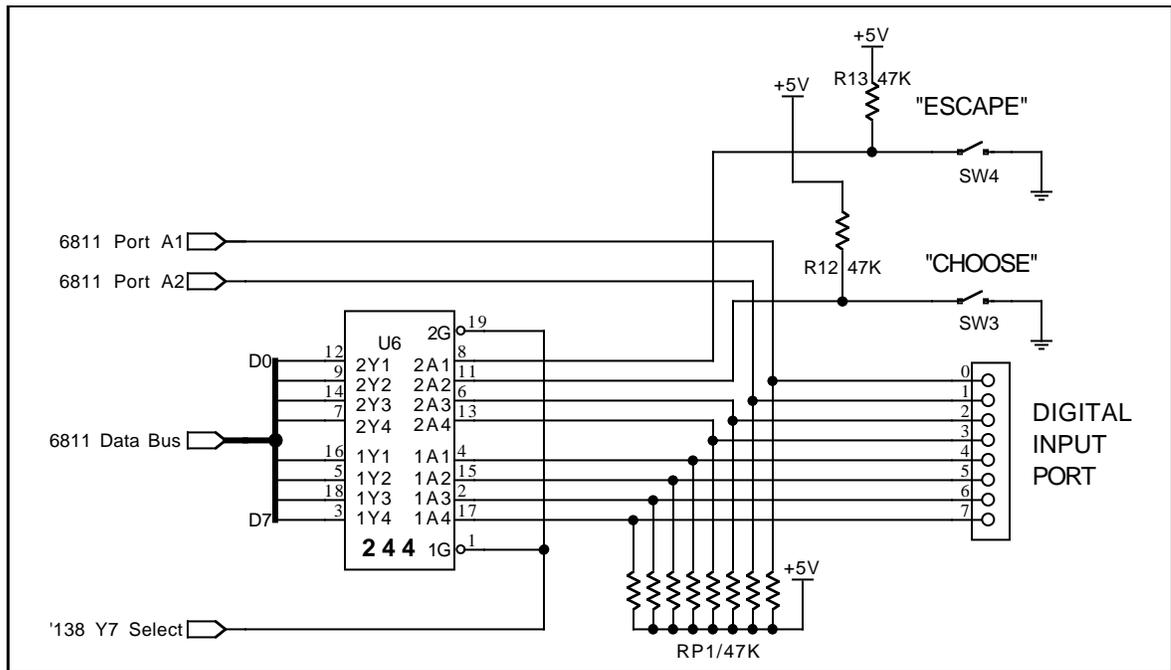


Figure B.8: Digital Input Circuit

'374 are connected to the 6811 data bus. The Y6 select signal connects to the clock input of the '374; when Y6 is activated, the '374 latches the value present on the data bus.

The outputs of the '374 connect to the motor driver chips. This circuitry is explained in the following section.

Figure B.16 is the schematic of the motor circuit present on the 6.270 Expansion Board.

### B.2.6 6811 and Memory Schematic

Figure B.9 presents the schematic of the 6811, memory, address decoding, and supporting main circuitry on the 6.270 Processor Board. By the end of this chapter, most of the circuitry depicted here will be explained.

## B.3 The Motor Drivers

Motors are high-powered devices in the world of digital electronics. A typical digital output can supply about 10 to 20 milliamperes (mA) of current; a small permanent-magnet motor requires anywhere from 500 to 4000 mA of current. It should not come as a surprise that special circuitry is required to drive motors.

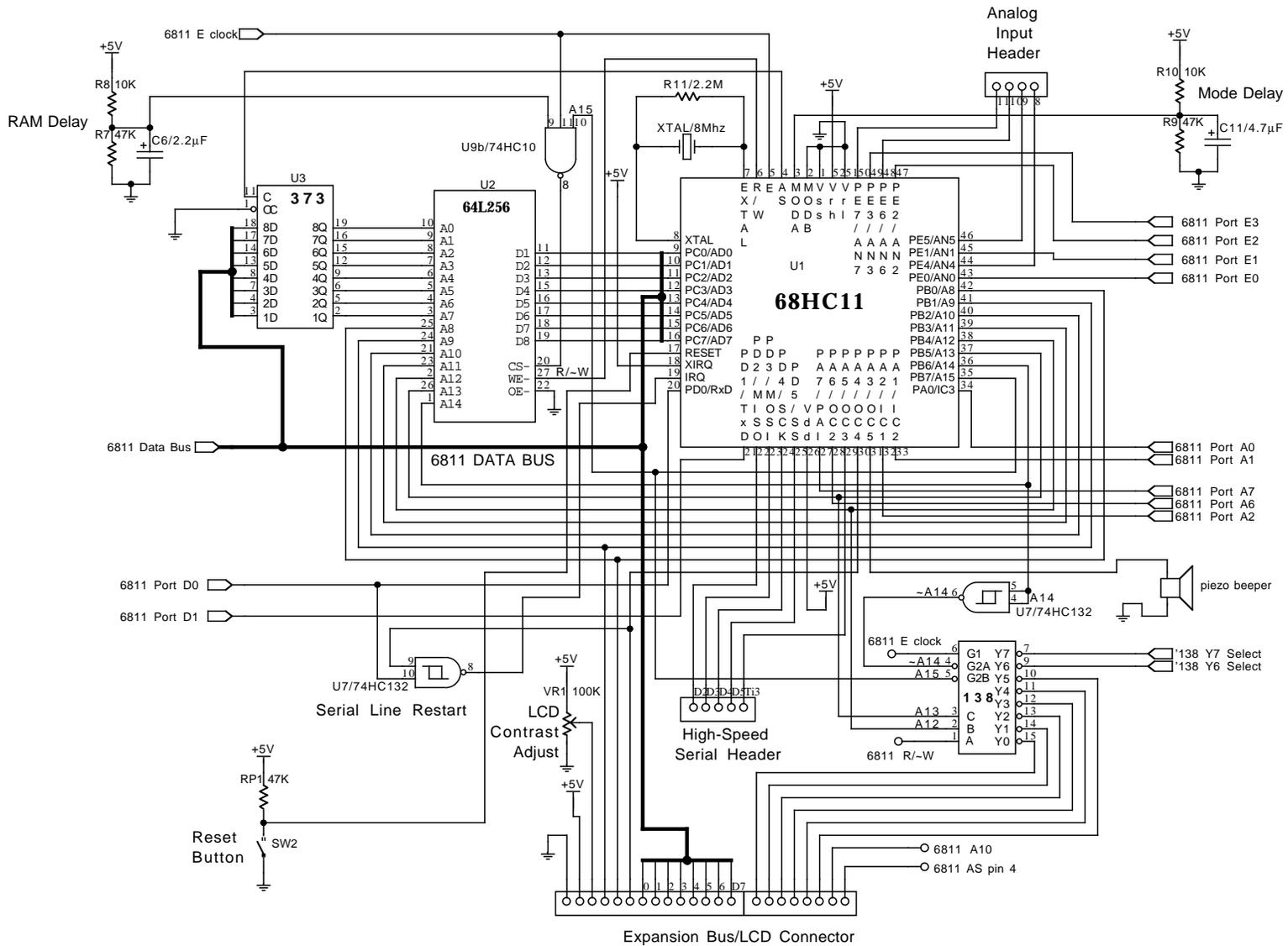


Figure B.9: 6811, Memory, Address Decoding and Miscellaneous Circuitry

### B.3.1 The H-Bridge Circuit

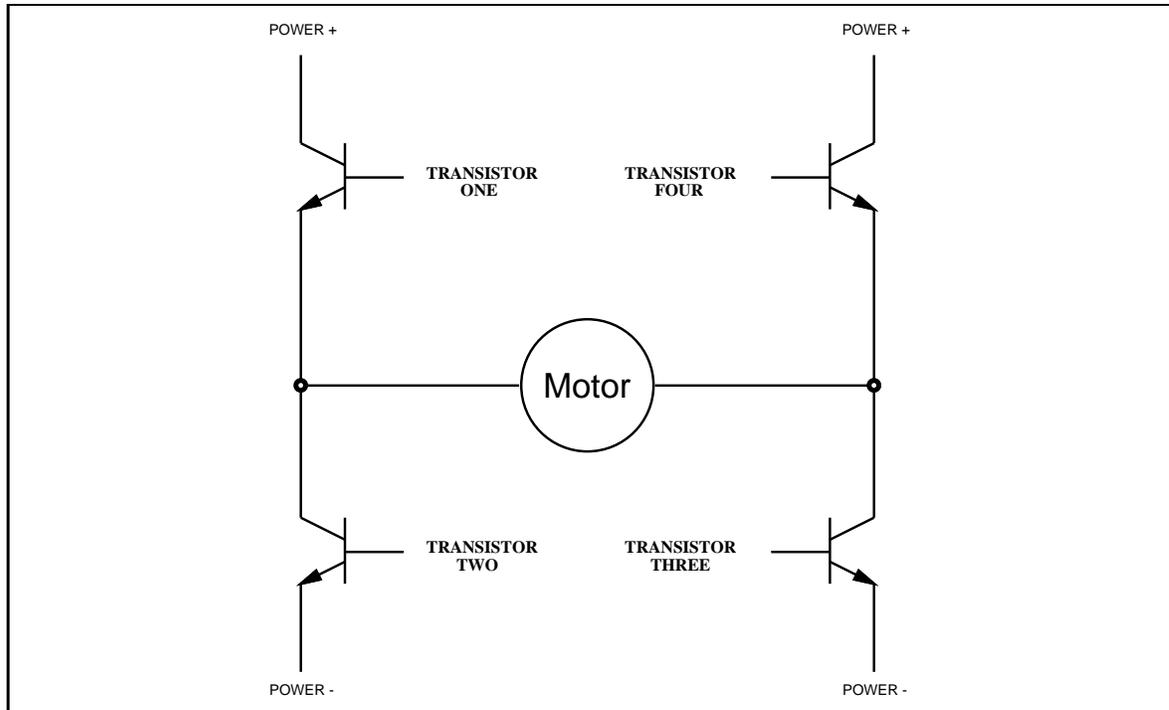


Figure B.10: The H-Bridge Circuit

A circuit known as the *H-bridge* (named for its topological similarity to the letter “H”) is commonly used to drive motors. In this circuit (depicted in Figure B.10), two of four transistors are selectively enabled to control current flow through a motor.

As shown in Figure B.11, an opposite pair of transistors (Transistor One and Transistor Three) is enabled, allowing current to flow through the motor. The other pair is disabled, and can be thought of as out of the circuit.

By determining which pair of transistors is enabled, current can be made to flow in either of the two directions through the motor. Because permanent-magnet motors reverse their direction of turn when the current flow is reversed, this circuit allows bidirectional control of the motor.

### B.3.2 The H-Bridge with Enable Circuitry

It should be clear that one would never want to enable Transistors One and Two or Transistors Three and Four simultaneously. This would cause current to flow from Power+ to Power- through the transistors, and not the motors, at the maximum current-handling capacity of either the power supply or the transistors.

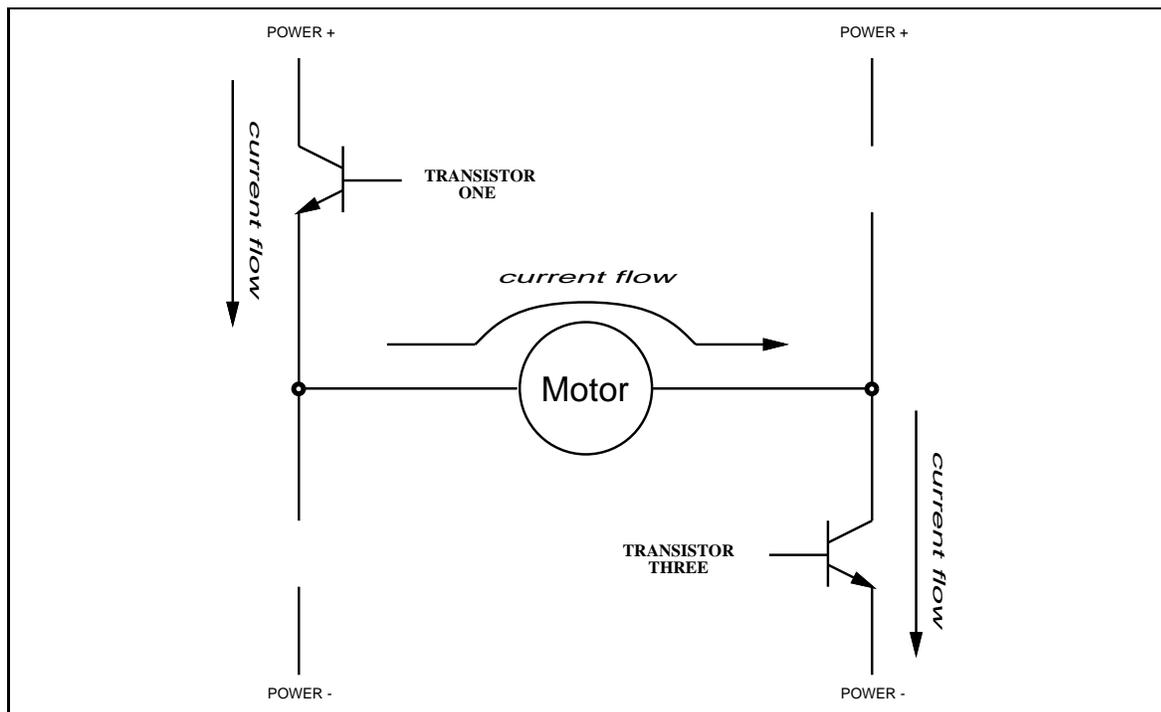


Figure B.11: The H-Bridge with Left-to-Right Current Flow

To facilitate control of the H-bridge circuit, enable circuitry as depicted in Figure B.12 is typically used.

In this circuit, the inverters ensure that the vertical pairs of transistors are never enabled simultaneously. The *Enable* input determines whether or not the whole circuit is operational. If this input is false, then none of the transistors are enabled, and the motor is free to coast to a stop.

By turning on the *Enable* input and controlling the two *Direction* inputs, the motor can be made to turn in either direction.

Note that if both direction inputs are the same state (either true or false) and the circuit is enabled, both terminals will be brought to the same voltage (Power+ or Power−, respectively). This operation will actively brake the motor, due to a property of motors known as *back emf*, in which a motor that is turning generates a voltage counter to its rotation. When both terminals of the motor are brought to the same electrical potential, the back emf causes resistance to the motor's rotation.

### B.3.3 The SGS-Thomson Motor Driver Chip

A company named SGS-Thomson makes a series of chip called the L293 that incorporates two H-bridge motor-driving circuits into a single 16-pin DIP package.

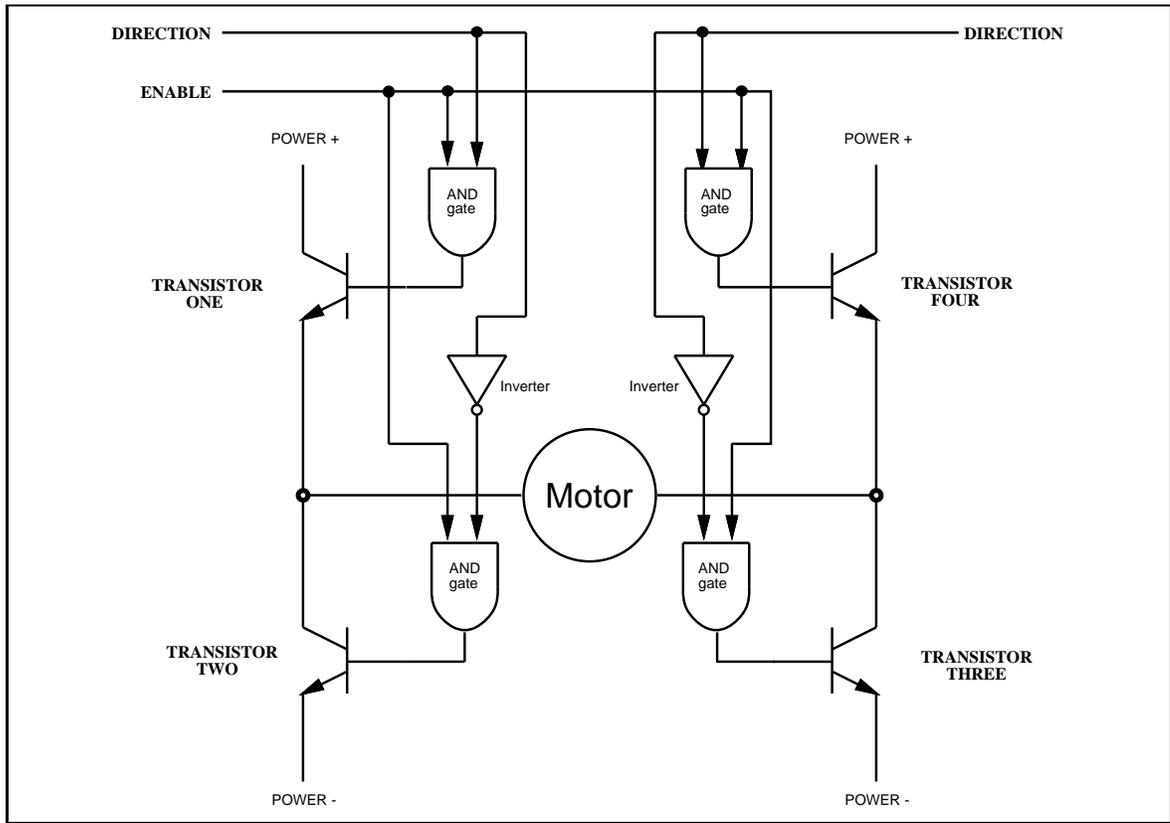


Figure B.12: The H-Bridge with Enable Circuitry

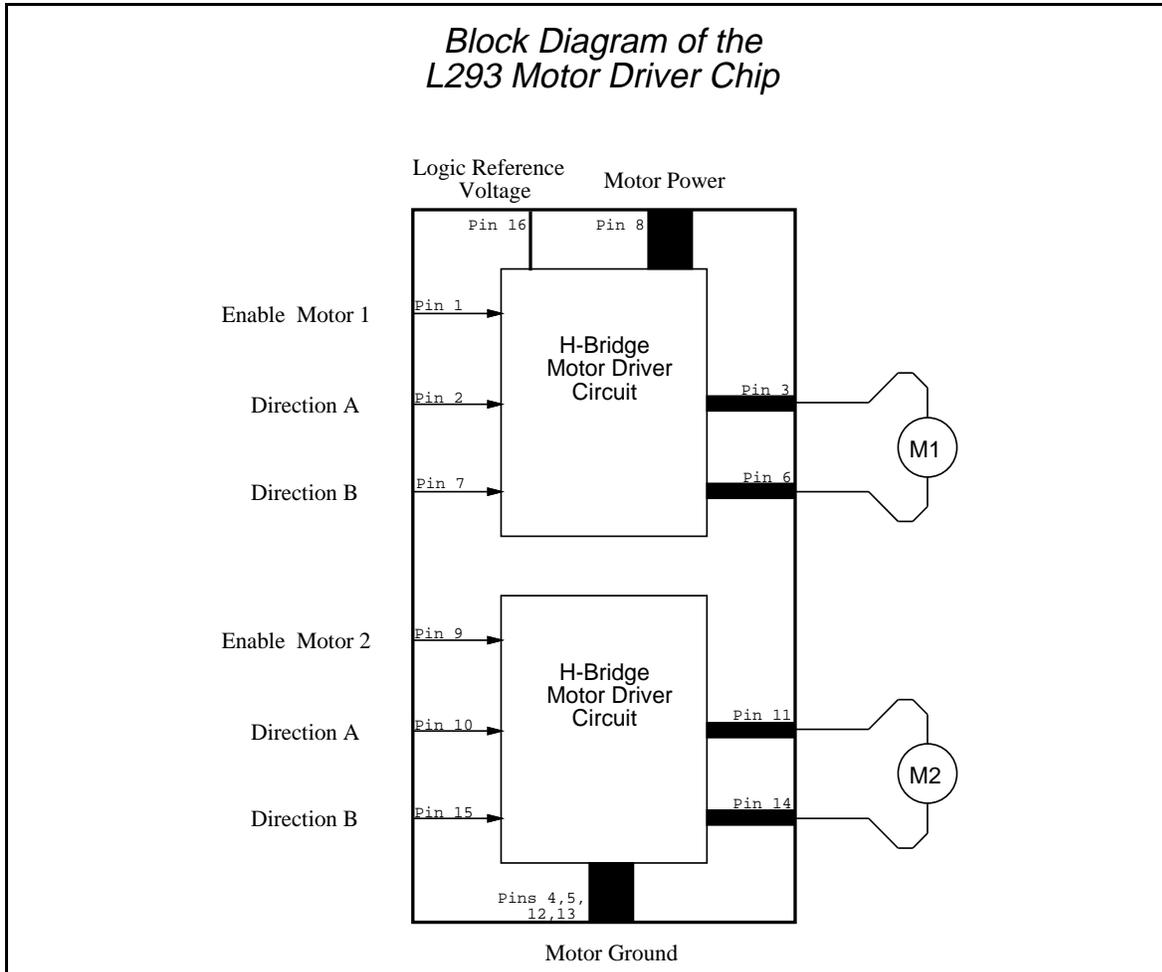


Figure B.13: The SGS-Thomson L293 Motor Driver IC

Figure B.13 shows a block diagram of this incredibly useful integrated circuit.

The schematic of the motor circuit (Figure B.14) shows how the L293 chips are used in the 6.270 board design. Eight bits are used to control four motors. Four of the bits determine the direction of the motors (with the assistance of inverters) and four bits determine when the motors are on or off.

Notice that braking a motor is not possible with this circuit configuration, because the inverters do not allow both direction inputs of a given motor to be the same state.

The speed of a motor may be controlled by pulsing its enable bit on and off. This technique, called *pulse width modulation*, is explained in the chapter on motors.

### B.3.4 Power Considerations

#### Current Handling and Spike Protection

In the 6.270 circuit design, two L293 chips are used in parallel to control each motor. This is an unconventional circuit hack add to the current-handling capacity of the motor drivers.

Two different L293 chips are used in this circuit. One chip, the L293D, has internal spike-protecting diodes on the motor outputs. These diodes protect the motor chip and the rest of the circuit from electrical noise generated by the motors. The other chip, the L293B, does not have these diodes, but has a greater current handling ability than the 'D chip.

The L293D can supply 600 mA of current per channel; the L293B, 1000 mA. Used in parallel, the circuit can supply 1600 mA per channel. Because of the spike-killing diodes contained in the 'D chip, the overall circuit is safe to use.

#### Power Supply Isolation

The electrical noise generated by motor can be hazardous to a microprocessor circuit even with the use of the diodes. For this reason, separate power supplies are used for the motors and the rest of the microprocessor electronics.

Figure B.15 shows the power-supply circuitry. Notice that *Logic Power*, for the microprocessor circuitry, is a configuration of four AA cells, while + *Motor*, power for the motors, is supplied through the J1 connector.

The motor ground and the logic ground must be kept at the same potential so that the control signals from the '374 chip shown in Figure B.14 can communicate with the L293 chips. These grounds are kept at the same potential by the inductor L1.

The inductor is used to provide reactance (frequency-dependent resistance) to trap spikes that might travel from the motors, through the L293 chips, and into the microprocessor circuit.

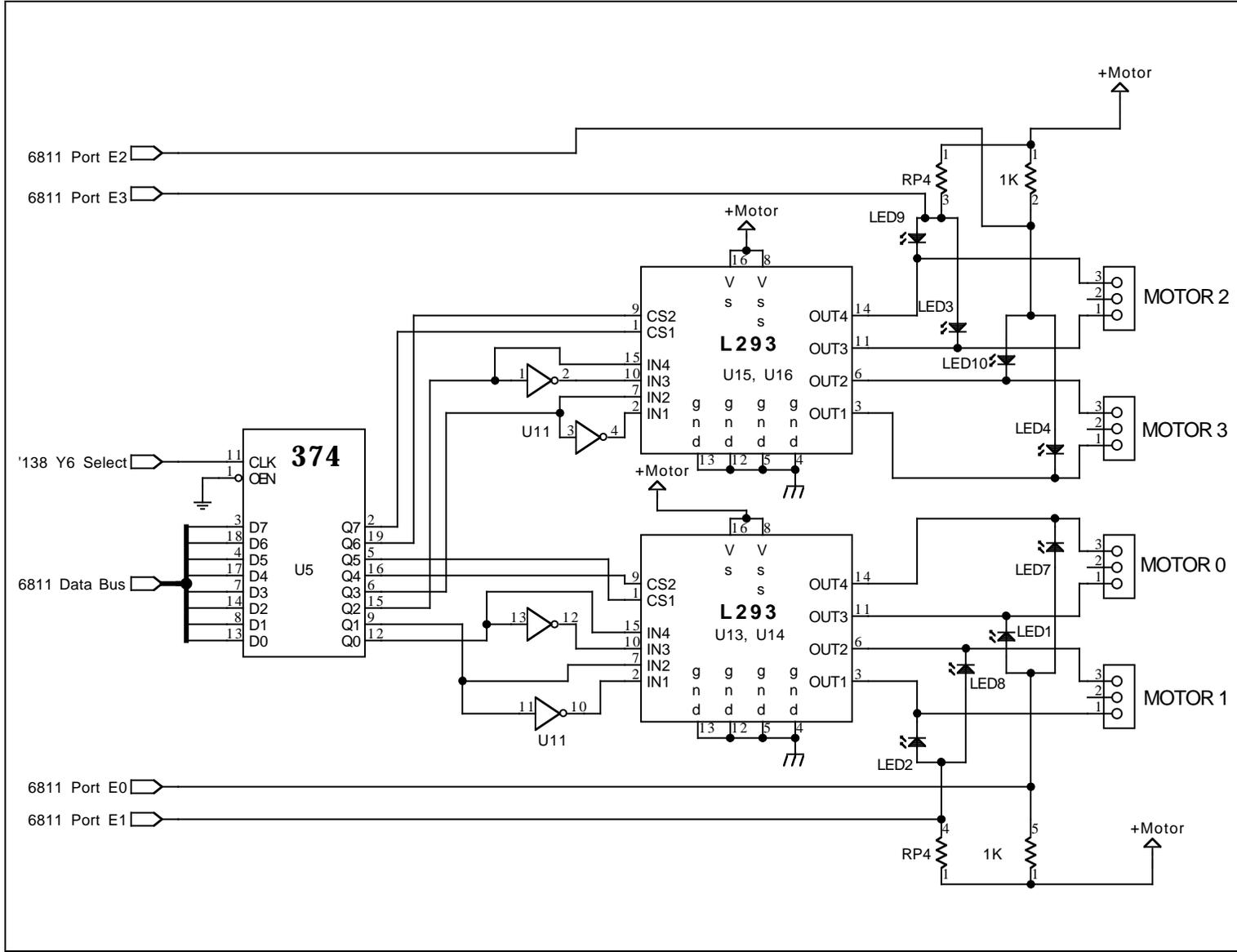


Figure B.14: Motor Driver Circuit

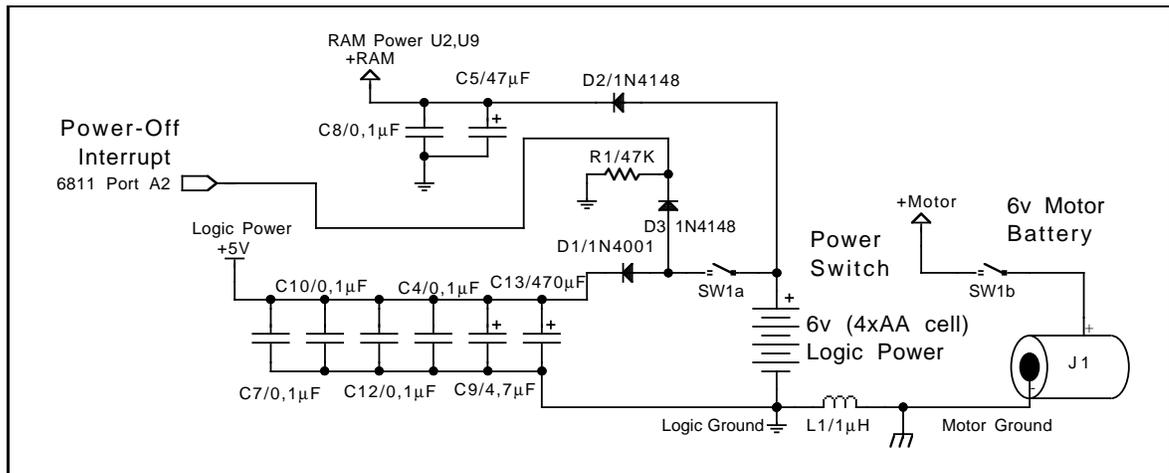


Figure B.15: Power Filtering and Switching Circuit

### B.3.5 Expansion Board Motor and LED Circuitry

The 6.270 Expansion Board plugs into the Expansion Bus header depicted in Figure B.9. This header connects to the 6811 data bus and to the six '138 select signals that are not used on the main board.

Figure B.16 illustrates how a single L293D chip is used on the Expansion Board to provide outputs for two additional motors. Because six outputs of the '374 chip are wired to control all four direction inputs and the two enable inputs of the L293D, the motors can be braked if desired. Or, four unidirectional devices may be powered.

The remaining two bits of the '374 are connected to transistor drivers. These transistor circuits are well-suited for powering light-load devices, such as LEDs.

## B.4 Analog Inputs

The 6811 has on-chip circuitry to perform an analog-to-digital signal conversion. In this operation, a voltage from 0 to 5 volts is linearly converted into an 8-bit number (a range of 0 to 255). This feature is one of the many that make the 6811 very well suited for control applications.

The 6811 has eight of these analog inputs. In the 6.270 board design, four of these pins are wired to a *motor current monitoring circuit*, and four of them are wired to input connectors.

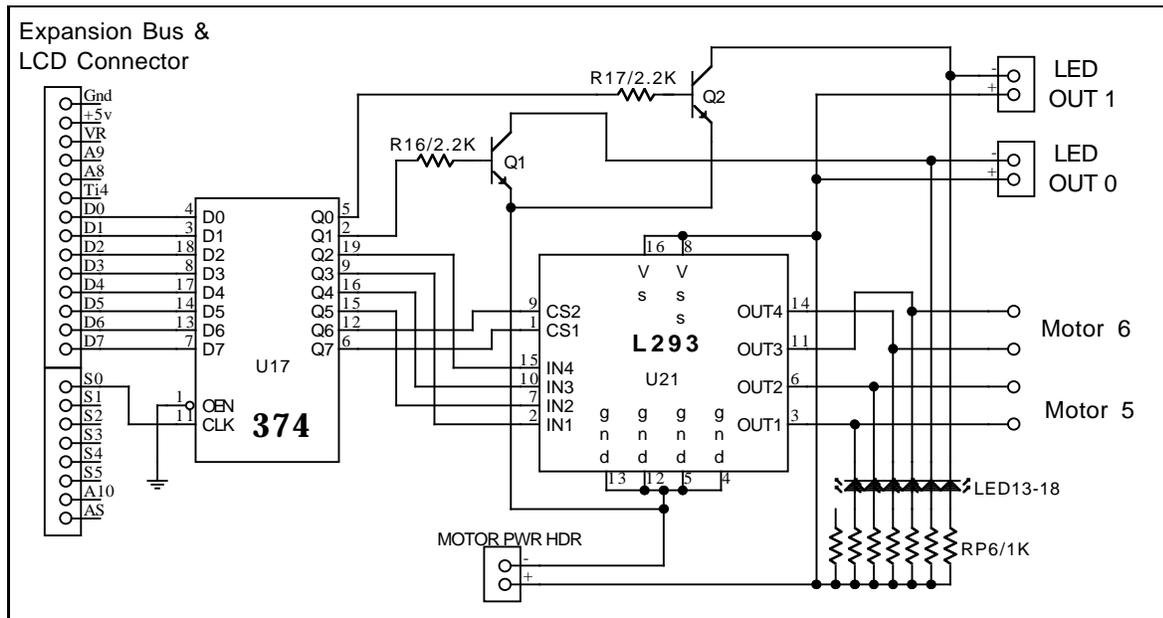


Figure B.16: Expansion Board Motor and LED Circuitry

### B.4.1 Motor Current Monitoring Circuit

When the L293 chips drive a motor, there is a voltage drop across the transistors that form the H-bridge. The transistor connected to motor ground (0 volt potential) might drive the motor at some voltage between .2 and .8 volts; the transistor connected to the positive terminal of the battery (say it's at 6 volts) might drive the motor between 5.2 and 5.8 volts.

The amount of this voltage drop is proportional to the amount of current being supplied by the motor-driving transistor. When more current is being supplied, the transistor drops more voltage.

This undesirable property of the L293 transistors is exploited to give a crude measurement of the amount of current being driven through the motor. A fundamental property of motors is that as the amount of work they are performing increases, the amount of current they drawn also increases. So the current measurement yields data on how hard the motor is working—if it is turning freely, if it is stalled, or if it is working somewhere in between.

As indicated in Figure B.14, the voltage feedback point is tapped from the indicator LEDs that are connected to the motor outputs. The voltage across the LEDs will decrease as a result of increased current draw of the motor (and the corresponding decreased performance of the L293's). This voltage is fed to a 6811 analog input and can be measured by the 6811 analog-to-digital conversion hardware.

Each of the four motor circuits is wired in this way to a 6811 analog input.

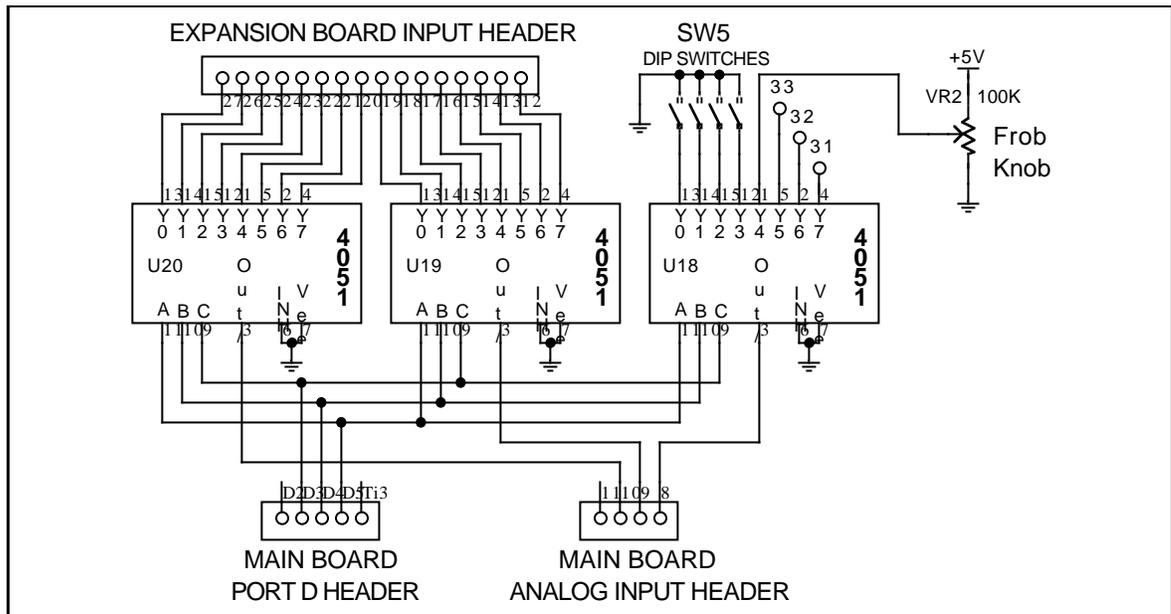


Figure B.17: Expansion Board Analog Input Circuitry

### B.4.2 Analog Input Multiplexing on the Expansion Board

The Expansion Board has three *eight-to-one analog multiplexer* ICs. These chips (the 74HC4051) have eight inputs and one output; depending on the state of three selector inputs, one of the eight input lines is connected to the output.<sup>2</sup>

The outputs of the '4051 chips are wired into the 6811 analog inputs when the 6.270 Expansion Board plugs into the main board. Three signals from the 6811 are used to control the multiplexers and select which analog input is mapped to the 6811 analog input<sup>3</sup>.

Figure B.17 is a schematic of the analog input circuitry on the 6.270 Expansion Board. It is easy to see how the use of the analog multiplexer chips greatly expands the analog input capability of the 6.270 hardware:

- Two of the '4051 chips have their inputs wired to a bank of sixteen open sensor inputs.
- The other chip is wired from the *Frob Knob*, a general-purpose analog input knob, and four DIP switches (for user configuration input).

<sup>2</sup>Actually, the chip's signals are bidirectional, but for the purpose of this discussion, it is convenient to think of the chip as having eight inputs and one output.

<sup>3</sup>These signals are taken from the 6811's *High Speed Serial Port*, a special sub-system of the 6811 that allows it to communicate at high speeds with other 6811's. In the 6.270 application, this functionality is not needed; instead, the signals are used as simple digital outputs.

- Three of the inputs to this third chip are open, as is one of the 6811's analog inputs.

## B.5 The Serial Line Circuit

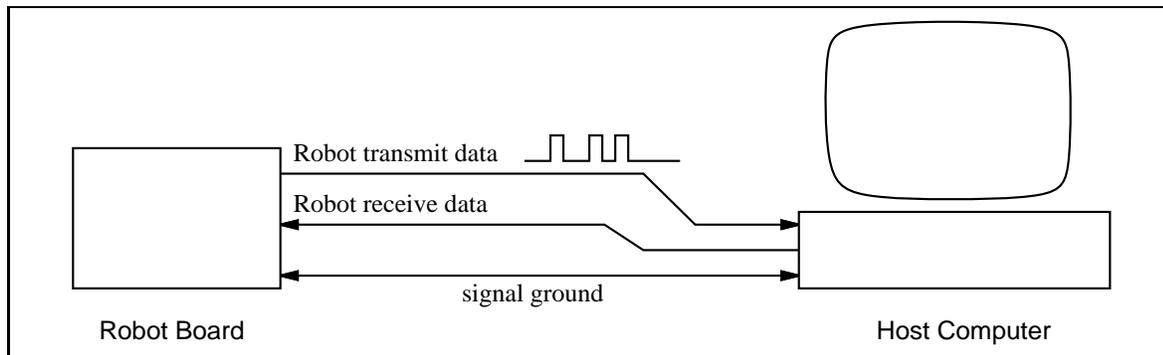


Figure B.18: Host and Board Communications over 3-Wire Serial Link

The 6.270 Board communicates with a host computer over an RS-232 serial line. “RS-232” refers to a standard protocol for communications over a three-wire system, as depicted in Figure B.18. Nearly all of today’s computers have serial ports that conform to the RS-232 standard.<sup>4</sup>

In the RS-232 system, a “logic zero” is indicated by a +15 volt signal with respect to ground, and a “logic one” is indicated by a –15 volt signal. Note that this is different from standard digital logic levels in several ways. Negative voltages are used, higher voltages are used, and negative voltages connote a logic one value.

The 6811 chip includes circuitry to generate waveforms compatible with the RS-232 systems, but requires external circuitry to convert its own signals, which obey the digital logic norms, to RS-232 signals as described.

There exist off-the-shelf single-chip solutions to this problem (most notably, the MAX232 and MAX233 chips made by Maxim, Inc.), but these chips are typically expensive and consume a fair bit of power. The solution implemented on the 6.270 board requires a few more components, but is significantly cheaper and less power-hungry.

### B.5.1 Serial Output

One of the difficulties in generating RS-232 signals is obtaining the negative voltage required to transmit a logic one. However, it turns out that the specified –15 volts

<sup>4</sup>The actual RS-232 standard involves quite a few more wires for conveying various status information, but the data itself is transmitted on two uni-directional wires.

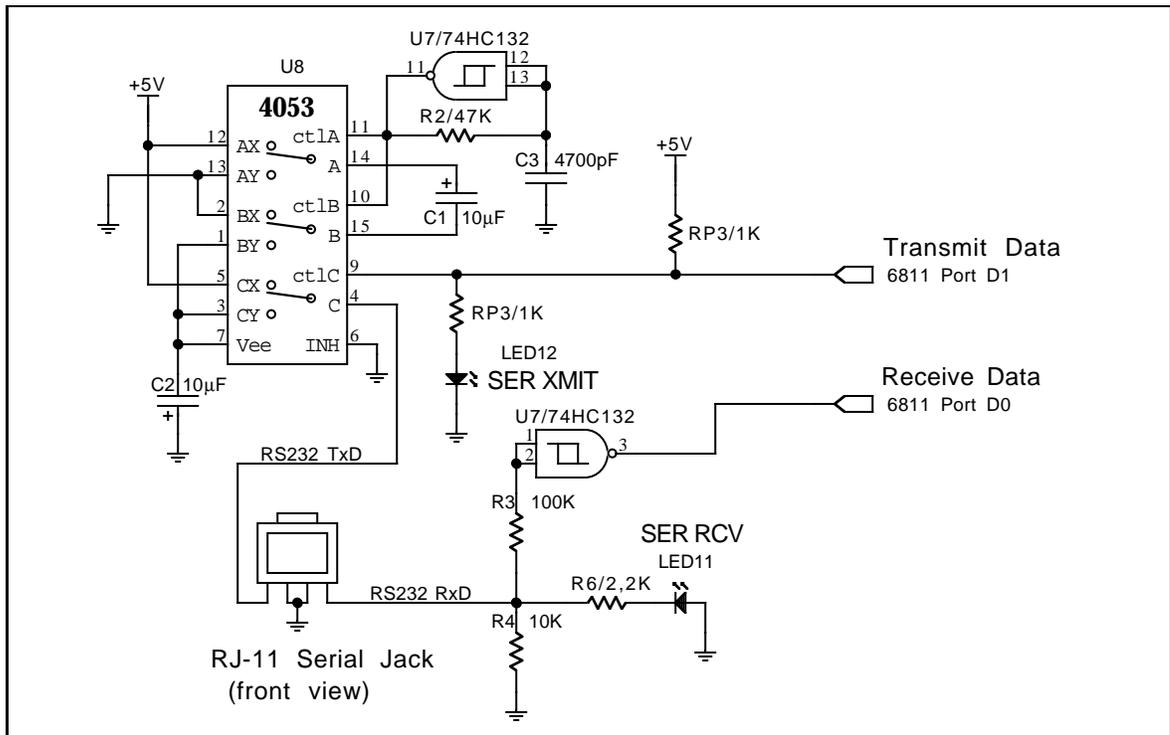


Figure B.19: Serial Line Circuit

is not required:  $-5$  volts will do for most applications.

A circuit called a *charge pump* is used to generate this negative voltage. A charge pump consists of two capacitors and a switch. One of the capacitors is charged to a positive voltage by the main power supply. Then the terminals of this capacitor are switched to the terminals of the second capacitor. The first capacitor discharges rapidly into the second, charging it negatively with respect to system ground. This process is switched rapidly, and a steady negative voltage supply is produced in the second capacitor.

The schematic for this circuit and the rest of the serial line circuitry is shown in Figure B.19. The heart of the circuit is a 74HC4053 chip, which is a triple analog SPDT switch that can be controlled digitally.

The charge pump is built from switches A and B of the '4053 chip. Capacitor C1 is charged from system voltage when the switches are in the X position (as is illustrated in the diagram). When the switches are flipped to the Y position, C1 discharges into capacitor C2, creating a negative voltage on C2 with respect to system ground.

The C switch is used to switch either the  $-5$  volts from C2 or  $+5$  volts from system power out over the serial line. This is done by wiring the 6811's logic-level "Transmit Data" signal to the control input of switch C.

Switches A and B are repeatedly alternated between the X and Y positions by an oscillator built from a schmitt-trigger NAND gate wired as an inverter (U7) and an RC delay (R2 and C3). This oscillator is tuned to about 10,000 Hertz, a frequency that has been experimentally determined to yield good results.

The commercially-available single-chip solutions mentioned earlier implement a similar circuit. In fact, they use two charge pumps. The first is used to double the system voltage of  $+5$  volts to obtain a  $+10$  volt supply that more closely matches the RS-232 standard. The second charge pump inverts this  $+10$  volts to obtain a  $-10$  volt supply.

## B.5.2 Serial Input

A schmitt-trigger NAND gate is wired as an inverter to convert the negative-true RS-232 standard to the positive-true logic level serial standard. Resistor R3 limits the current that can flow into the gate when the serial line voltage is negative, preventing the possibility of damage from a high negative voltage.

The RS-232 standard dictates that a serial line should be in the logic true (negative voltage) state when it is not transmitting data. LED11, the serial receive indicator, is wired such that it will light in this state, being powered directly by the serial voltage generated by the host computer. This LED serves as an indicator that the 6.270 board is properly hooked up to the host.

## B.6 Battery-Backing the Static RAM

The static RAM used in the 6.270 board is a special low power device, a relatively recent innovation in widely-available memory technology. This memory chip requires only an infinitesimal amount of current to store its contents when it is not being used.

The actual amount of current—less than one *microampere*—is so small that a standard alkaline battery does not notice it. That is, the battery will last as long as its shelf life, whether or not it is supplying one microamp to a circuit. (Alkaline batteries have a shelf life of several years.)

Having a battery-backed static memory greatly increases the usability of the 6.270 board. A robot can simply be turned on and operated immediately, without having to be connected to a computer first.

Unfortunately, implementing a battery-backed RAM can be complicated. The difficulty arises from unpredictabilities in microprocessor behavior when system power is either switched on or off. During these transition periods, the microprocessor is powered by illegal voltages, and its behavior is not defined. In order to make sure that the microprocessor does not corrupt the contents of the memory, orderly transitions from the powered-off to power-on states, and vice-versa, must be implemented.

### B.6.1 Powering the Memory Chip

Figure B.15 illustrates how power is always provided to the memory (through diode D2) even when microprocessor and motor power is turned off. Capacitors C5 and C8 help to smooth the power supply of the memory, and also can provide power to the memory while batteries are being changed. Because the current draw is so small, capacitor C5 will actually keep the memory “alive” for periods of up to thirty minutes when the system is powered off and batteries are removed.

### B.6.2 The Power-Off Interrupt

Diodes D1, D2, and D3 provide isolation amongst the three parts of the circuit:

- the memory’s power supply
- the microprocessor’s power supply
- the power-off interrupt circuit

This isolation is necessary to ensure clean transition of the power-off interrupt circuit when power is shut off. The power-smoothing capacitors (both for the memory and for the microprocessor circuit) retain charge for a brief period after power is

switched off. The diodes prevent this charge from “flowing backward,” and allowing one part of the circuit to power another.

When power is switched off, the power-off interrupt signal *immediately* goes low. However, system capacitors (mostly, C13) will keep the microprocessor powered up for a short while (about one-tenth of a second).

The interrupt signal generates a hardware-level interrupt to the 6811. A special-purpose software driver is activated, which has the job of shutting down the 6811 in an orderly fashion before the capacitor power supply runs down.

Sometimes, a brief physical jolt to the microprocessor board will dislodge a battery momentarily, causing the interrupt to be triggered. It would be incorrect for the software to shut down the system in this case. So, the interrupt software waits for a short while to see if the interrupt line goes high (indicating that power has returned). If power does return, the interrupt exits without taking action.

If power does not return after about one-hundredth of a second, the software routine executes a machine-language HALT instruction, which shuts off the microprocessor. This sequence of actions implements an orderly shutdown sequence.

### B.6.3 The Power-Up Delays

To make sure that the 6811 does not corrupt memory while it is being powered on, the 6811 is made to enter a particular mode that does not access external memory. Also, the 6811 is hardware-protected from writing into the RAM immediately after power is turned on.

After power has normalized, the user can press the system reset button, causing the microprocessor to reset into the “run” state.

This solution is implemented with resistor-capacitor delay circuits connected to a mode input pin of the 6811 and the logic gate that enables the RAM. These circuits are shown in Figure B.9.

The C11, R9, and R10 circuit is connected to the mode input pin MODA of the 6811. During a power-on condition, the 6811 will come out of its reset state before capacitor C11 is fully charged; thus, it will register MODA as being a logic zero. This will put the 6811 into a bootstrap download mode in which a program is executed from internal ROM rather than external RAM. (This mode is used to download the operating system software to the microprocessor.)

If the user presses reset after power has been turned on for a little while, C11 will have been charged. In this case the 6811 will latch MODA as a logic one, and will go into the normal run mode, executing a program in external memory.

In order to ensure that the 6811 does not access external memory except when the user presses reset, the C6, R7, and R8 circuit keeps the RAM disabled for a short period after power is turned on. This circuit acts as extra insurance for proper system power-on.

This collection of circuits, modes, and interrupts used to protect the RAM during power-up and power-down is a fairly arcane solution to a common problem. Most circuit designers opt for single-chip power management circuits that do all of this work in one place. Dallas Semiconductor is a leading manufacturer of such chips.

Perhaps the only advantage of the solution implemented in the 6.270 board is lower cost: the few diodes, resistors, capacitors, and a software driver are cheaper than a power-management IC.

## B.7 The Infrared Transmission Circuit

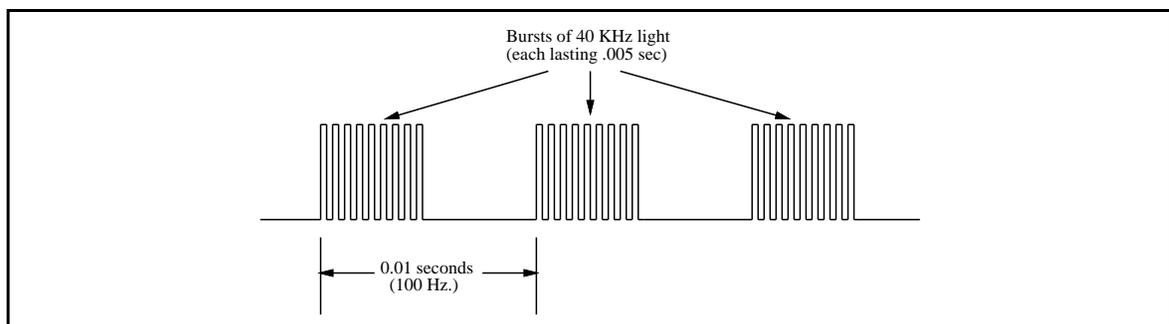


Figure B.20: Square Wave Consisting of Bursts of 40 KHz Signals

The Sharp GP1U52 sensor, and others like it commonly used in TVs, VCRs, and other devices controlled by infrared, is sensitive to *modulated* infrared light. It detects the presence of infrared light that is blinking on and off at a particular rate. The GP1U52 sensor is tuned to 40,000 Hertz (40 KHz).

In TV remote applications, a data stream is then generated around the 40 KHz carrier frequency. The signal consists of bursts and gaps of the 40 KHz transmissions.

For the 6.270 application, the 40 KHz carrier is used to transmit a square wave of relatively low frequency (100 or 125 Hz), as shown in Figure B.20. When the Sharp IR sensor decodes this signal, it removes the 40 KHz carrier, yielding a copy of the square wave that was originally transmitted (Figure B.21).

The 6.270 system software continuously checks the Sharp sensors for square waves of the specified frequency. The software actually locks on to the square wave when it is present and counts the number of consecutive cycles that have been detected.

A special circuit is used to generate infrared emissions modulated at the 40 KHz frequency. A block diagram of this circuit is shown in Figure B.22.

The diagram shows that the '390 chip, wired in a divide-by-fifty configuration, is used to generate a 40 KHz signal from the 6811 E clock, a 2 Mhz signal. In actuality, the '390 chip contains two *decade counters*. Each these consists of a separate divide-by-five counter and a flip-flop (a divide-by-two device). The '390 is wired in the

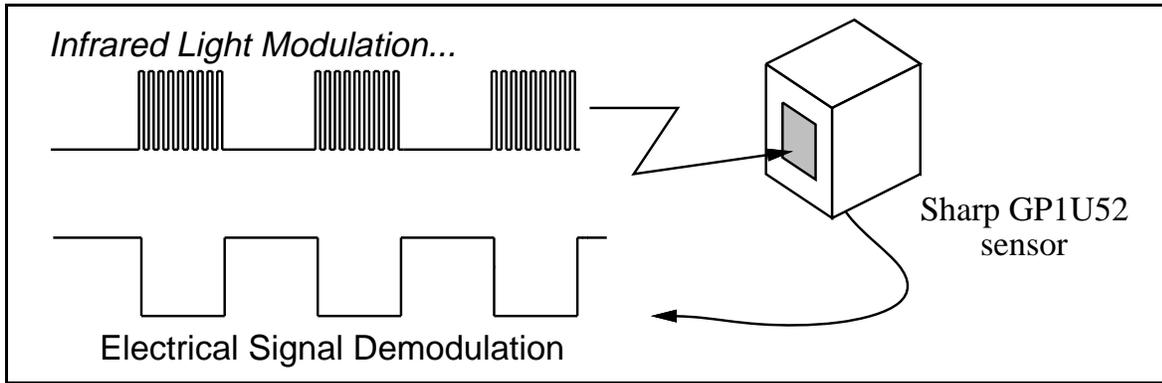


Figure B.21: Sharp IR Sensor Decoding IR-Encoded Square Wave

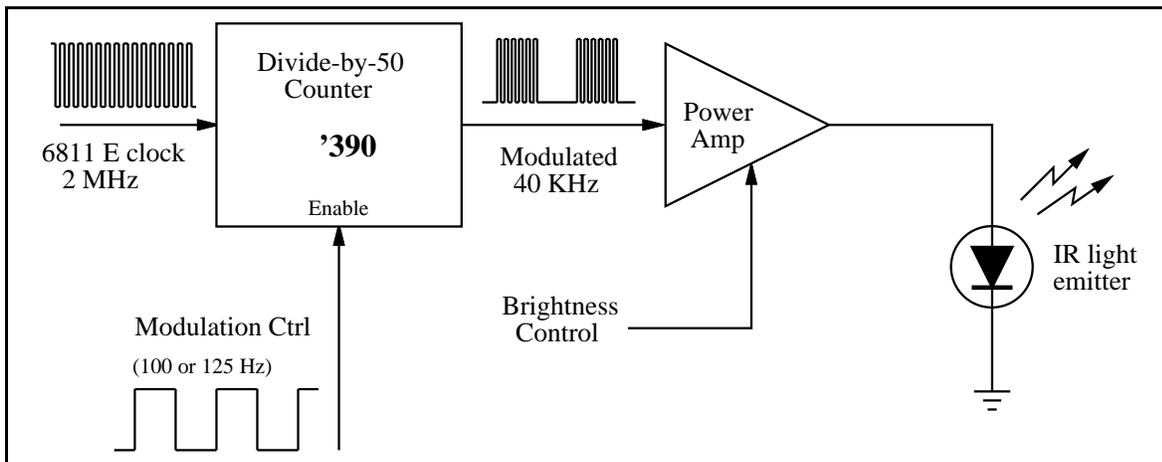


Figure B.22: Block Diagram of Infrared Circuitry

divide-by-fifty function by ganging two of the divide-by-five counters and one of the flip-flops.

The *IR control* signal is wired to the clear input of the '390 chip; when this signal is true, the counters will reset and will be prevented from counting. By modulating this signal, the 6811 can generate the low-frequency square wave that ends up being transmitted to the Sharp sensor.

The *IR brightness* signal can be used to adjust the gain of the output amplified, thereby changing the amount of light being transmitted by the IR LEDs. For 6.270 purposes, the amplifier is kept at full brightness, but there may be applications where this feature is useful.

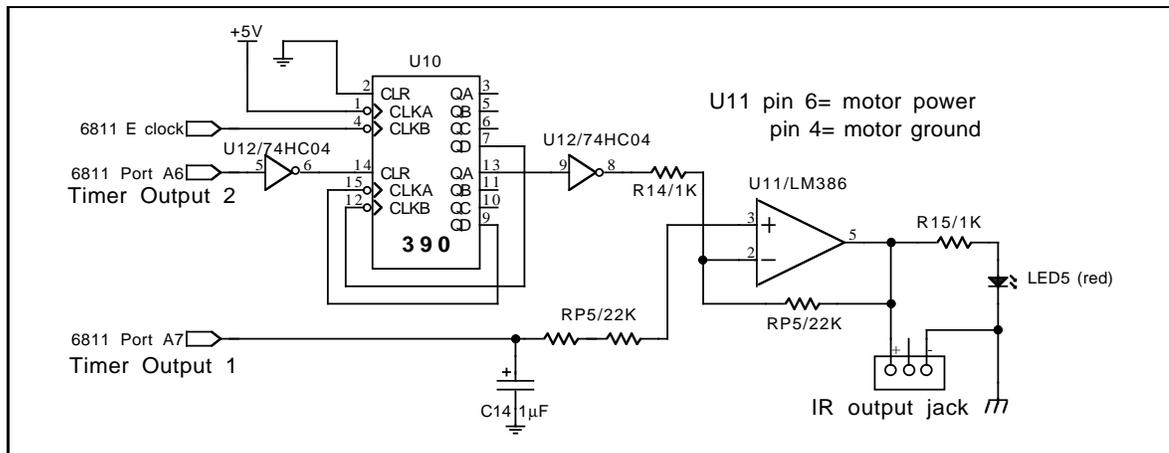


Figure B.23: Infrared Transmission Circuit

Figure B.23 shows the full circuit schematic for the IR subsystem.

The LM386 chip, a power op-amp, is used to drive the infrared LED's. The output of the '390 chip is inverted and presented to the  $-$  input of the op-amp. When this signal is high, the op-amp will have a large negative differential input, and its output will be forced to the negative extreme (0 volts), turning off the IR LEDs.

When the  $-$  input is low, the op-amp will operate in a linear gain mode (with a gain of 22, as determined by R14 and one resistor of RP5). The voltage present on the op-amp's  $+$  input determines the strength of the output. This voltage is controllable from the 6811 pin (Port A7), which can charge C14 to whatever voltage is desirable. Thus, the gain of the op-amp, and thereby the brightness of the IR LEDs, is software-controllable.

### B.7.1 The IR Beacon

Figure B.24 shows the schematic for the IR beacon. Each infrared LED has a visible LED in series with it so it should be easy to ascertain that the device is transmitting

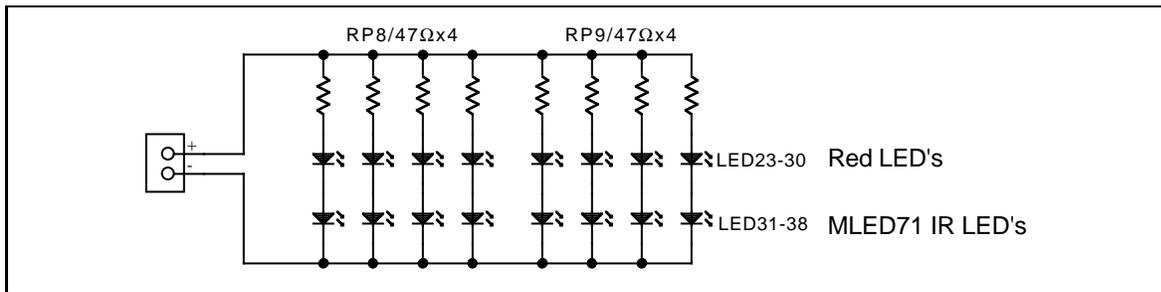


Figure B.24: Infrared Beacon Circuit

infrared light properly. The resistors act as current-limiters, limiting the amount of current that can travel through any branch of the circuit to between 10 to 20 mA.

## B.8 The LCD Display

The first fourteen pins of the 6.270 Board's Expansion Bus are designed to be compatible with a 14-pin standard LCD bus. A variety of character-based LCD devices with different screen sizes use this standard bus.

The LCD bus standard is fairly simple, consisting of the following signals:

- an 8-bit data bidirectional bus
- two mode select input signals
- a clock line
- a voltage reference for contrast adjustment
- +5 volt logic power
- signal ground

In fact, reading and writing data to an LCD is much like reading and writing data to latches or to memory. There is one problem, however: LCDs only work at data transfer rates up to 1 MHz. The 6811 in the 6.270 board operates at 2 MHz—too fast for most LCDs.

One straight-forward solution to the speed problem would be to use a '374-type latch between the 6811 and the LCD. The '374 could be written to at the full bus rate of the 6811; its outputs would drive the data bus of the LCD. A separate signal could be used to toggle the LCD's clock line, causing it to latch the data that had been written to the '374<sup>5</sup>.

<sup>5</sup>This solution assumes that one does not need to read status data back from the LCD.

An unconventional, zero-additional-hardware solution has been implemented in the 6.270 system, which takes advantage of an obscure feature of the 6811 microprocessor.

The 6811 has two main operating modes, known as *single chip mode* and *expanded multiplexed mode*. The discussion of memory read and write cycles that has been presented in this chapter has been based on the expanded multiplexed mode, which is the 6811 mode that is used when external memory is part of the 6811 circuit.

When the 6811 is operated in single-chip mode, the upper-eight-bit address bus and multiplexed address/data bus become general purpose inputs and outputs of the 6811, controllable by system software. Thus, in single-chip mode, the 6811 could communicate with the LCD with a software driver, rather than the too-fast hardware communication.

There is a problem with this, however: when the 6811 is placed into single-chip mode, it can no longer execute a program from its external RAM. In fact, as far as the 6811 is concerned, there *is no* external memory anymore.

Fortunately, the 6811 has 256 bytes of internal RAM, from which it can execute a program when in single-chip mode. Thus, a software driver could execute out of internal RAM, perform a transaction with the LCD, and then switch back to expanded-multiplexed mode and return control to the main program in external memory.

The obscure feature mentioned is not the fact that the 6811 has both of these modes, but the idea of dynamically switching between them. Here is the solution that has been implemented:

1. Start by copying a software driver from external system memory into the 256 bytes of internal 6811 memory.
2. Begin execution of the driver program located in internal memory:
  - Place the 6811 into single-chip mode; external memory disappears.
  - Execute a low-speed transaction with the LCD by directly controlling the data bus via software.
  - Place the 6811 into expanded-multiplexed mode.
  - Return to the main program in external memory.
3. Continue normal program execution.

The actual LCD driver routine buffers characters to be printed to the LCD; one thousand times per second, an interrupt routine calls the internal memory driver as described, writing a single character to the LCD. The whole process operates transparently to the 6.270 system user.

## B.9 The Low-Battery Indicator

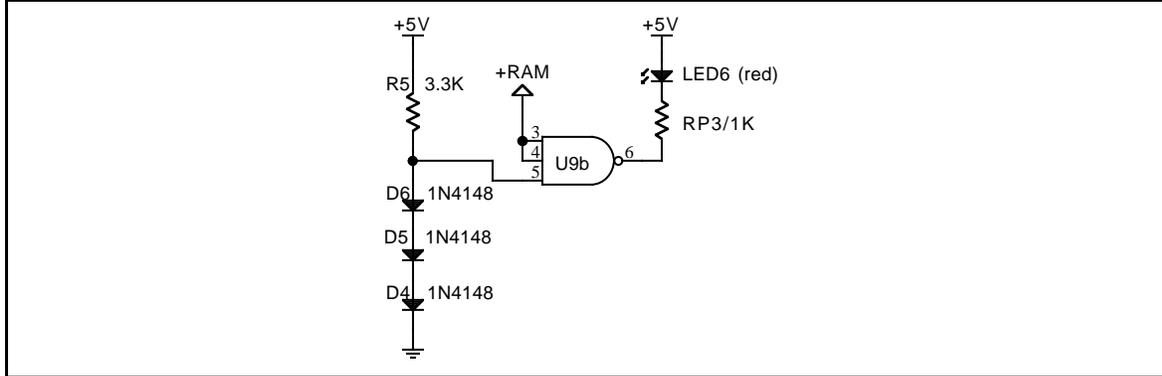


Figure B.25: Low Battery Indicator Circuit

A spare gate on U9 has been used to implement a low-battery indicator. The schematic is shown in Figure B.25.

The transition point for determining if a digital input is logic one or logic zero is normally one-half of the supply voltage. Assuming a 5 volt supply, signals greater than 2.5 volts will be interpreted as logic ones, and signals less than 2.5 volts will be interpreted as logic zeros.

Diodes have the interesting property that they drop exactly 0.6 volts when current travels through them. Thus the input voltage to the gate U9? will be about 1.8 volts, over a wide range of system supply voltages.

Assuming a 5 volt supply, this input would be interpreted as logic zero. U9? is wired as an inverter, so it will output a logic one. Since the LED is wired from supply voltage, it will be off in this state.

Suppose supply voltage falls to 3.5 volts. Now the transition point is around 1.75 volts. The input to the gate is 1.8 volts, so it becomes a logic one. U9? inverts this to obtain a logic zero, and drives zero volts on its output, lighting the LED.

The actual transition point in the circuit is closer to 4 volts, because the diodes tend to drop a bit more than 0.6 volts that are usually specified. Surprisingly, nearly all of the 6.270 electronics, including the 6811 microprocessor, work fine at voltages as low as 4 volts. One notable exception is the Sharp GP1U52 sensor: its performance decreases sharply at supply voltages less than 4.5 volts.

# Appendix C

## Printed Circuit Layouts

This appendix section has the printed circuit board artwork patterns for the 1992 6.270 boards:

- the Microprocessor Board
- the Expansion Board
- the Battery Charger Board
- the Motor Switch Board
- the Infrared Beacon Board

The board artworks are provided to facilitate debugging; they are not intended to serve as master artworks for fabricating new printed circuit boards. The layouts are reproduced at actual size given the limits of Laserwriter reproduction technology.

## C.1 Microprocessor Board

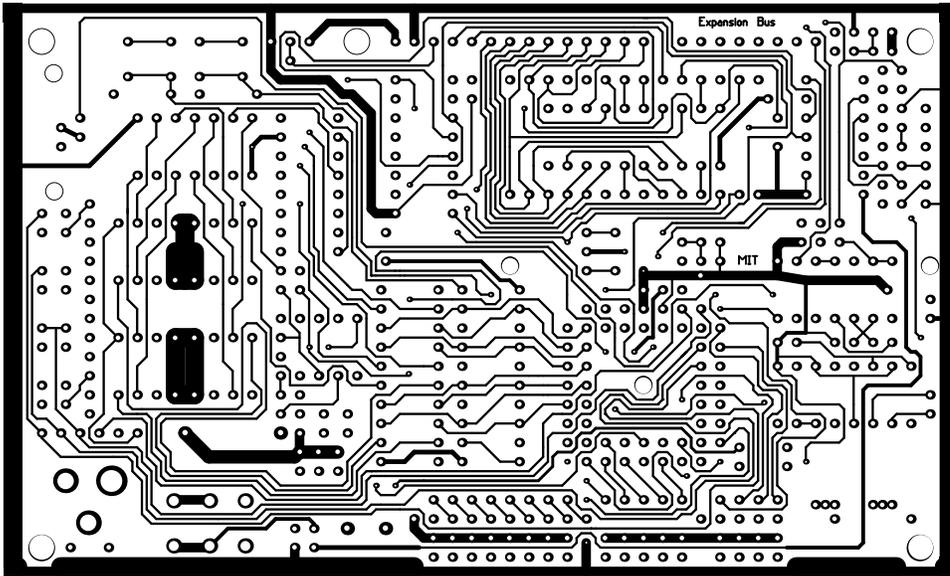


Figure C.1: Microprocessor Board, Component Side

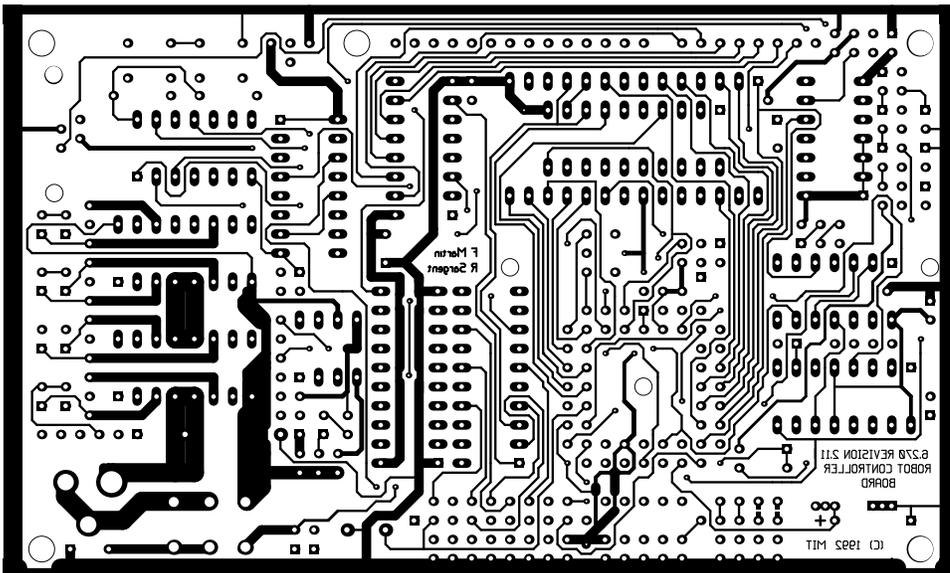


Figure C.2: Microprocessor Board, Solder Side

## C.2 Expansion Board

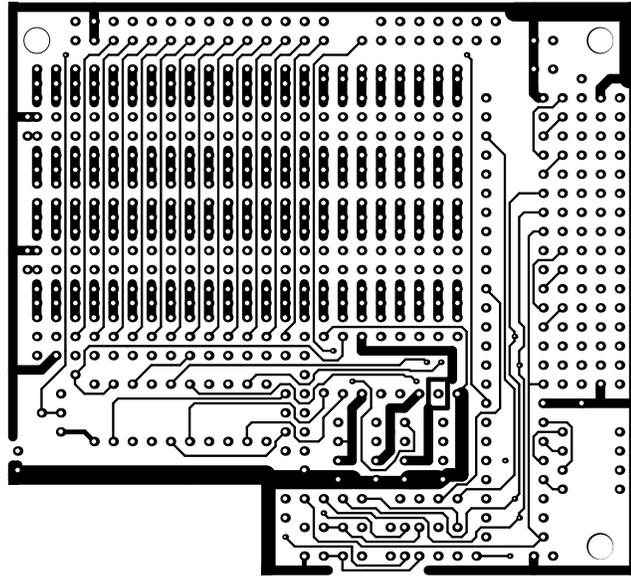


Figure C.3: Expansion Board, Component Side

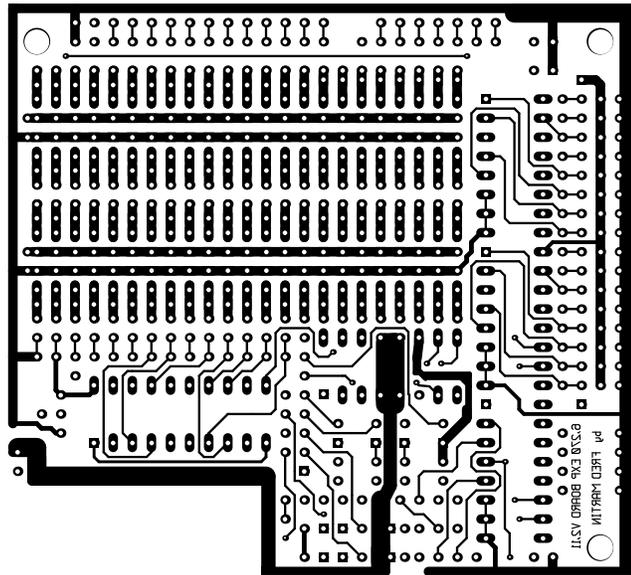


Figure C.4: Expansion Board, Solder Side

### C.3 Battery Charger Board

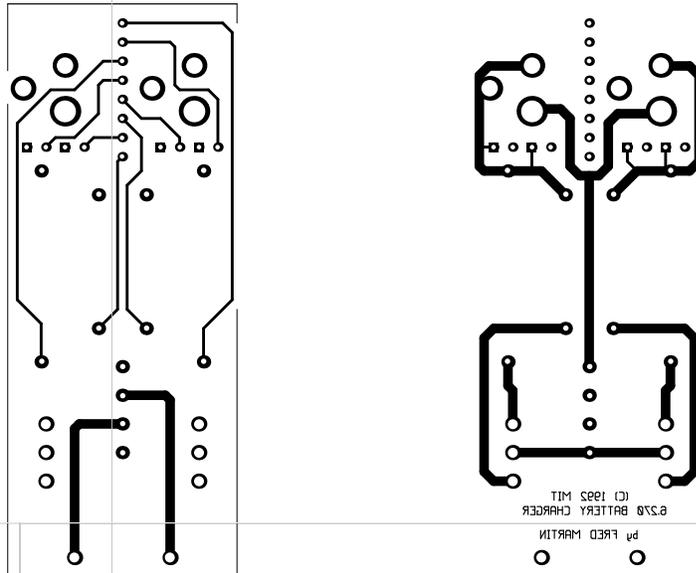


Figure C.5: Battery Charger Board, Component and Solder Sides

### C.4 Motor Switch Board

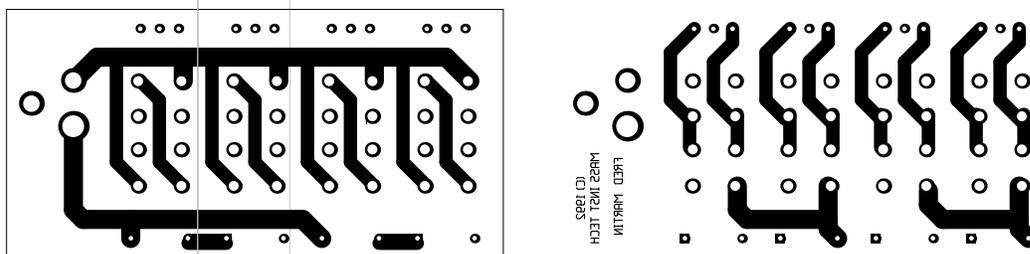


Figure C.6: Motor Switch Board, Component and Solder Sides

## C.5 Infrared Beacon Board

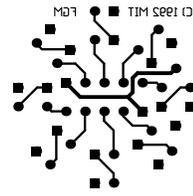
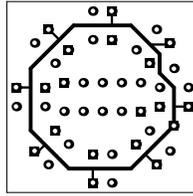


Figure C.7: Infrared Beacon Board, Component and Solder Sides



# Appendix D

## Electronic Suppliers

Knowing how to find and purchase electronic parts is a big part of being a resourceful robot builder. This appendix section will help you get started on the right track.

### D.1 Introduction

When ordering electronic parts, availability is a key issue. Many times, a firm will list an item in its catalog, but be “out of stock” when you call them to make an order. *Make sure that you ask the sales person to “check stock” when you order anything.* If an item is not in stock—meaning it’s not in the company’s own warehouses—the company must order it from “the factory”. The factory is a huge distribution center over which your electronics dealer has little control. So, it can take anywhere from a couple of weeks to *several months* for the parts you want to come to your dealer.

Some companies will try to deceive you about how long it will take to receive parts from the factory. In general, the rule is: if the company doesn’t have it in stock, order it from somewhere else.

The company listing that follows is divided into two categories: retail companies and surplus companies.

#### D.1.1 Retail Suppliers

Retail companies sell brand-new parts and generally keep a reliable, stable inventory of the products that they sell. Most retail suppliers care a lot about their customers and try to keep them by giving good service: by keeping most parts in stock in their own warehouses. By my definition, “good service” means that the company (1) delivers product quickly, and (2) does not try to deceive the customer about the availability of an item.

In exchange for the reliability of the retailer's line, you pay a price. In general, new retail parts cost anywhere from two to ten times the cost of the same part purchased from a surplus house. But many times, particularly when designing products for manufacture, you need the dependability of parts that the retailer offers. Also, many important parts can't be found in surplus.

### D.1.2 Surplus Suppliers

Surplus companies buy bulk lots of discontinued, closed-out, old, used, or otherwise "surplus" merchandise. Many a fascinating, useful, and most importantly, *cheap!* part can be found in the surplus market.

The catalog of a given surplus company changes from issue to issue, as the company sells out of some items and has made purchases of another items. Many surplus companies try to keep some of their product line stable, but the most exciting and cheap products are usually from a one-time bulk purchase. Sometimes, you see the same item hitting the catalogs of several different surplus companies at the same time—a good indication that some manufacturer has just cleared their warehouse of that item!

Because of this flux in product line, it's very important to check that a surplus dealer has a particular item in stock when you order. Many surplus dealers will tell you the approximate quantity of an item that they have in stock—useful if you're going to base a design on the availability of a particular part.

You should be extremely wary if a surplus dealer tries to have an item in your order put on backorder (meaning that they don't have it in stock). Usually, they're just *hoping* to make another bulk purchase of the item you want. Unless they're particularly convincing that the item will indeed be available in a short period of time, cancel the order and find what you need somewhere else.

This said, surplus companies are a great boon to electronics hobbyists of all types. They have fascinating stuff at super-low prices. Generally surplus stock is at least half the cost of retail, but for many items, you will pay as low as *one-tenth* of the retail cost. For funding-strapped educators and hobbyists, the work involved in tracking down parts in surplus is well worth it, and can make a huge difference in the final cost of a project or class. Also in surplus, you can find items that are just too obscure, or must be ordered in huge quantities if you tried the retail market.

### D.1.3 Ordering Catalogs

Most companies, both retail and surplus, are glad to send you their catalog.

In the listings, if you see the phrase "orders only" listed in front of an 800 number, then you should not call that company on their 800 number unless you are placing an

order. If the phrase “orders and info” is listed, then the company has not requested the 800 number to be used for orders only.

Don't be bashful: go ahead and order catalogs from all of the companies listed here. You will find overlapping stock in both retail and surplus, but you'll be surprised by the variations in the prices of the retailers, and practically every surplus catalog has at least one exciting item you'll find nowhere else. So, happy hunting in the electronics market!

## D.2 The Surplus Companies

In the descriptions that follow, the phrase “basic parts” refers to an assortment of common digital, linear, and microprocessor ICs, resistors, capacitors, and LEDs. Some surplus dealers keep a regular stock of these items and others do not.

In most cases, these basic parts are better purchased in retail. With these items, prices in surplus are often no better than retail. Also, surplus ICs are often “pullouts,” meaning they've been removed from junk equipment. Chips are tested before being sold to you, but a new part beats a used one at the same price.

There are some exceptions when surplus is still better on ICs and other basic parts items. Occasionally, surplus can still undercut retail price by a significant margin, particularly on products such as optoelectronics, or RAM chips, for example.

Also, there may arise situations when *all* retail dealers may be out of stock of a particular IC, due to fluctuations in the factory supply. In this situation, go straight to your basic parts surplus dealer—they will have chips that have been lying around for years. Sure, they're not brand new, but most old chips work just as well as new ones.

### D.2.1 Surplus Company Listing

#### All Electronics Corp

**P.O. Box 567 • Van Nuys, CA 91408**

**orders only: (800) 826-5432**

**information: (818) 904-0524**

**fax: (818) 781-2653**

Good basic parts assortment. Many interesting surplus boards and assemblies, excellent motor selection, rechargeable batteries, switches, optoelectronics, etc. Prices very good.

Terms: \$10 minimum order; MC/VISA/Discover accepted; flat shipping/handling charge of \$3.50.

**American Design Components****815 Fairview Avenue • POB 220 • Fairview, NJ 07022****orders and info: (800) 776-3700****local: (201) 941-5000****fax: (201) 941-7480**

Surplus computer equipment, power supplies, lots of motors and batteries, switches, MOVIT robot kits. Overpriced basic parts assortment. Some other stuff very well priced.

Terms: \$15 minimum order; MC/VISA/AMEX/ accepted; annoying shipping/handling charges of \$3.00 plus 10% of cost of merchandise. Mentally add 10% to their catalog prices as you're going through their catalog and you won't be as annoyed by this excess "handling" charge.

**American Science & Surplus****601 Linden Place • Evanston, Illinois 60202****orders and info: (708) 475-8440****fax: (708) 864-1589**

Formerly known as **JerryCo Inc.**, this company produces the most entertaining, humorous, and informative catalog in the surplus industry. A diverse assortment of stuff: tools, toys, stationery, electronics, motors, batteries, gears and bearings, etc., etc. You never knew you needed it until you saw it in this catalog. The catalog is fun to read even if you don't buy anything. Prices are generally good but not great.

Terms: \$12.50 minimum order (including handling charge); MC/VISA accepted; flat shipping/handling charge of \$4 for most orders.

**Edlie Electronics****2700 Hempstead Turnpike • Levittown, NY 11756-1443****orders and info: (800) 645-4722****local: (516) 735-3330****fax: (516) 731-5125**

Strange and interesting surplus assortment. Tools, cables, test equipment, electronic kits, car audio, PCB materials. Catalog is disorganized, but some stuff is *very* cheap. Minimal basic parts.

Terms: minimum cash order \$25.00; minimum credit card order \$50.00; MC/VISA accepted; shipping/handling charges starting at \$4.75 and increasing with size of order. Service is sometimes slow; inquire about availability and delivery when placing order.

**The Electronic Goldmine**

**POB 5408 • Scottsdale, AZ 85261**

**orders and info: (602) 451-7454**

**fax: (602) 451-9495**

Excellent catalog comprised mostly of packaged electronic kits, with a small amount of very good surplus stock mixed in.

Terms: \$10 minimum order; MC/VISA accepted; minimum \$3.50 for UPS shipping.

**Electronic Supermarket**

**P.O. Box 988 • Lynnfield, MA 01940**

**orders: (508) 532-2323**

Odd surplus assortment composed mostly of close out junk. Retail serious car audio stuff.

**Herbach and Rademan Company**

**18 Canal Street • P.O. Box 122 • Bristol, PA 19007-0122**

**orders only: (800) 848-8001**

**info: (215) 788-5583**

**fax: (215) 788-9577**

Mixture of surplus and retail with an emphasis on industrial equipment: large motors, pumps, tools, batteries, powers supplies, optics. Retail closed-circuit TV/security equipment.

Terms: \$5 handling charge for orders under \$25; MC/VISA accepted; shipping charges are UPS rates.

**Hosfelt Electronics, Inc.**

**2700 Sunset Boulevard • Steubenville, OH 43952-1158**

**orders and info: (800) 524-6464**

**local: (614) 264-6464**

**fax: (614) 264-5414**

Wide surplus selection. Big relay assortment, transistors, inductors, motors, fans, enclosures, xformers, opto's, caps, pots, speakers, connectors, electronic kits, books, tools. Good random stuff too. Prices range from so-so to excellent depending on the item.

Terms: no minimum order for credit card, check or COD payment; MC/VISA/-AMEX/Discover accepted; shipping charges are UPS rates + \$1.00.

**Jameco Electronics**

**1355 Shoreway Road • Belmont, CA 94002**  
**orders: (415) 592-8097**

Used to be a surplus dealer; now deals PC kits, power supplies, cases, motherboards. Listed because remaining surplus stock and chip assortment is useful. Get their catalog in any event for the comprehensive semiconductor manufacturer directory.

Terms: \$50 minimum order to discourage hobbyists. The blatant commercialism of this company is upsetting, especially considering their past as a valued member of the surplus industry.

**Kelvin Electronics**

**7 Fairchild Avenue • Plainview, NY 11803**  
**orders and info: (800) 645-9212**  
**local: (516) 349-7620**  
**fax: (516) 349-7830**

Huge selection of inexpensive and interesting educational/science kits—robotics, radio, audio, etc. Hobbyist electronic kit assortment. Excellent inexpensive tool and test equipment selection. Small surplus selection; excellent prices on LEDs. Very good basic parts.

Terms: \$20 minimum order; MC/VISA accepted; shipping/handling charges of 5% of total order or \$5, whichever is greater, with exceptions for heavy items, etc.

**Marlin P. Jones & Associates**

**P.O. Box 12685 • Lake Park, FL 33403-0685**  
**orders and info: (407) 848-8236**  
**fax: (407) 844-8764**

Good robotic surplus: motors, transformers, lots of relays, switches, optical stuff, power supplies, batteries. No chips. Good capacitor selection.

Terms: \$1 handling charge for orders under \$10; MC/VISA accepted; shipping charges are UPS rates.

**Mendelson Electronics**

**340 East First Street**  
**Dayton, OH 45402-1257**  
**orders and info: (800) 422-3525**  
**local: (513) 461-3525**  
**fax: (513) 461-3391**

Huge family owned surplus supplier. Would rather sell you 100 or 1000 of an item than 10 of them. Lowest prices to be found on many computer items, motors, switches, connectors, displays, fans, transformers, capacitors. Catalog often lists manufacturers' part numbers only: it's helpful if you have a data book from the manufacturer.

Terms: \$50 minimum order; MC/VISA/Discover accepted; shipping charges are UPS rates.

**R&D Electronics**

**5363 Broadway • Cleveland, OH 44127**

**orders only: (800) 642-1123**

**info: (216) 441-5577**

**fax: (216) 621-8628**

A small but very useful surplus assortment, good service, very good prices. Lots of switches, motors, transformers, LCD displays, wire, good other random stuff.

Terms: \$10 minimum order; MC/VISA/Discover accepted; shipping charges are UPS rates.

**Surplus Traders**

**P.O. Box 276 • Winters Lane**

**Alburg, Vermont 05440**

**voice: (514) 739-9328**

**fax: (514) 345-8303**

Small but extremely cheap surplus assortment; minimum purchases per item of 50 to 100 quantities. Lots of AC/DC adapters, some switches, telephone equipment, CATV stuff.

**Unicorn Electronics**

**10010 Canoga Ave. • Chatsworth, CA 91311**

**orders and info: (800) 824-3432**

**local (818) 341-8833**

**fax: (818) 998-7975**

Extensive basic parts assortment. Very good prices on chips; price break at 25 pcs. on most parts. Laser diodes.

Terms: \$15 minimum order; MC/VISA accepted; no shipping charges on pre-paid orders.

## D.3 The Retail Companies

### Active Electronics

*several regional centers across the USA; call Telemarketing Department at (800) 677-8899*

Leading retail dealer, oriented toward both small orders and large. Very good assortment of all necessities. Prices are sometimes high and sometimes very good. 33% discount on most items if \$300 or more of any individual item is ordered.

### Arrow Electronics

**Catalog Division • 1860 Smithtown Avenue  
Ronkonkoma, NY 11779  
orders and info: (800) 93-ARROW**

Huge retail dealer. Prices are high unless large quantities are involved. Excellent selection.

### C-Gate International

**3529 Ryder Street • Santa Clara, CA 95051  
orders and info: (408) 730-0673  
fax: (408) 730-0735**

Computer accessories dealer. Unbelievably low prices on many items. How about a 6-outlet metal power strip for under \$5 in single quantities? Or null modem adapters for \$1.70? This is *the* ultra-cheap accessories warehouse. Look for their ad in *Computer Shopper* magazine.

Terms: no minimum order; MC/VISA accepted; you pay exact UPS shipping charge plus \$2.95 handling charge per order.

### Digikey Corporation

**701 Brooks Ave. South • P.O. Box 677  
Thief River Falls, MN 56701-0677  
orders and info: (800) DIGI-KEY  
fax: (218) 681-3380**

Large parts and accessories assortment. Chips, displays, power supplies, cases, connectors. Extensive. Prices okay, better in quantity.

Terms: \$5 handling charge for orders under \$25; MC/VISA, check, money order or COD; customer pays shipping on credit card and PO orders. Increasing volume discounts for orders over \$100.

**Electronic Buyers' Club****1803 Northwest Lincoln Way****Toledo, OR 97391-1014****orders and info: (800) 325-0101**

Organization with a \$35 yearly membership fee. Unusual, but once you join you get incredibly low prices on basic electronics: semiconductors, ICs, resistors, caps, diodes, ceramic resonators, LEDs, more. Prices start low at single quantities and go down if you buy tens or hundreds. Beats any of the other retailers hands down for most of the parts that they stock.

Catalog includes many manufacturers' data sheets and is a reference work in and of itself. Quick delivery on stocked parts. If you will spend \$100 or more on electronic supplies in the next year, the \$35 fee will earn its price. Many parts sold are available at literally one-fourth the prices that Digikey (for example) advertises.

See also *International Components Corporation*, the twin of EBC with slightly higher prices but no membership fee.

Terms: No minimum order but \$2 fee on orders less than \$25; shipping charges are exact UPS rates.

**International Components Corporation****1803 Northwest Lincoln Way****Toledo, OR 97391-1014****orders and info: (800) 325-0101**

The non-membership version of *Electronic Buyers' Club*, above. This catalog offers slightly higher prices and smaller selection than members have; still, the prices are extremely good.

Terms: No minimum order but \$2 fee on orders less than \$25; shipping charges are exact UPS rates.

**LEGO Dacta****555 Taylor Road • Enfield, CT 06082****orders and info: (800) 527-8339 or (800) 243-4870**

LEGO Dacta is the educational branch of the LEGO company (which has its U.S. headquarters in Enfield, CT). Dacta sells the *LEGO Technics* product line—the geared and motorized version of the LEGO system.

Call Dacta and get their “Gear Up for Learning” catalog, which has many LEGO Technics kits. Recommended starting kits are the *Technic Universal Buggy* (item number 1038, about \$57), which is a small, turtle-like vehicle with

dual motor drive, and the *Technic Supplemental Set* (item number 9605, about \$185), a general-purpose Technic kit with two motors, battery packs, and a large gear and beam assortment.

### **MCM Electronics**

**650 Congress Park Drive • Centerville, Ohio 45459-4072**

**orders and info: (800) 543-4330**

**fax: (513) 434-6959**

Tools, connectors, replacement VCR parts. Car audio and alarms. Phone and cable accessories. Large transistor assortment. Extremely good prices especially on tools: most stock is of acceptable Taiwanese manufacture. A must-have catalog. Quick and efficient delivery.

Terms: \$25 minimum for credit card orders; MC/VISA accepted; shipping charges are UPS rates plus \$2.10 handling charge.

### **Mouser Electronics**

**P.O. Box 699 • Mansfield, TX 76063**

**orders and info: (800) 34-MOUSER**

**fax: call for nearest number**

Wide selection of parts, connectors, and accessories. Excellent customer service will fax you detailed specs on any part they carry. Regional distribution centers across US. Good prices for a top-of-the-line retail dealer.

Terms: \$5 handling charge for orders less than \$20; MC/VISA/AMEX/Discover/-Diners Club/Carte Blanche accepted; shipping charges are actual UPS rates.

### **Ocean State Electronics**

**P.O. Box 1458 • Westerly, RI 02891**

**orders only: (800) 866-6626**

**fax: (401) 596-3590**

**info: (401) 596-3080**

Small retail dealer: chips, resistors, pots, semi's, LEDs, caps, relays, fans, enclosures, plugs. Generally overpriced across the board. Useful for extensive inductor assortment and amateur radio parts.

Terms: \$5.00 minimum order; MC/VISA, COD accepted; flat \$4 S/H most orders shipped UPS ground.

## Speaker Driver Hack

These optional assembly instructions explain how to use Motor Port #5 (on the Expansion Board) for the purpose of driving a small 8  $\Omega$  speaker. After performing the modification, the motor port will no longer be able to drive a motor.

The hack involves cutting a trace (that leads from the 'HC374 latch to a motor driver input) and jumpering in the control signal from the piezo beeper. A hole has been provided to inject the signal.

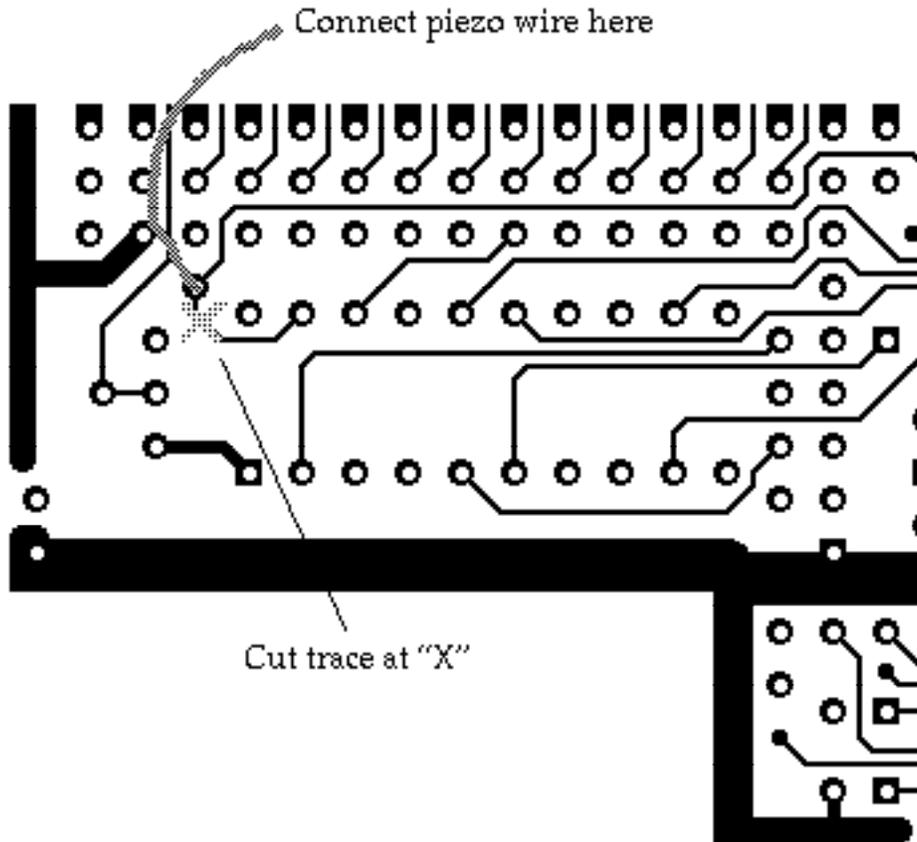


Figure 1: Component Side of Expansion Board

Figure 1 illustrates the hack. The piezo wire should be tapped from the positive (+) terminal of the piezo. There are several extra holes accessible on the solder side of the Microprocessor Board to make this connection.

Figure 2 illustrates the wiring of the speaker to a standard motor connector plug. *The 100 to 200 ohm potentiometer, used as a volume control, is required. Without the*

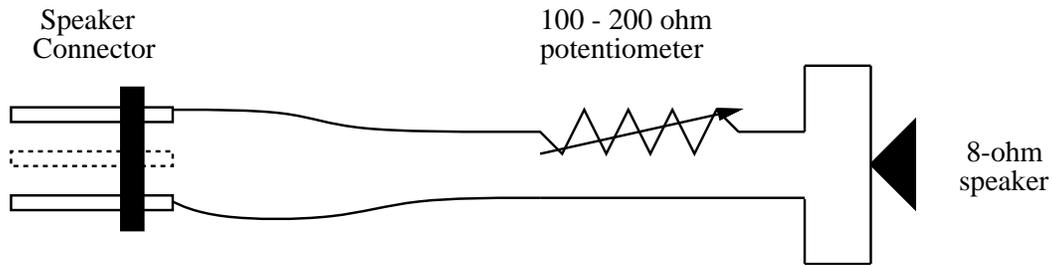


Figure 2: Speaker Plug and Wiring

variable resistance, the speaker will be extremely loud. Note that the potentiometers included in the kit—a 10 K $\Omega$  slide pot and a 125 K $\Omega$  rotary pot—are much too large in value to work effectively as a volume limiter.

After wiring the speaker, plug it into the Motor 5 port (the two connector pins closer to the motor chip).

From IC, execute the command `fd(5);`. This will turn on Motor Port 5, enabling the piezo signal to control the speaker. Execute any command that creates a tone (e.g., `beep();`), and the tone should be played through the speaker as well.

*Note:* The speaker circuit actually does not require the use of the right side of Motor Port 5. The speaker could be wired from the left side of Motor 5 to ground. However, the additional pin of each uni-directional driver provides a +6 volt signal, not ground. In configuration presented here, when the `fd(5)` command is issued, the right side of the motor 5 driver acts as speaker ground.