

# **ADSL PUBLICATIONS**

FOR EDUCATIONAL PURPOSES ONLY

**VOLUME 17  
NUMBER 17.2**

**LEGO ROBOTS: A NEW EDUCATIONAL TOOL FOR ECE 291**

**BY**

**RAJEEV GOEL**

**B.S., University of Illinois, 1993**

**THESIS**

**Submitted in partial fulfillment of the requirements  
for the degree of Master of Science in Electrical Engineering  
in the Graduate College of the  
University of Illinois at Urbana-Champaign, 1995**

**Urbana, Illinois**

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

THE GRADUATE COLLEGE

MAY 1995

WE HEREBY RECOMMEND THAT THE THESIS BY

RAJEEV GOEL

ENTITLED LEGO ROBOTS: A NEW EDUCATIONAL TOOL FOR ECE 291

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF MASTER OF SCIENCE

*W. K. ...*

Director of Thesis Research

*N. Narayana Rao*

Head of Department

Committee on Final Examination†

Chairperson

† Required for doctor's degree but not for master's.

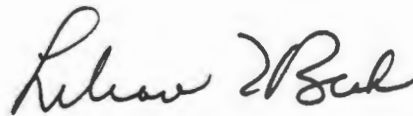
# UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

## GRADUATE COLLEGE DEPARTMENTAL FORMAT APPROVAL

THIS IS TO CERTIFY THAT THE FORMAT AND QUALITY OF PRESENTATION OF THE THESIS SUBMITTED BY RAJEEV GOEL AS ONE OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE ARE ACCEPTABLE TO THE DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING.

MAY 1, 1995

*Date of Approval*



Departmental Representative

## ACKNOWLEDGMENTS

I would like to thank Professor W. Kent Fuchs, my thesis advisor, for providing the inspiration for this project. He trusted my judgment and gave me the responsibility I needed to succeed. He was extremely supportive and enthusiastic at all times. I will always value his brilliant ideas and his constructive criticism.

I would like to thank Dr. Ricardo Uribe for all of the advice he has ever given me, both technical and otherwise. I will forever have a great deal of respect for the amount of practical experience he possesses. He has helped me in every way from debugging my circuits to dealing with people. I also thank Dr. Uribe for creating the Advanced Digital Systems Laboratory (ADSL), a noncompetitive environment in which motivated students can express their creativity and rise to their highest potential.

## TABLE OF CONTENTS

	Page
1. INTRODUCTION .....	1
1.1 MIT's 6.270 Course .....	2
1.2 ECE 291 at the University of Illinois .....	3
1.3 Overview of the Results .....	4
2. LEGO ROBOTS .....	6
2.1 "Woody" -- Infrared Controlled Robot Using the 6.270 Microcontroller Board .....	7
2.2 The Search for an 8088 Compatible Microcontroller .....	10
3. SPRING 1994 LEGOBOT CONTEST: CAPTURE THE TORCH .....	13
3.1 The Microcontroller and Programming Language .....	13
3.2 Contents of the LegoBot Kits .....	15
3.3 Contest Task and Rules .....	17
3.4 Contest Results .....	19
4. FALL 1994 LEGOBOT CONTEST: EIGHT BALL .....	25
4.1 The Vesta SBC88A Microcontroller and Additional Hardware .....	25
4.2 Contest Task and Rules .....	32
4.3 Contest Results .....	34
5. ENGINEERING OPEN HOUSE 1995 .....	38
5.1 "Cliff" -- Infrared Controlled Robot Which Received Commands from a PC .....	38
5.2 Results of Engineering Open House 1995 .....	43
6. SPRING 1995 LEGOBOT CONTEST: BASKETBALL .....	44
6.1 Contest Task .....	44
6.2 Improvements Over Previous Semester .....	45
7. CONCLUSIONS .....	47
7.1 Advantages and Limitations .....	47
7.2 Additional Resources .....	48
7.3 Future Considerations .....	49
APPENDIX A. SOURCE CODE FOR "WOODY" .....	52
A.1 Listing of WOODY.C .....	52
A.2 Listing of WOODY.ASM .....	56
APPENDIX B. SOURCE CODE FOR "NORM" .....	58

APPENDIX C. SOURCE CODE FOR "CLIFF" .....	62
C.1 Listing of CLIFF.C .....	62
C.2 Listing of CLIFF.ASM .....	67
C.3 Listing of CLIFF.PDS .....	69
APPENDIX D. SOURCE CODE FOR "SAM" .....	71
APPENDIX E. LEGOBOT CONTEST HANDOUT: CAPTURE THE TORCH .....	78
E.1 Introduction .....	78
E.2 The Task .....	79
E.3 The Playing Field .....	80
E.4 The Torch .....	82
E.5 Parts List .....	82
E.6 Rules and Restrictions .....	83
E.7 The Microcontroller and Software .....	84
E.8 Helpful Tips and Hints .....	85
E.9 Resources .....	85
APPENDIX F. LEGOBOT CONTEST HANDOUT: EIGHT BALL .....	87
F.1 Introduction .....	87
F.2 The Task .....	88
F.3 The Pool Table .....	90
F.4 Rules and Restrictions .....	91
F.5 Resources .....	92
F.6 Parts List .....	92
F.7 Sample Code .....	94
APPENDIX G. LEGOBOT CONTEST HANDOUT: BASKETBALL .....	98
G.1 Introduction .....	98
G.2 The Basketball Game .....	99
G.3 The Basketball Court .....	100
G.4 Additional Rules and Regulations .....	102
APPENDIX H. LEGOBOT HANDOUT: USING THE MICROCONTROLLER .....	103
H.1 Introduction .....	103
H.2 Connecting Devices to the Microcontroller .....	104
H.3 Reading the Value of a Digital Sensor .....	105
H.4 Reading the Value of an Analog Sensor .....	106
H.5 Writing to the Motor Ports .....	107
H.6 Downloading Your Program to the Microcontroller .....	108
H.7 Serial Communication .....	109
H.8 The Sensors .....	111
H.9 The Motors .....	114
H.10 Resources .....	114
LIST OF REFERENCES .....	116

## LIST OF FIGURES

2.1	Infrared Signal From a Remote Control Unit .....	9
3.1	Circuit Diagram for the Torch .....	18
3.2	Circuit Diagram for 100 Hz and 125 Hz IR Transmitters.....	20
4.1	Circuit Diagram for Additional Hardware on the Vesta Microcontroller.....	26
4.2	Pulse Width Modulation .....	29
4.3	Circuit Diagram for Pool Table IR Transmitters .....	34
4.4	Circuit Diagram for IR Test Module .....	34
5.1	Circuit Diagram for Cliff .....	39
5.2	Circuit Diagram for IR Transmitting Expansion Card .....	41
E.1	Top View of Playing Field for "Capture the Torch" .....	81
E.2	The Torch.....	82
F.1	Top View of Pool Table.....	90
G.1	Top and Side Views of Basketball Court.....	101
H.1	Microcontroller Port Locations.....	105



## 1. INTRODUCTION

*“If the colleges were better, if they really had it, you would need to get the police at the gates to keep order in the onrushing multitude. See in college how we thwart the natural love of learning by leaving the natural method of teaching what each wishes to learn, and insisting that you shall learn what you have no taste or capacity for. The college, which should be a place of delightful labor, is made odious and unhealthy, and the young men are tempted to frivolous amusements to rally their jaded spirits. I would have the studies elective. Scholarship is created not by compulsion, but by awakening a pure interest in knowledge. The wise instructor accomplishes this by opening to his pupils precisely the attractions the study has for himself. The marking is a system for schools, not for the college; for boys, not for men; and it is an ungracious work to put on a professor.”*

-- Ralph Waldo Emerson

Within the scope of engineering at the college level, most professors and students have witnessed at first-hand the phenomenon that a student's learning is more effective, complete, and permanent if the student is allowed to discover how the world works for himself or herself, as opposed to being told how it works. This is why universities are gradually increasing the number of laboratory courses available in their engineering curricula. Unfortunately, often the original intent of these laboratory courses is lost when the discoveries are made *for* the students instead of *by* the students. For example, the students are told in advance exactly what steps to perform and what results to expect from their experiments. Because of this, the learning is more passive than active. This is the problem that the “LEGO Robots” project attempts to solve.

The goal is to allow teams of students to design, construct, program, and debug their own mobile autonomous LEGO robots to accomplish a specific task. Through these means, the students shall discover for themselves which techniques work well and which techniques fail. They will learn to devise their own informal experiments to test their various control algorithms, and they will learn to evaluate the results of these experiments critically. They will experience the entire engineering design process involved in taking a product from its conception to its delivery. It is expected that the skills that they gain and the concepts that they learn will stay with them forever and help them in their future engineering careers.

### **1.1 MIT's 6.270 Course**

The inspiration for bringing LEGO robots to the University of Illinois (Urbana-Champaign) was a course at the Massachusetts Institute of Technology, which was originally started in 1987 by Michael B. Parker. The students taking this course, called 6.270, were originally involved in a programming competition in which they wrote programs to control computer-simulated robots. The course eventually evolved into a competition in which the students built their own mobile autonomous robots from LEGO parts. Every year, there was a different task, usually in the form of a game or modified sport, which the robots had to be designed to accomplish.

Fred Martin and Randy Sargent, graduate students at MIT, designed a custom printed circuit board, which served as a microcontroller for the LEGO robots. The board used a Motorola M68HC11 microprocessor and had convenient ports for four dc motors, a servo motor, and several different types of sensors. Eventually the board also supported a small LCD screen and gradually gained other features. Today, the students can program their robots in the C

programming language by using "Interactive-C" which was also developed at MIT specifically for use within the 6.270 course.

The MIT 6.270 course has been a great success every year since its beginning, and today it is one of the most well-known courses around the world.<sup>1</sup> Many universities have realized the great potential of a course where students have the opportunity to build LEGO robots and have started similar programs in their engineering departments.<sup>2</sup> The University of Illinois (Urbana-Champaign) is now among the growing number.

## 1.2 ECE 291 at the University of Illinois

In the Spring of 1994, the Department of Electrical and Computer Engineering at the University of Illinois had its first LEGO robot contest as part of a course, ECE 291, called "Computer Engineering II." This is an undergraduate level course in which the students study machine-level programming, use of computers for real-time data acquisition, control of input and output devices, and design and implementation of complex computer programs. These topics are covered thoroughly in both the lecture and laboratory sections of the course.

The laboratory portion is divided into six machine problems, or MPs. For each of the first five MPs, the students are given a detailed specification for a program which they must individually design and implement. The code is written in Intel 8088 Assembly language and C, and is run on Intel-based PCs. The ECE 291 laboratory is currently equipped with 15 to 20 Intel Pentium machines on which the students design, write, debug, and demonstrate their programs.

---

<sup>1</sup> More information about the M.I.T. 6.270 course can be found from the anonymous FTP site "cherupakha.media.mit.edu" or on the World Wide Web at URL "<http://www.mit.edu:8001/courses/6.270/home.html>".

<sup>2</sup> Information about the University of Maryland Robotics Competition can be found on the World Wide Web at the URL "<http://www.cs.umd.edu/projects/amrl/Robot-Comp/index.html>".

For the sixth MP, the students choose their own final project; they can pick from a list of ideas or they can use their creativity to invent their own. Depending on the professor and the semester, the students may have the option of working on the final project alone or they may work in groups of two, three, or four. MP6 usually lasts from three to four weeks. Beginning in the Spring 1994 semester, one of the options for MP6 was to design and implement a mobile autonomous LEGO robot to accomplish a given task. At the end of MP6, the LEGO robots from that semester would battle each other in a fun, noncompetitive, high-spirited, and exciting contest, called the ECE 291 LegoBot contest.

### **1.3 Overview of the Results**

At the time of this writing, ECE 291 students are participating in the LegoBot contest for the third consecutive semester. The LegoBot contests from each of the previous two semesters were very successful. The participants gave excellent feedback on the project, describing both the areas they especially liked and also the ways it could be improved. Almost all of them mentioned that they enjoyed building and programming their robots, and that they learned more by doing this project than they have learned all semester in other courses. While most aspects of the contests were highly successful, there were several areas which failed. As the LegoBot contest evolves from semester to semester, the glitches will be gradually ironed out until the project runs smoothly and flawlessly.

The remainder of this thesis is organized in a chronological fashion. Chapter 2 describes LEGO robots and the preliminary research and preparation that went into setting up a LEGO robot contest at this university. Chapter 3 describes in complete detail the Spring 1994 LegoBot contest including contest rules, robot designs, successes, and failures. Chapter 4 does the same

for the Fall 1994 LegoBot contest. Chapter 5 describes Engineering Open House 1995 and the robots which were presented there. Chapter 6 briefly describes this semester's (Spring 1995) contest which is currently in progress. Finally, Chapter 7 will summarize the overall results of the project and demonstrate the great potential for using LEGO robots as an educational tool. It will also provide suggestions for related future work.

Throughout this thesis, where appropriate, one will find discussions on the implementations of four different LEGO robots, named Woody, Norm, Cliff, and Sam. These robots were built only for experimentation and demonstration purposes; they did not participate in any of the contests. This thesis can be interpreted as a journal of what was accomplished each semester and how it was accomplished. It is hoped that this information will be useful to those who will be organizing the ECE 291 LegoBot contest in the future semesters. The appendices contain duplicates of documents that were distributed to the students in various semesters; by obtaining the electronic form of this thesis on diskette, one can easily reuse these handouts.

## 2. LEGO ROBOTS

A LEGO robot is a mobile machine constructed primarily from LEGO parts designed to accomplish a specific task. LEGO parts refer to the colored bricks, plates, gears, axles, wheels, pulleys, and other parts which are included in the LEGO Technic line of products.<sup>3</sup> These parts can lock together easily to form virtually any type of structure or machine and can be taken apart easily as well. The attraction for using LEGO to build small robots is that it is easy and does not require any tools except creativity of the mind. LEGO kits can be found in most toy and department stores. Depending on the kit, they come with complete instructions on how to build miniature devices and structures such as a house, a car, a motorcycle, a fire truck, a crane, an elevator, and a conveyor belt. The possibilities are literally endless.

The most interesting LEGO robots are those which are autonomous, meaning that they do not require any external control or intervention to perform the task. All of the control logic (i.e., microcontroller plus software) is mounted on the robot itself eliminating the need for wires stretching between the robot and a stationary computer, for example. Also there may be no form of wireless communication between the robot and any human or computer which is not part of the robot. All of the power sources (i.e., batteries) must be mounted on the robot as well, in order to qualify it as autonomous. The robots that are designed to participate in the ECE 291 LegoBot contest must be autonomous.

---

<sup>3</sup> LEGO Dacta, the educational branch of the LEGO company, manufactures the LEGO Technic line of products. For a catalog, contact LEGO Dacta / 555 Taylor Road / Enfield, CT 06082 / (800) 527-8339.

## 2.1 “Woody” – Infrared Controlled Robot Using the 6.270 Microcontroller Board

The purpose in designing and building this first robot, called Woody, was to gain some basic experience with LEGO robots. Woody was a nonautonomous LEGO robot which simply responded to commands sent to it via infrared signals. The IR signals were generated using a Sony television remote control device. The microcontroller board I used to control the robot was the same one used at MIT in their 6.270 course.

Professor Fuchs had ordered three unassembled 6.270 microcontroller boards from an individual on a robotics mailing list<sup>4</sup> who had volunteered to gather parts and sell them for a small profit in the form of a kit. Since the board came unassembled, the first step was to put it together. This involved learning how to solder, identify various components, and test for short circuits. Using the instructions in the 6.270 course notes [1], the assembly of the microcontroller board went quickly and smoothly.

For the motor power supply, three 2 V Gates rechargeable batteries, which came with the 6.270 microcontroller boards, were wired together. In addition, I made a custom serial cable to allow communication between the microcontroller and the desktop computer, which, in this case, was a Sun workstation. To test the board, I downloaded the Sun version of the Interactive-C software from MIT’s FTP site,<sup>5</sup> along with documentation and a test program written in C which tested all of the microcontroller’s features. The board worked perfectly the first time it was tested.

---

<sup>4</sup> This electronic mailing list focuses on discussions about small mobile robots and microcontrollers. The list name is “robot-board@oberon.com”. To subscribe, send mail to “listserv@oberon.com”.

<sup>5</sup> Interactive-C can be downloaded from the anonymous FTP site “cherupakha.media.mit.edu”.

The next step was to build the robot. Each of the kits that contained the unassembled 6.270 boards also included four dc motors and a servo motor. While a walking LEGO robot is indeed possible, it introduces many issues such as balance which greatly complicate the physical design. The simplest designs for mobile robots usually look much like tricycles or cars. Woody took the shape of a four-wheel wagon.

The servo motor, mounted in the front of the robot, controlled the rack-and-pinion steering mechanism. I attempted several different configurations for the drive wheels. A single motor driving both the left and right wheels did not work well because of the different speeds required when the robot had to turn. In retrospect, a differential gearbox run by a single dc motor would have been the ideal solution, because it would have allowed the two wheels to turn at different speeds when necessary. Eventually, however, separate motors were used to drive the left and right wheels of the vehicle. Thus, the software for the microcontroller had to adjust the speeds of the two motors individually (in addition to the angle of the servo motor) in order to make the vehicle go straight, turn right, or turn left. Using the 6.270 board, the directions of the dc motors could also be controlled through software, allowing the vehicle to go forward or backward.

The final stage was to write the software for the microcontroller to interpret the encoded IR signal and drive the motors appropriately. A portion of the IR signal coming from a standard remote control device is shown in Figure 2.1, with the 40 kHz carrier removed. Since the IR receiver modules<sup>6</sup> used have built-in circuitry to remove this carrier, this also is the exact waveform which is fed directly into the microcontroller.

---

<sup>6</sup> The IR receiver modules used were the Sharp GP1U52X.



The start code is followed normally by 32 bits of data, 8 of which are shown in Figure 2.1. Each bit is separated by a 0.7 ms high pulse. A "0" bit is represented by a low time of approximately 0.5 ms, and a "1" bit is represented by a low time of approximately 1.5 ms. Since there is a significant difference between the times for a "0" and a "1," it is quite straight-forward to regularly sample the waveform in software and determine each bit by counting the number of cycles the waveform stays low.

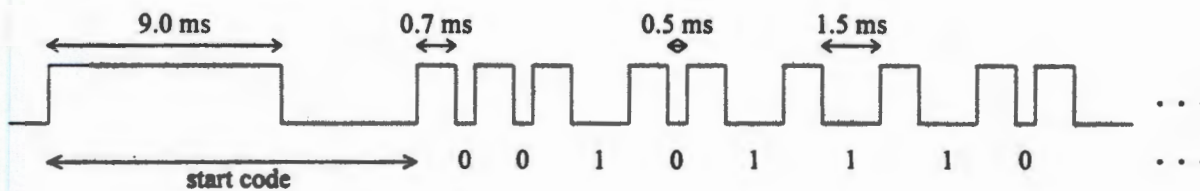


Figure 2.1. Infrared signal from a remote control unit

I wrote the original program in C to do just this, but it ran too slowly; it could not sample the waveform frequently enough to decode it reliably. Fortunately, the Interactive-C software also supported Assembly language programs. When the code to sample the waveform and store it in memory was rewritten in Assembly, it ran faster and returned enough data such that the 32-bit pattern could be decoded. The code which recognized which button was pressed and drove the motors accordingly was written in C.

The Interactive-C libraries came with subroutines to control the direction and speed of the dc motors and to control the angle of the servo motor. The only remaining challenge was to determine a threshold value for the number of cycles below which the bit would be read as a "0" and above which it would be read as a "1." I decided to let the microcontroller determine this threshold automatically using a subroutine I wrote called "calibrateremote" in the C portion of the code. The subroutine expected the user to press a certain button on the remote control for which the bit pattern was already known. By sampling the waveform and comparing it with the

known bit pattern, the subroutine could determine an appropriate threshold value and print it out on the LCD screen of the microcontroller. At the end of this project, the robot could move in six different directions based on commands sent to it using the remote control. The source code for both the C portion of the program and the Assembly portion are given in Appendix A.

## **2.2 The Search for an 8088 Compatible Microcontroller**

As mentioned, the goal of the project was to incorporate a LEGO robot contest into the final few weeks of ECE 291. Since during the first 12 weeks of the semester the students learn and practice low-level programming in 8088 Assembly language and C, it would be most convenient if they could program their LEGO robots in the same languages for MP6. This would eliminate the overhead of learning an entirely new language just for the purposes of the final project.

One option was to design and lay out an 8088-based microcontroller board complete with motor drivers and sensor inputs, starting from scratch; after all, this is exactly what the MIT 6.270 course organizers had done with the Motorola M68HC11 microprocessor. Under the time constraints, however, this seemed unfeasible. Another option was to purchase 8088-based microcontrollers that were specially designed to be used in robotics applications; a long comprehensive search revealed that no such board existed. The final option was to purchase a general-purpose microcontroller and then add extra hardware to interface to the motors and sensors.

Thus the search began for a suitable microcontroller board with an 8088 compatible processor. The microcontroller had to be small so that it could be supported easily by small LEGO robots. It had to have low power consumption at 5 V since it would be running off

batteries. It had to have plenty of digital inputs and outputs and also had to have A/D (analog-to-digital) support. Professor Fuchs, Jonathan Kua, and I contacted several companies which made microcontroller boards and SBCs (single board computers) and requested detailed information on their products; the partial results of this research effort are summarized in Table 2.1.

Many of the microcontroller boards surveyed were not suitable for use in robotics applications, even if extra hardware had been added to them. Several of them did not have latched inputs and outputs, meaning that the only way to interface to the board was through the system bus; this would have unnecessarily complicated the additional circuitry. Most of the 8088 compatible microcontroller boards were modeled directly after the IBM desktop PC, which has neither A/D support nor a large number of easy-to-use digital input/output ports. A couple of microcontrollers, however, did support the necessary features: Micromint RTC-V25 and Vesta SBC88A. We chose the latter because of price and exceptional software support for both C and Assembly language programs. An order for one Vesta SBC88A microcontroller was placed so that we could evaluate it further and determine whether it was indeed suitable for use in controlling LEGO robots.

Table 2.1. Comparison of 8088 Compatible Microcontroller Boards

Company, Product	Processor	Speed (MHz)	Size	Power	Memory	Serial Ports	A/D channels	Digital Inputs	Digital Outputs	Interrupts	Additional Notes
AMD, AM186EM Sparrow	80C186	?	?	?	256K	?	?	?	?	?	80C186 compatible AMD processor, no A/D
Ampro, CoreModule/PC	8088	9.8	3.6"x3.8"	5V, 115mA	256K	1			8		8088-compatible processor, PC/104 bus
Ampro, CoreModule/XT	V20	9.8	3.6"x3.8"	5V, 280mA	256K	1		8	8		NEC V20 compatible w/ 80C88, 1 parallel port, \$290
Ampro, LittleBoard/PC	V40	7.16	5.7"x8.0"	5V, 700mA	256K	2		8	7		\$450
Basicon, MC-2N	V25	?	3.0"x4.0"	5V, ?	32K	2	10	11	5		21 digital I/O lines, A/D on separate board, \$275
HTE, 188SBC	80C188	?	10"x8.4"	12V, 6.3W	512K	2	16	22	23	?	HiTech Equipment Corp., D/A, 45 I/O lines, \$749
Kila Systems, KS-2	V40	7.27	6.2"x3.9"	5V, 200mA	768K	2	1	12	12	?	8088 compatible, 24 I/O lines, bus available, \$187
Kila Systems, KS-3	V40	8	6.2"x3.9"	5V, 200mA	256K	2		12	12	?	8088 compatible, 24 I/O lines, bus available, \$211
Kila Systems, KS-6	V53	12	8.0"x3.9"	5V, 400mA	256K	2		8	8	?	8088 compatible, 16 I/O lines, bus available, \$289
Megatel, PC/+i	V40	10	4.0"x6.0"	5V, 400mA	704K	3			8		Compatible with 80C88, parallel port, bus available
Megatel, PC/+v	V40	16	4.0"x6.0"	5V, 400mA	704K	3			8	8	Compatible with 80C88, parallel port, bus available
Megatel, WCPU/104	80C88	4.77	3.6"x3.8"	5V, 300mA	640K						PC/104 bus available
Micromint, RTC-V25	V25	10	3.5"x5.0"	5V, 350mA	256K	2	8	16	16	?	NEC V25 compatible with 80C88, 32 I/O lines, \$339
Micro/sys, SBC2186	80C186	10	4.5"x7.5"	5V, 575mA	512K	2				3	bus available
Micro/sys, SBC2040	V40	8	4.7"x8.0"	5V, 280mA	512K	2	6	4	7	8	\$425 with A/D and extra RAM
MMT, Inc., MMT-188EB	80C188	?	4.0"x4.0"	5V, 125mA	256K	2	?	12	12	?	Midwest Micro-Tek, Inc., 24 I/O lines, \$348
New Micros, NMIS-0025	V25	?	2.0"x4.0"	5V, 45mA	64K	2		12	12		New Micros, Inc., 3 parallel ports, \$115
R.L.C., PC-186EB	80C186	10	8.3"x6.0"	5V, 450mA	512K	2		16	16	5	R.L.C. Enterprises, Inc., PC/104 bus, 32 I/O lines
R.L.C., MICRO-C188EB	80C188	10	3.0"x6.7"	5V, 250mA	512K	2				3	R.L.C. Enterprises, Inc., PC/104 bus
R.L.C., MINI-C186EB	80C186	10	5.6"x5.8"	5V, 350mA	512K	2				3	R.L.C. Enterprises, Inc., PC/104 bus
Vesta, SBC88A	8088	4	5.5"x4.5"	5V, 350mA	24K	1	8	23	23		Vesta Technology, Inc., \$134
Vesta, Micro88A	8088	8	5.3"x3.0"	5V, 100mA	512K	2					Vesta Technology, Inc., A/D on separate board, \$218
Vesta, Tiny188A	80188	8	3.9"x6.3"	5V, 125mA	128K	2				4	A/D and I/O on separate board, \$506 w/ options
WinSystems, SAT-V40	V40	8	4.5"x7.0"	5V, 350mA	2M	3	8	12	12	8	NEC V40 compatible with 80C88, 24 I/O lines, \$430
Ziatech, ZT88CT01	V40	8	4.5"x6.5"	5V, 430mA	1M	1		24	24	8	48 digital I/O lines, \$375
Z-World, Little Giant	Z180	9.2	4.8"x5.6"	5V, ?	?	4	8	8	8	?	Z-World Engineering, D/A, bus available, \$395

### **3. SPRING 1994 LEGOBOT CONTEST: CAPTURE THE TORCH**

In the middle of the Spring 1994 semester, we began planning the very first ECE 291 LegoBot contest to take place at the end of the semester in the months of April-May. It was in our best interest to have only a few participants in this first contest in order to leave plenty of room for mistakes and experimentation. To keep it easily manageable, we decided to have two teams of two students each. In future semesters after we had gained some experience, the number of participants allowed in the contest would gradually increase to allow the contest to grow at a steady rate. The LEGO parts used for this semester's robots came directly from Professor Fuchs' personal supply and were returned to him after the contest was finished.

#### **3.1 The Microcontroller and Programming Language**

One of the most important aspects of the ECE 291 LegoBot contest is the microcontroller and programming language used. At this time, we had not yet evaluated the Vesta SBC88A microcontroller board nor had the opportunity to design any add-on hardware for it. Therefore, the best choice was to use MIT's 6.270 microcontroller boards for this first contest. The participants would have to learn either C or Motorola 68HC11 Assembly language on their own in order to program their robots. Since there were only four participants, however, this was not a problem; in addition to being given copies of portions of the 6.270 course notes, they could receive plenty of individual assistance from the contest organizers, Kevin Safford and me. As part of the preparation for the contest, Kevin assembled the second 6.270 microcontroller board.

Including the one I had assembled for controlling my first robot, Woody, we now had two working 6.270 boards for use in the contest.

As far as development platforms were concerned, the MS-DOS based PC seemed to be the obvious choice since the ECE 291 laboratory was equipped with enough of them and also since most students had their own PCs at home. Furthermore, the MS-DOS version of IC (Interactive-C), the software used to communicate with the 6.270 board, was well-supported by MIT and relatively bug-free. Once this decision had been made, IC was installed on the machines in the ECE 291 laboratory, and copies were made on 3-1/2" floppy diskettes to be included in the kits. One of the nice features of IC is that it comes with simple library subroutines, which can be used to read data from the sensors and control the motors. A small sample of such routines is listed below.<sup>7</sup>

```
void motor (int m, int p)
void alloff ()
void servo_on ()
void servo_off ()
int servo_deg (float angle)
int digital (int p)
int analog (int p)
int dip_switch (int sw)
void beep ()
int start_process (...)
int kill_process (...)
```

As implied by the last two functions listed, IC allows multitasking. One could, for example, have a process which continuously checks the status of a particular sensor, and sets a global variable based on it. Meanwhile, the main process could be continuously checking that global variable and controlling the motors based on that information. One of the drawbacks of

---

<sup>7</sup> One should consult the 6.270 course notes for more complete and detailed information about IC's capabilities.

the version of IC that was available at the time was that it only supported a subset of the ANSI C language. For example, it did not support complex data structures such as pointers. Newer versions of IC, however, do support most of the standard C language.

### **3.2 Contents of the LegoBot Kits**

The next set of decisions to be made involved the parts to be included in each of the LegoBot kits that would be provided to the two teams. In addition to the LEGO parts, the four dc motors, the servo motor, the microcontroller board, and the diskette containing a copy of IC for MS-DOS, we had to consider what other types of actuators and sensors should be included.

As far as actuators were concerned, we decided that the four dc motors and the servo motor would be sufficient to allow creative yet simple robot designs. However, since the 6.270 package ordered also had come with solenoids, these were included in the LegoBot kits as well. The solenoids had a magnetic shaft and a spring; when a voltage was applied across the two leads of a solenoid, the shaft would get “sucked” into the solenoid, and when the voltage drop was removed, the spring would push the shaft out again. One possible application of this type of actuator might be to open and close a claw or gripper on the robot. These solenoids could be wired directly into the motor ports and could be controlled through software in the same way that the motors were controlled. For power supplies, we would provide a 6 V AA battery pack to drive the microcontroller and a 6 V Gates battery pack to drive the motors, just as in the 6.270 course.

The only sensors that came with the 6.270 board were the GP1U52X infrared receiver modules. Aside from these, we had to select a set of sensors which would be included in the

LegoBot kits. It was important to give the teams a reasonable variety of both digital and analog sensors. Digital sensors, when wired correctly, can be read into the program as simply 0 or 1. The analog sensors, on the other hand, return an analog voltage between 0 V and 5 V, which is then converted by the processor to a digital value between 0 and 255. We wanted to have as little redundancy as possible among the different types of data the sensors could provide to the microcontroller. For example, it did not make much sense to include two different types of phototransistors (light sensors) in the kits if they both were essentially capable of returning the same type of information. Availability, easy-of-use, and cost were also important considerations in selecting appropriate sensors. Based on these criteria, we decided to give each team various quantities of each of the following types of sensors: infrared receivers (digital), microswitches (digital), a condenser microphone (digital), reflectance sensors (analog), phototransistors (analog), and potentiometers (analog).

The infrared receivers simply respond with 0 or 1 based on whether or not they are currently receiving light in the IR range. The microswitches are low-force switches, which are ideal for detecting collisions with other objects. The condenser microphone could be used to detect sound; the robots would have to use this to "listen" for the starting signal, which was the sound of a gunshot. A reflectance sensor determines how reflective a surface is in the infrared range. For example, a white surface would be very reflective whereas a black surface would not reflect any infrared light. The reflectance sensors must be placed physically close to the reflective surface (less than 1/8") in order to work properly. A phototransistor measures the amount of visible light reaching it. Finally, a potentiometer measures angle of rotation. It could, for example, be used to determine the current position of a robot arm.



The generic 3-pin digital and analog sensor ports on the 6.270 microcontroller board were designed to interface easily to any type of sensor. The ports essentially operated on the principle that if the board provides the sensor with power (5 V) and ground, then the sensor should be able to provide the board with the appropriate data (on the third pin). Therefore, some of the sensors we selected interfaced easily to the digital and analog ports of the board. Other sensors, however, such as the reflectance sensors and the phototransistors required that we solder resistors with certain values directly to them so that they could be plugged into any of the analog input ports and give appropriate voltages between 0 V and 5 V. Circuit diagrams showing how each sensor was wired are given in Chapter 4.

All of the parts that were contained in the Spring 1994 LegoBot kit are listed in the handout which was distributed to the LegoBot participants. This handout is duplicated in Appendix E. For the motors and most of the sensors, we glued them to appropriate LEGO pieces before giving them to the participants. The intent was that all the kits would be consistent so they could be used every semester. Otherwise, each team might have attached the motors and sensors to the LEGO in their own way, which might not be convenient for a team with the same kit next semester. Therefore, we glued LEGO gears and plates to parts such as the motors and potentiometers in a way that they could be mounted on the robot and still interface easily to the surrounding LEGO parts (other gears, axles, and bricks).

### **3.3 Contest Task and Rules**

In order to have a successful LegoBot contest, an interesting task for the robots to accomplish was necessary. It had to be challenging, but at the same time simple enough that the

participants could design a successful robot in the four-week time period. One of the goals was to have a contest in which the robots would have to use data from almost all of the sensors which were provided. We were also striving for a contest in which the robots would be forced to interact with each other.

The task we finally decided upon was called "Capture the Torch." The two robots would start at opposite corners of a 4' x 4' table. In the center of the table was a "torch," which was essentially a light bulb on a stick. The torch was stuck into a hole in the center of the table. The objective for the robots was to find the torch, pick it up out of the hole, and deposit it in the opponent's corner. The first robot to accomplish this would win that round. A more detailed description of the task and the rules is given in Appendix E.

The torch contained a small flashlight bulb, a power switch, and two AA batteries. A circuit diagram is shown in Figure 3.1.

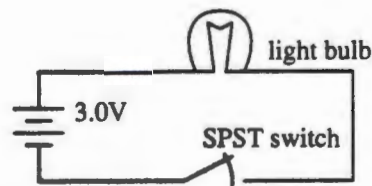


Figure 3.1. Circuit Diagram for the Torch

The playing field was designed to allow the robots to gather a large variety of information so that they could determine their position on the field. The table was surrounded by walls so that the robots could use the microswitches (bump sensors) to detect the edge of the field. The surface of the field had been painted with different colored rings to help the robot determine with the reflectance sensor how far it was from the center (see Figure E.1). Also on two opposite corners of the field, there were infrared transmitters to help cue the robots as to which direction

they were facing. The transmitters we used were the infrared beacons that came with the 6.270 board. The beacon is basically a small printed circuit board with eight infrared LEDs and eight red LEDs. The beacon at one corner of the field would emit IR at 100 Hz and the one at the other corner would emit at 125 Hz. The Interactive C software came with routines which could be used to easily determine which of these two frequencies of IR the robot was receiving. Since the IR receiver modules only detect IR on a 40 kHz carrier signal, we had to superimpose this carrier frequency on top of the 100 Hz or 125 Hz signal in order for the robots to be able to detect it.

The circuitry used to drive the IR beacons is shown in Figure 3.2. A 556 dual timer is used to generate the 100 Hz and 125 Hz waveforms. These signals are sent into the Master Reset pin of the 74LS390 decade counters, which are configured as divide-by-fifty counters. Since the clock speed for the counters is 2 MHz, they will output signals modulated at 40 kHz as shown in the figure. This waveform is then fed into a simple noninverting amplifier stage so as to provide enough current to drive all the LEDs. The LM386N-1 is actually an audio amplifier, but any operational amplifier would have sufficed. The LM386N-1 was used only because of availability at the time. We used four AA batteries (6 V) to drive the circuitry and the LEDs.

### **3.4 Contest Results**

Before MP6 officially began, I introduced the concept of LEGO robots to the ECE 291 class and described the project, giving the students a flavor for what was involved. I gave a brief description of the contest rules for the Spring 1994 semester, and also demonstrated to the class the nonautonomous LEGO robot which I had built called Woody (see Section 2.1). The class

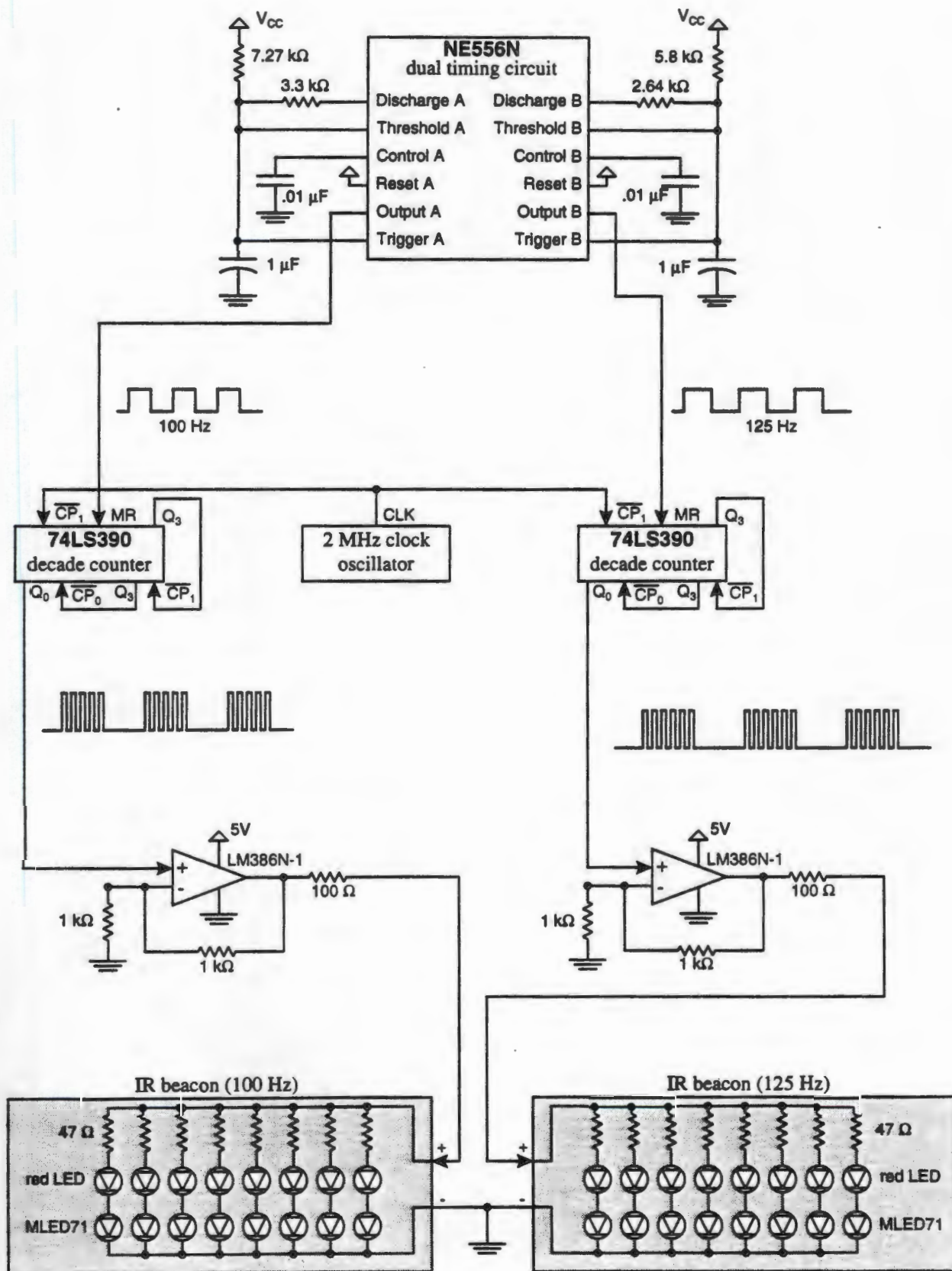


Figure 3.2. Circuit Diagram for 100 Hz and 125 Hz IR Transmitters

seemed excited, and it was immediately obvious that there would be considerable interest in this type of project. Since we had already decided that only four students would be involved in the LegoBot contest this semester, not everyone interested was able to participate. Dan Moore and Jason Wessel formed one team; Michael Landauer and Ted Briggs formed the other team. The two teams were given their kits, and they began work on their LEGO robots.

At the end of the four-week period, both teams had built successful robots programmed entirely in C. The contest took place in class on the same day as the rest of the class demonstrated their MP6 projects. It was originally intended that there would be three rounds of competition between the two robots, and the robot that won the best two out of the three rounds would be declared winner of the contest. However, in each of the first two rounds, both robots got entangled with each other in the middle of the table, and neither was able to complete the goal. At this point, we decided to do away with the formal contest, and simply have informal individual demonstrations. Since both robots were able to successfully carry the torch to the opposite corner in the absence of another robot on the table, the contest was declared a tie. All four participants had gained valuable experience in the use of embedded microcontrollers, and they all mentioned having learned a lot by doing the LegoBot project.

Despite the success of this very first ECE 291 LegoBot contest, many problems were discovered throughout the semester which would have to be corrected. The following are brief descriptions about some of the troubles encountered.

- The task for the robots ended up being more of a hardware challenge rather than a software challenge, as it was intended. The final software for both of the robots was quite

short and simple. A trivial algorithm, which requires almost no sophistication in terms of closed feedback loops and intricate control mechanisms, is outlined below:

- 1.) Spin until light is detected.
  - 2.) Move forward for a predetermined amount of time.
  - 3.) Pick up torch.
  - 4.) Move forward for a predetermined amount of time.
  - 5.) Drop the torch.
- The table, which was 4' x 4' in size, was too small for two robots to play at the same time. It took very little time for the robots to travel from a corner to the center, and it was too difficult for the robots to get around each other once they became entangled.
  - Originally, it was planned that the top of the torch would be round as shown in Figure E.2. However, the torch we constructed ended up with a square-shaped top instead. This made grabbing the torch slightly more difficult for the robots, because the distance from the edge of the torch to the stick varies depending on the angle from which the robot is approaching.
  - The robot designers encountered other problems with the torch as well. It was originally powered from five AA batteries which made it especially heavy. Later the power supply was reduced to two AA batteries to reduce the weight. In addition, the torch was slightly unbalanced, causing it to fall unpredictably when the robots grabbed it. Also, the light bulb on the torch was not bright enough, especially when the robots were operating in a well-lit room.

- One of the most frustrating physical problems with the contest was that there was too much friction between the stick of the torch and the hole in the playing field. This meant that unless the torch was perfectly vertical, it required a considerable amount of force to lift the torch out of the hole.
- The IR transmitters on the two corners of the playing field were not mounted securely and kept falling off. Fortunately, in the end, the robots did not need to use their IR receivers to navigate around the table.
- Most of the various sensors given to the teams were not used due to the simplicity of the algorithm required to solve the task. The environmental information provided by the torch and the playing field was found to be redundant and often unnecessary. For example, the teams did not make use of the microswitches (bump sensors), potentiometers, infrared receivers, or reflectance sensors.
- I was unable to find an easy way to interface the condenser microphone element to the microcontroller. Rather than having the robots start each round automatically when they detected the sound of a gunshot, the teams ended up starting their robots manually by pressing the “reset” button on the microcontroller.
- Many of the parts given in the kit were incessantly breaking and coming apart during the development. For example, the LEGO gears and plates would not stay glued to the motors and sensors. Also the male header pin connectors kept coming unsoldered as the teams would continuously bend and twist the wires to fit them into the ports of the microcontroller.

- There was very little space in the ECE 291 laboratory to store the playing field. The teams would have to move it out into the hallway every time they needed to work with it.



#### **4. FALL 1994 LEGOBOT CONTEST: EIGHT BALL**

In the Fall 1994 semester, ECE 291 was taught by Professor Michael C. Loui. Professor Loui was amenable to letting the students participate in the LegoBot contest again as part of the final project, MP6. He was extremely supportive and enthusiastic. Early in the semester, we had ordered enough parts and equipment to put together six complete kits so that we could have four teams taking part in the project. This would leave two kits unclaimed for use as spare parts and for experimentation purposes. Since Professor Loui decided to have all of the students work in groups of four for MP6, we ended up having a total of 16 participants for the Fall 1994 LegoBot contest.

##### **4.1 The Vesta SBC88A Microcontroller and Additional Hardware**

One of the goals for this semester was to use the 8088-based Vesta SBC88A microcontroller board. Since it was only a general-purpose microcontroller, it did not have any motor drivers or PWM circuitry on it. These items had to be added in the form of an add-on board, which would sit directly on top of the microcontroller, connecting to its ports. The complete circuit diagrams for this additional hardware are shown in Figure 4.1. In these diagrams, all logic is powered from  $V_{CC}$ , which is 5-6 V obtained from four AA batteries, the same source which powers the microcontroller board. The motors are driven off  $V_{motor}$ , which is obtained from three 2 V (total 6 V) Gates rechargeable cells, the same ones used for the LegoBot contest the previous semester.

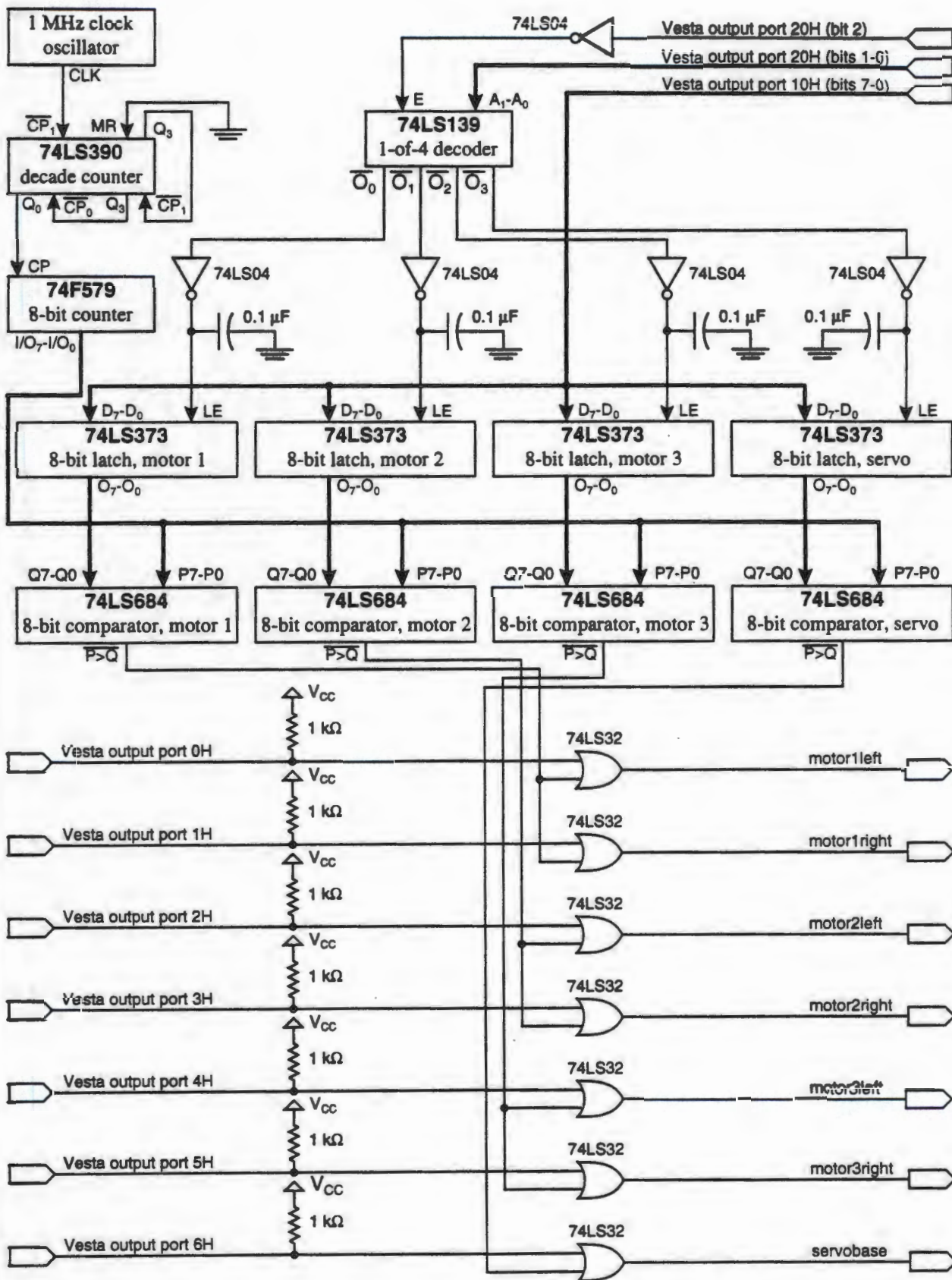


Figure 4.1. Circuit Diagram for Additional Hardware on the Vesta Microcontroller

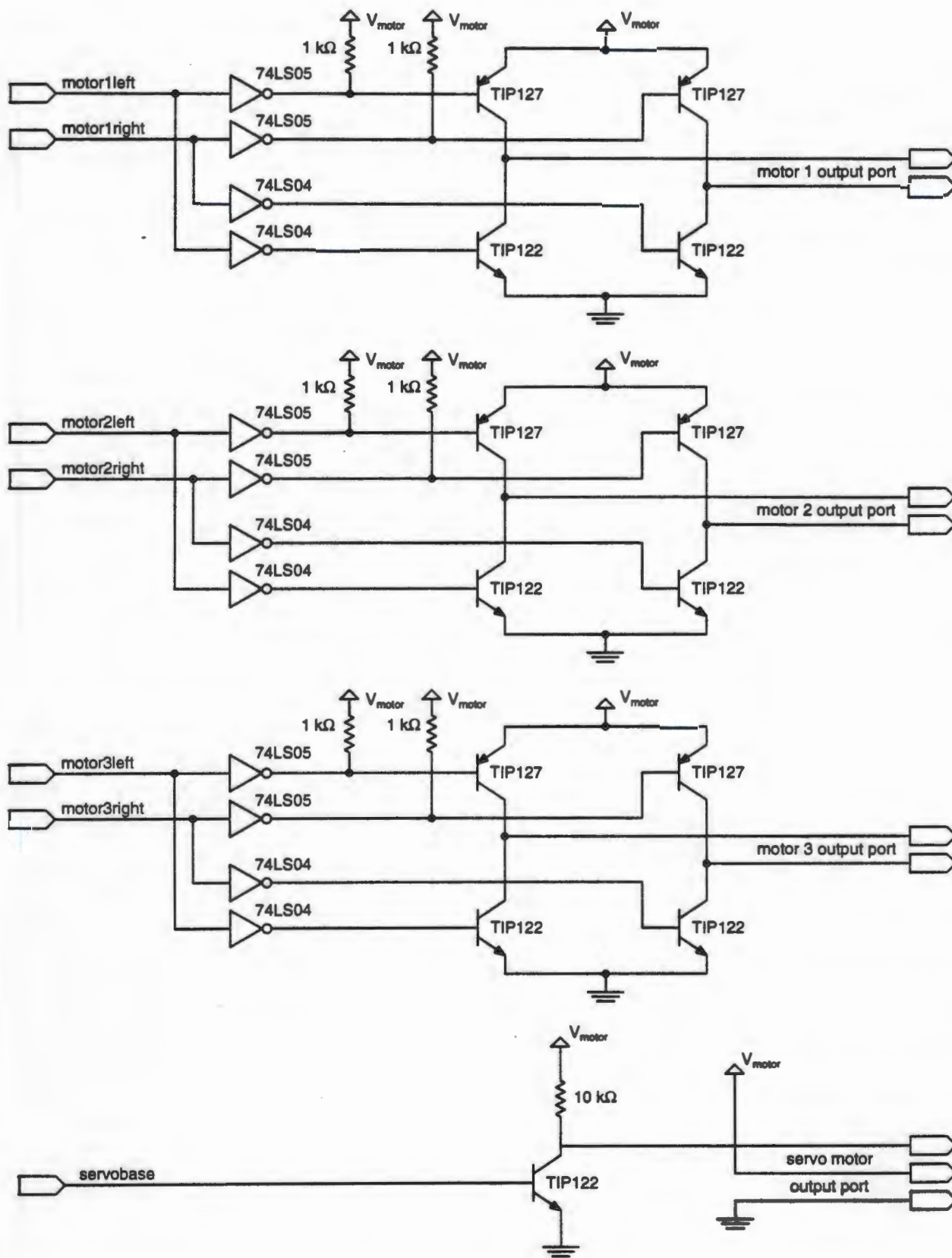
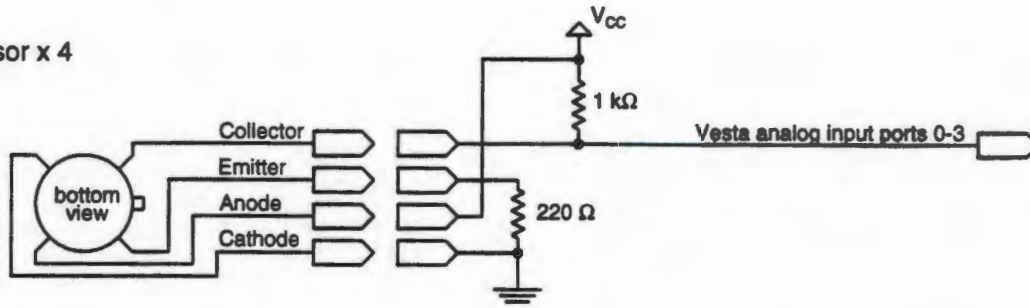
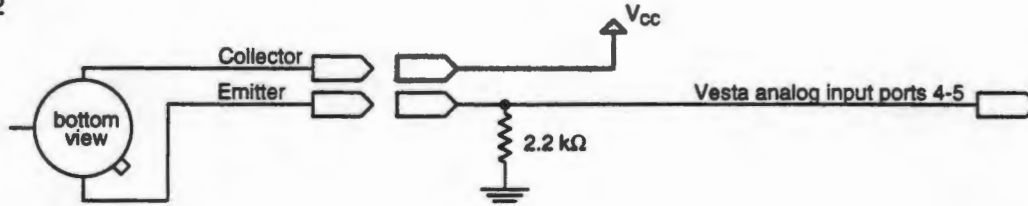


Figure 4.1 (cont.). Circuit Diagram for Additional Hardware on the Vesta Microcontroller

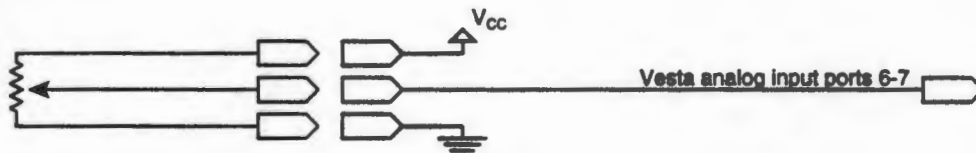
Reflectance Sensor x 4



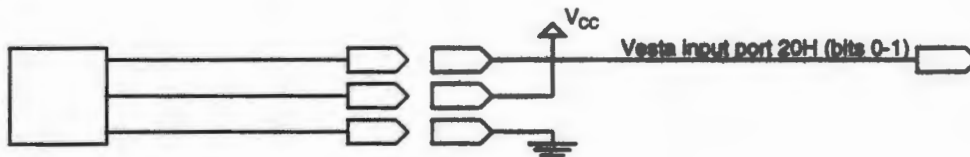
Phototransistor x 2



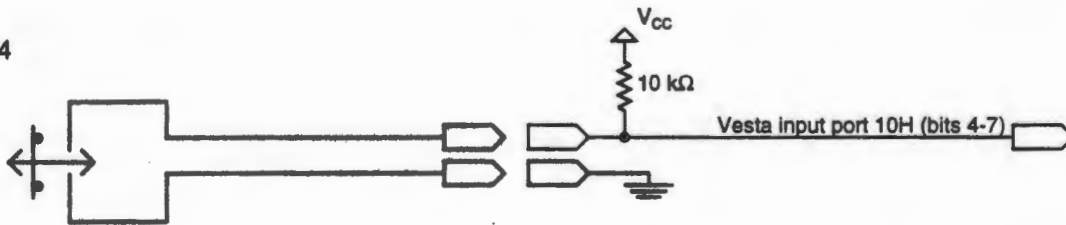
Potentiometer x 2



Infrared Receiver x 2



Microswitch x 4



DIP Switch x 4

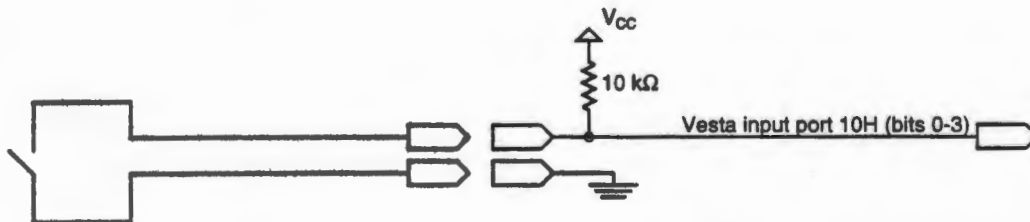


Figure 4.1 (cont.). Circuit Diagram for Additional Hardware on the Vesta Microcontroller

The majority of this hardware is designed to control the speed of the dc motors. This is done by using a well-known technique called Pulse Width Modulation (PWM). The theory behind PWM is that by turning the motors on and off rapidly, the motors actually spin at a lower speed since the average power being supplied is reduced. By varying the width of the pulse being applied to the motor drivers, we can control the speed of the dc motors. The duty cycle is the width of the pulse divided by the total period. Examples are shown below in Figure 4.2. The waveforms in this diagram have a frequency of 100 Hz, but PWM frequencies can range from 100 Hz to 20 kHz depending on the motor.

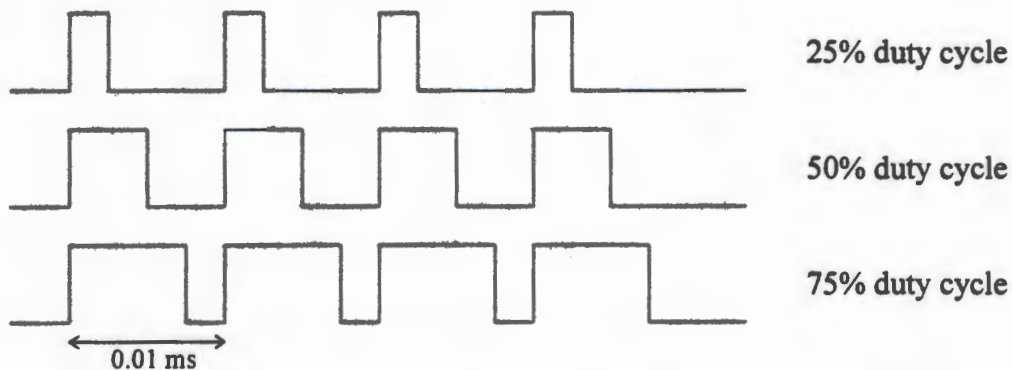


Figure 4.2. Pulse Width Modulation

To generate these PWM waveforms, we first store 8-bit values into the 74LS373 latches; these values represent the duty cycle for each of the three dc motors. A 0 corresponds to a 100% duty cycle (motor is on full speed) and a 255 corresponds to a duty cycle of almost 0%. The program running in the microcontroller can store values into these latches by writing the 8-bit value to port 10H and then writing the appropriate address (0, 1, 2, or 3) out to port 20H. The 74LS139 will make sure the value is stored in the appropriate latch. The capacitors were necessary to eliminate false latching due to glitches in the output port 20H.

If we compare the latched value for a motor with the value coming out of a free running 8-bit counter (74F579), the output of the comparator will be a PWM waveform. For example, if the latched value is 127, then the value coming from the counter will be greater than 127 half of the time and less than 127 the other half of the time. Therefore, the duty cycle of the signal coming from the comparator will be approximately 50%. The frequency of this signal will be exactly  $1/256$  of the frequency of the counter's clock. The counter's clock, in this case, is running at 20 kHz, which is obtained by dividing down a 1 MHz clock oscillator. This is done with a 74LS390 decade counter in a divide-by-fifty configuration.

The outputs of the comparators are fed into the motor driver stage, once they have been checked against the values that control the motor direction (Vesta output ports 0H through 5H). The motor drivers consist of H-bridges with bipolar PNP and NPN transistors that allow a motor to be driven in both directions. If two transistors diagonally across from each other are both on simultaneously and the other two are off, the motor spins in one direction. If the opposite two transistors are turned on, the motor spins the other way.

It is important that a PNP and an NPN transistor, whose collectors are connected, do not turn on simultaneously to avoid short circuits between power and ground. Thus the logic has been designed such that this can not happen. The base of the PNP transistor has to be pulled up to  $V_{\text{motor}}$  in order for the transistor to be off, and it must be at 0 V for the transistor to be on. This is the reason for the open collector inverters (74LS05). Since the NPN transistor operates on the opposite voltages (0 V for off and  $V_{\text{motor}}$  for on), we can feed the same voltage into its base so that corresponding PNP and NPN transistors can never be on simultaneously. However, we use a separate 74LS04 inverter to avoid "fighting" signals.

The servo motor also operates on the PWM concept. Different pulse widths cause the servo motor to turn to different angles. Specifically, a pulse width of approximately 0.7 ms causes the servo to go to one extreme while a pulse width of 2.0 ms causes it to turn to the other extreme (180 degrees of rotation). The frequency of the pulses should ideally be 50 Hz, but here we use 80 Hz to minimize the number of components in the circuit. The servo motor has three lines coming from it: power, ground, and control. Since the servo has only one control line, we do not require a full H-bridge circuit here; a single NPN transistor suffices. The output port 6H from the Vesta microcontroller turns the servo motor on and off.

The additional hardware to interface with the sensors was minimal. Diagrams of all six types of sensors are shown in Figure 4.1, along with the interface circuitry on the add-on board. Convenient 2-pin, 3-pin, and 4-pin ports were put on the board, so the users could plug in only the sensors which they needed.

The Vesta SBC88A microcontroller came with MS-DOS based software to download programs to the microcontroller and debug it in real time. This software, called "C\_thru\_ROM," is produced by Datalight.<sup>8</sup> With it, one can download an executable file created using any version of Microsoft Assembler (MASM) or version 6.0 or earlier of Microsoft C/C++. The software communicated with the microcontroller board through a 4800 baud serial connection. The only special consideration is that the program must be linked with an object file which was provided (ST.OBJ); this object file contains the unique startup code required to run a program on the microcontroller as opposed to running it on a desktop PC. Brief instructions on how to download a program to the microcontroller are given in Appendix F.

---

<sup>8</sup> For more information, contact Datalight / 307 N. Olympic Ave., Suite 201 / Arlington, WA 98223 / (206) 435-8086.

In addition, one could link the program with a library containing special input and output routines for the microcontroller. This library had a "printf" routine which would send the data through the serial port out to the desktop PC. As long as the C\_thru\_ROM software was still running and the serial cable still connected, the data would show up on the screen. This turned out to be extremely useful for debugging purposes.

For demonstration purposes, a robot called Norm was built using the Vesta microcontroller and the new hardware. This nonautonomous robot was designed to perform the same function as Woody: respond to infrared commands from a remote control unit. The source code to control this robot was written entirely in C and is listed in Appendix B. The program waits for an IR signal, decodes it, and sets the left and right motors to the appropriate state (forward, backward, or stop) depending on which button was pressed. This allowed the robot to move in any of eight different directions. The hardware for the robot simply involved two motors and an infrared receiver all connected to the completed microcontroller board.

#### **4.2 Contest Task and Rules**

After brainstorming with Jonathan Kua and Steve Sawatzky to come up with a more interesting contest task than the one from the previous semester, we decided on a game of pool known as Eight Ball. In the real game, there are seven solid colored balls, seven striped balls, and one black eight ball. In our modified version, there would be seven red balls, seven green balls, and one black eight ball. The game involved one robot trying to place the green balls into the pockets and the other robot trying to place the red balls into the pockets. At the end of the time limit (3 min), the robot which had more of its own balls in the pockets won the round. Both



robots had to try to avoid pocketing the black eight ball until all of its own balls were sunk. The robots would play on a specially designed pool table with large pockets and stiff wooden beams as side walls.

The complete contest rules are given in Appendix F, which contains a copy of the exact handout distributed to the students. The handout also contains a list of all the parts and equipment distributed to the teams, including the companies they were ordered from. The add-on boards, which attached to the microcontroller, were built and wired by hand since there were insufficient resources available to have printed circuit boards made. The parts used in the add-on board are included in the list of parts. In addition, the handout gives some sample source code for interfacing to the motors and sensors. Finally, it contains brief instructions for using the microcontroller board.

The pool table had six pockets: one on each of the four corners and one in the middle of each of the two sides (see Figure F.1). An infrared transmitter was mounted on each of the pockets to aid the robots in finding the pockets. Since the IR receivers only detect signals modulated at 40 kHz, a small circuit had to be designed to drive the IR transmitters on the pool table with this carrier frequency. This simple circuit is shown in Figure 4.3. Also, to quickly test whether the IR transmitters were working, another small board containing an IR receiver and a visible LED was built. This is shown in Figure 4.4. The NOR gate is used to both invert the signal and provide enough current to drive the LED; any gate (NAND gate or inverter, for example) would have sufficed.

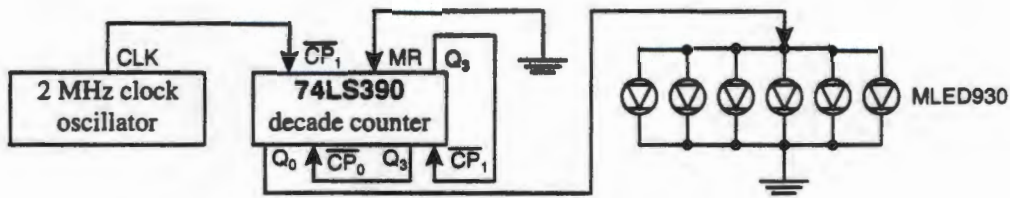


Figure 4.3. Circuit Diagram for Pool Table IR Transmitters

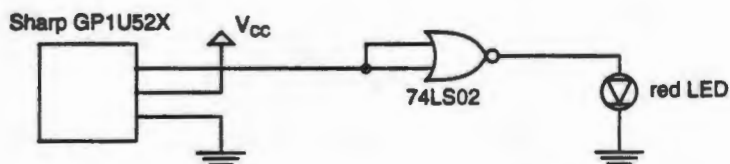


Figure 4.4. Circuit Diagram for IR Test Module

### 4.3 Contest Results

From the ECE 291 class of Fall 1994, sixteen students grouped into teams of four were selected to participate in the LegoBot contest. The people on these teams are named below.

Yellow Team: Todd Baker, Jake Battle, Jay Monkman, Gary Tsai

Red Team: Kian-Teik Beh, Matthew Bryan, Ben Cho, Zachary Zuzzio

Green Team: Ryan Akkerman, Kwun Ho, John Knapowski, Dennis Koutsoures

White Team: Chris Bertelsen, Amr Haggag, Kurt Lewinski, Kevin Sawatzky

Each team was given one complete kit including the microcontroller board with the add-on hardware.

Each team was able to design a creative LEGO robot, which solved the task in its own unique way. The Yellow Team built a robot which was extremely fast compared to the others. It did not detect collisions; rather it simply backed up on its own after a certain timing delay assuring it would never get caught in a corner. This robot had a gate which was normally in the up position. When a ball entered the gate, it would close the gate, detect the color of the ball, and

either open the gate to dispose of it or run full speed towards a pocket to allow the ball to fall through the bottom. The Green Team built a robot whose special feature was to start the contest off by backing up directly towards the triangle of balls. Since it had a large rear bumper, most of the balls of its own color would fall into the corner pocket. Furthermore, it would have protected nearly all of the balls so that the opponent robot could not access them. The Red Team's robot had a tunnel going through the robot. The balls would enter the tunnel and the robot would check the color of the ball. If the ball was its own color, it would close a small gate causing the ball to travel with the robot, and proceed to drop it in a pocket. If the ball was any other color, it would simply keep moving forward and the ball would exit from the rear untouched. Finally, the White Team designed a robot which had a mechanism to shoot the balls from a distance. It would normally line itself up with the strongest IR signal, and then shoot the ball. It was also programmed to travel along the wall until it reached a pocket, just in case IR could not be detected.

The actual LegoBot contest took place on a Friday afternoon in a large open classroom. All of the ECE 291 students and teaching assistants were invited to watch the event; in addition, several of the participants had invited their own guests. Thus there was a large crowd of people present, adding to the excitement of this occasion, which was the culmination of four weeks of hard work. By random selection, it had been decided that the Red and Yellow teams would battle each other first, after which the Green and White teams would battle. The losers of those rounds would compete once after that, and finally the two winners would battle for the championship. In the first round, the Yellow Team accidentally knocked the eight ball into a pocket, ending the round early and allowing the Red Team to move on. In the second round, the

Green Team was disqualified because their robot kept false starting; it would begin moving before the IR transmitters had been turned on. Since they were disqualified, the losers' round was canceled. And in the final round, the Red Team won over the White team simply by placing more of its own balls into the pockets. Overall the contest was a great success, and afterwards everyone stayed to hold "grudge" matches against each other and to do individual demonstrations.

Once again, despite the success of the Fall 1994 LegoBot contest, we encountered several frustrating problems over the course of the four weeks. These are briefly described below.

- Since the teams were allowed to use up to \$10 in additional parts, several of them generated the idea of placing IR transmitters on their own robots in order to "confuse" the opponent. Since the IR signal is used to start the contest, IR transmitting robots could potentially cause the whole contest to be a disaster. Therefore, a special rule was added late in the semester to avoid this problem.
- Since almost all of the chips on the microcontroller board were TTL chips, the AA batteries used to power the electronics were being consumed very quickly. Some teams went through two or three sets of batteries per day. In the future, it will be necessary to use CMOS boards, if possible, to reduce power consumption.
- Again, the gears and LEGO pieces kept coming unfastened, especially from the motors. Also, connectors occasionally came unsoldered from the wires and sensors.
- Another problem we encountered was with the servo motor. It was discovered that the servo motor would not operate correctly if any of the dc motors were on simultaneously.

This was due to noise generated by the dc motors which could not be filtered out even with the use of capacitors.

- Two of the teams were using the LEGO chain links to construct a tanklike vehicle. It turned out that there were not enough of these pieces to build a complete tank with the track drive on both sides of it. Therefore, the teams had to share parts.
- The wires attached to the sensors were often not long enough to reach from the sensors to the microcontroller. This was an easy problem to fix by making extension cables with the appropriate number of wires.
- For an unknown reason, the C\_thru\_ROM software could not download programs quickly to the Vesta microcontroller from the Pentium machines. It worked fine on 386 or slower PCs. Since the ECE 291 laboratory is equipped with Pentiums, this was a major problem. Fortunately, we managed to keep one 386 in the laboratory for the students to use when downloading their programs.
- The microcontroller add-on board had too much metal exposed, especially from the bipolar power transistors. This made it easy to accidentally create a short circuit while trying to plug in the sensors and motors.
- The pool table was kept propped up against the wall in the hallway outside of the ECE 291 laboratory. When the teams had to use it, they would have to place it flat on the ground. There was very little space in the hallway and also very little light, especially when all four teams were working simultaneously.

## **5. ENGINEERING OPEN HOUSE 1995**

At the same time as preparations were being made for the Spring 1995 LegoBot contest, I was putting together a LEGO robots exhibit for the Engineering College's annual Engineering Open House. This is an event in which students from all engineering departments and societies display their projects and compete for awards in various categories. We had kept the four LEGO robots from the previous semester intact, so that they could be used as part of our exhibit. In addition, I had designed a new robot called Cliff which was also on display during Engineering Open House.

### **5.1 "Cliff" -- Infrared Controlled Robot Which Received Commands from a PC**

This robot, like Woody and Norm, was able to receive infrared commands, decode them, and act on them. In this case, however, the infrared signals were being transmitted by an expansion card in a PC instead of a person holding a remote control unit. Indirectly, though, the user was still controlling the robot; the Microsoft Windows software interface on the PC had nine buttons which the user could click with the mouse to make the robot travel in different directions. This software on the PC communicated directly with the expansion card to transmit the appropriate infrared signals.

The M68HC11EVBU microcontroller board on the robot was responsible for decoding the infrared signal and controlling the motor drivers. The program for the microcontroller is given in Appendix C. Figure 5.1 contains a circuit diagram which shows exactly how the motor drivers and infrared receiver were interfaced to the microcontroller board. I used two MPM3004

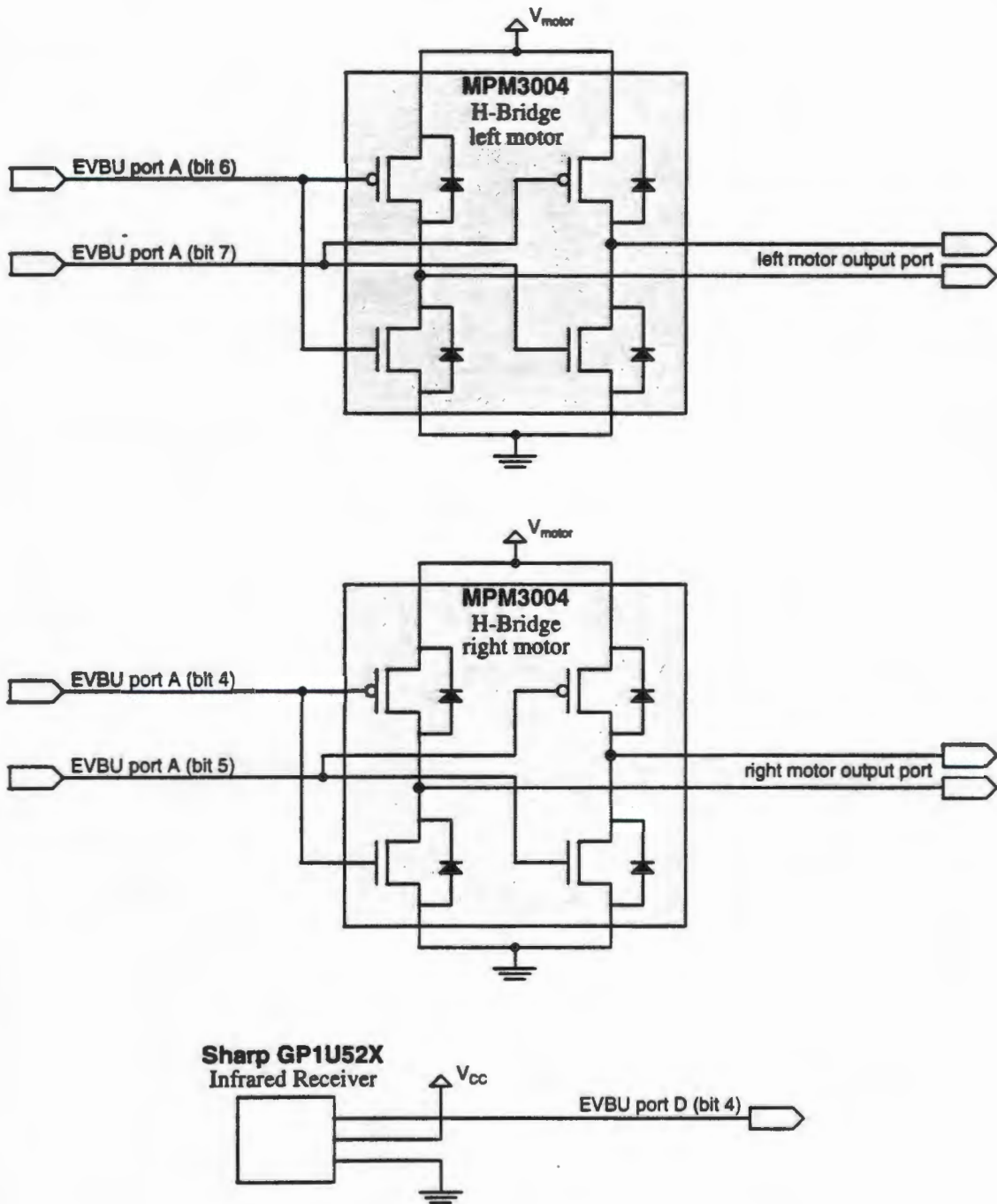


Figure 5.1. Circuit Diagram for Cliff

parts from Motorola, each of which contains a full H-bridge sufficient to drive a single dc motor in either direction.

The expansion card for the PC is designed to wait idly until a 16-bit data value is written to the I/O address 0300H. When this data is received from the ISA bus, it is parallel loaded into a 16-bit shift register (two 74LS323's). At this point, the data is serially shifted out one bit at a time through the pulse code modulation logic which creates the proper waveform to represent the bit pattern. A series of 555 timers is used to generate the proper timing delays. For example, the first 555 timer is used to generate the 9.2 ms start code present in the infrared encoding scheme; this timer is only used once for every 16-bit word that is transmitted. The second timer generates a 0.5 ms pulse for every "0" bit, and a 1.5 ms pulse for every "1." Finally, the last timer generates a 0.7 ms delay in between the pulses which represent the bits. Each timer automatically triggers the next timer in sequence until the 74LS161 counter determines that all 16 bits have been transmitted.

The final step in the sequence is to superimpose a 40 kHz carrier frequency on top of the generated waveform. This is done using a 74LS390 decade counter clocked at a frequency of 2 MHz. The decade counter divides the clock by 50 to generate the 40 kHz signal. This signal is transmitted through the high current infrared LEDs, part number MLED81. These LEDs were mounted so that when the expansion card was inserted into the PC, they would transmit from the back of the computer. The complete circuit diagram for the expansion card is shown in Figure 5.2. The program for the PAL which is used to handle the combinational logic is given in Appendix C.



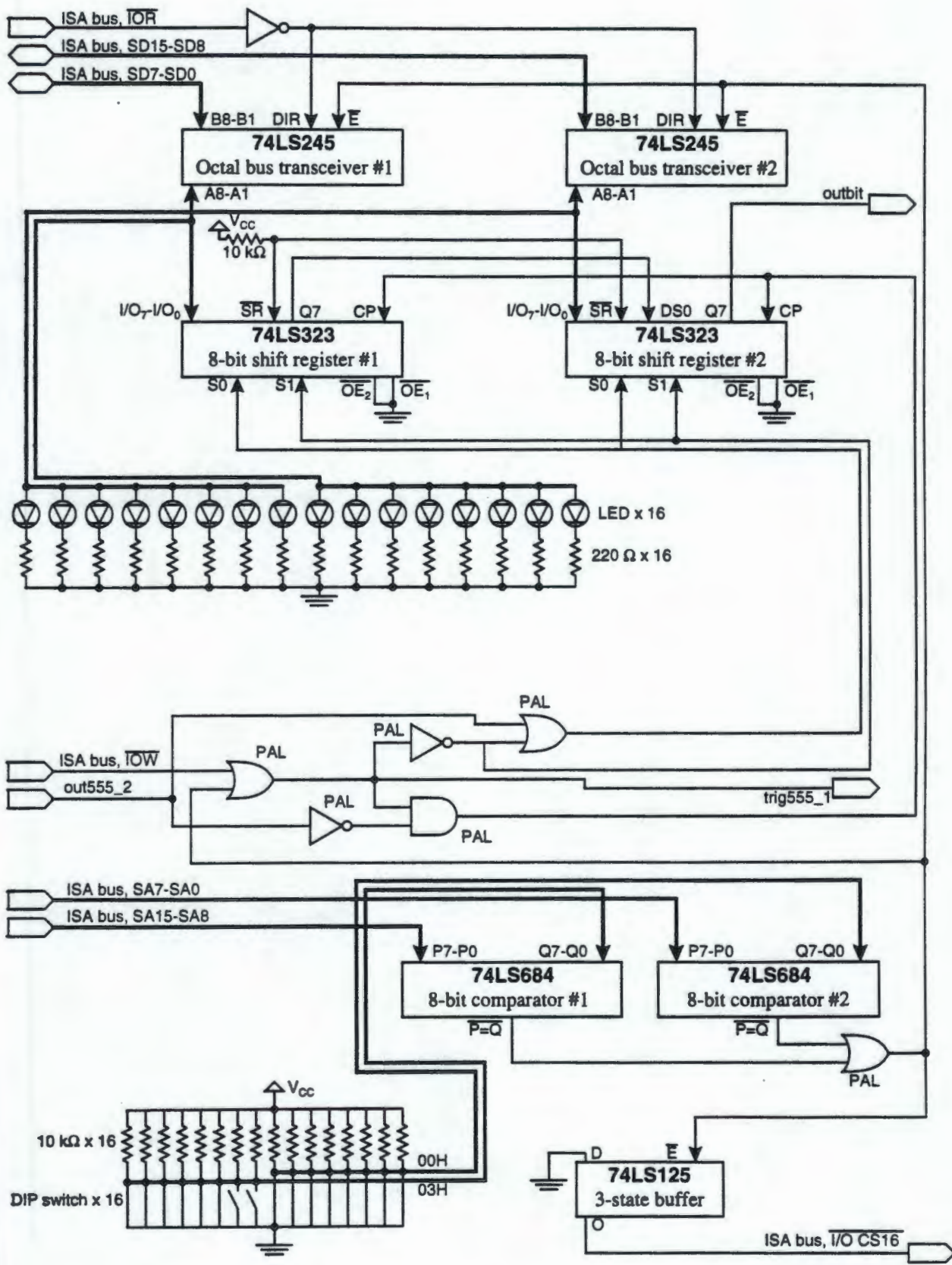


Figure 5.2. Circuit Diagram for IR Transmitting Expansion Card

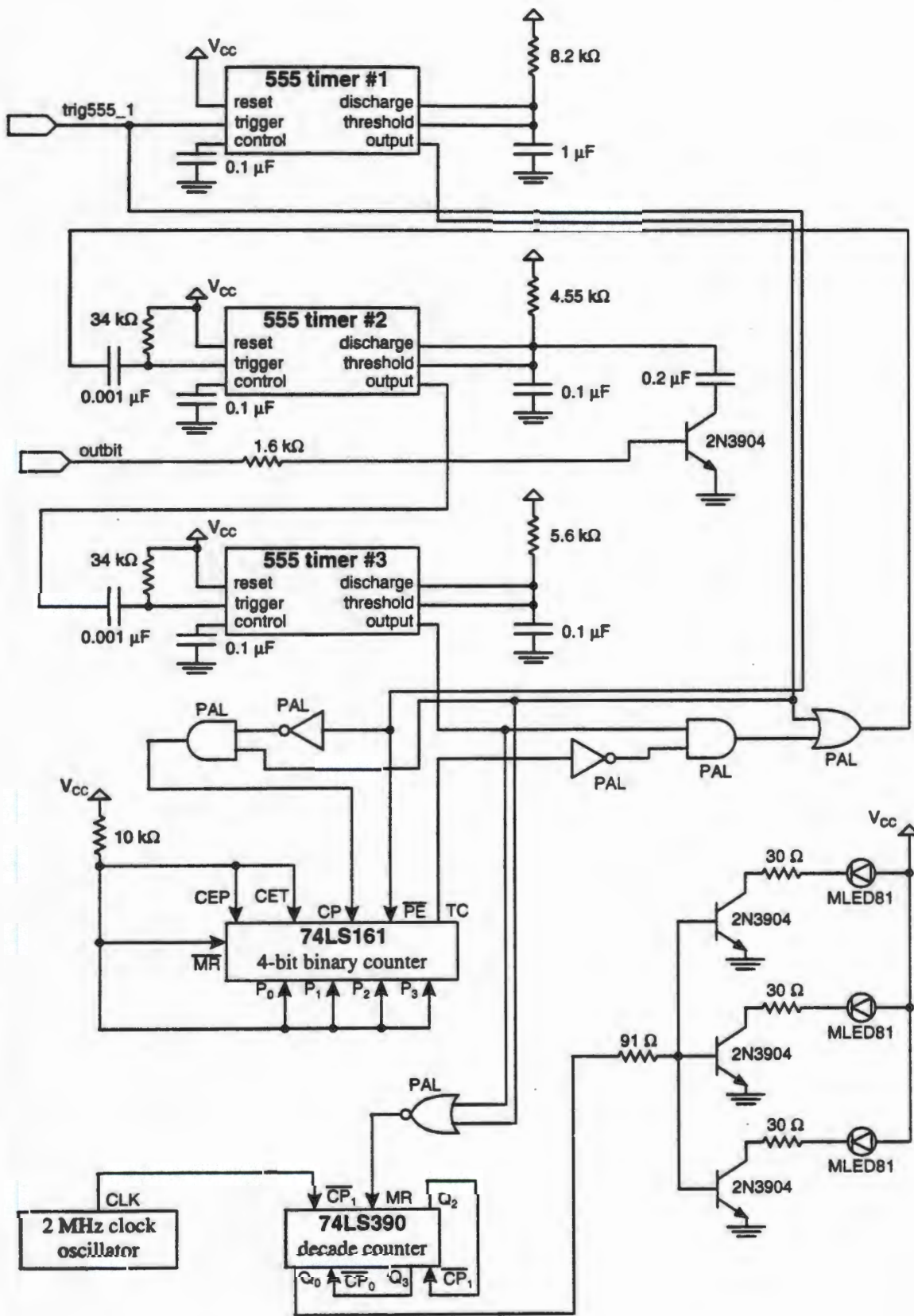


Figure 5.2 (cont.). Circuit Diagram for IR Transmitting Expansion Card

The Microsoft Windows program which communicated with the expansion card had a simple user interface with nine buttons. The user could click any of these buttons with the mouse to transmit a specific 16-bit value through the infrared transmitting expansion card. The robot would then interpret this data and act appropriately. The source code for this Windows program is given in Appendix C.

## **5.2 Results of Engineering Open House 1995**

Members from all four of the Fall 1994 LegoBot teams were present during the entire Engineering Open House. They took turns demonstrating their LEGO robots for the visitors while Cliff was demonstrated at a different table as part of the same exhibit. Several local elementary and middle school teachers brought their classes to come watch the demonstrations and see our exhibit. Many of them expressed interest in wanting to purchase LEGO kits for their schools so that their students could have similar experiences building simple mechanical devices. Overall, Engineering Open House went very well; our exhibit won first place in the Central Exhibits Category.

## **6. SPRING 1995 LEGOBOT CONTEST: BASKETBALL**

At the time of this writing, the Spring 1995 LegoBot contest, organized by Jonathan Kua, is in progress. This semester, there are eight teams of three ECE 291 students each. The same Vesta SBC88A microcontroller board is being used to control the LEGO robots. However, Nate Myers has designed a new add-on board to interface the motors and sensors to the microcontroller board, and this new board is being used by the teams this semester.

### **6.1 Contest Task**

The game for the LEGO robots to play this semester is a modified version of basketball. The contest specification and rules document was written by Doug Gerwitz, and is contained in Appendix G. The robots have to retrieve ping-pong balls from a ball dispenser in the center of the basketball court, and score points with them in one of three different ways: by placing or rolling the ball into the hole beneath the basketball hoop, by placing the ball through the basketball hoop from within the three-point line, or by shooting the ball through the basketball hoop from outside the three-point line.

John Knapowski and I have built an autonomous LEGO robot, named Sam, which solves part of the task outlined in the specification. The primary purpose of this robot was to determine if the task was indeed reasonable in terms of difficulty and complexity. Since at the time we designed this robot the new hardware was not ready, we used the microcontroller boards from the previous semester. The robot uses one dc motor on each of the left and right wheels for driving and steering. It uses a third dc motor on the catapult launching mechanism.

Once the robot is fed a ball into its launching mechanism, it spins clockwise until the rear of the robot is lined up with a basketball hoop. It uses the phototransistor to detect the light source mounted on the backboard of the hoop. Once it is lined up, it travels straight backward until the reflectance sensors detect the black surface marking the three-point area. Since we placed one reflectance sensor on each corner of the robot, it can go through a simple algorithm to line itself up such that each of the two rear reflectance sensors end up just outside the three-point line. This guarantees that the robot is aiming almost directly towards the basket. At this point, the robot simply turns the launcher motor on to activate its catapult (located at the front of the robot), sending the ping-pong ball through the basket. A microswitch detects when the catapult has gone far enough, at which point the motor is automatically turned off. The source code for Sam is given in Appendix D.

## **6.2 Improvements Over Previous Semester**

Several people have made contributions to the organization of the LegoBot contest this semester: Dennis Culley, Doug Gerwitz, myself, John Knapowski, Jonathan Kua, Matt Merten, and Nate Myers. This team was led and managed by Jonathan Kua, whose Master's thesis will describe this semester's events in more detail. As a team, we have designed the contest, organized the kits, constructed the basketball court, and most importantly designed new hardware to interface the motors and sensors to the Vesta microcontroller.

This add-on board, designed by Nate Myers, boasts several improvements over that of the last semester. For example, by using all CMOS technology, the board consumes less power, which saves on AA batteries. The add-on board has several new features such as power

switches, a reset switch, and a hexadecimal LED display which has proved useful for debugging purposes. Furthermore, since the new hardware is on a printed circuit board, there is no longer a risk of making wiring errors when constructing the boards. A copy of Nate's report, including circuit diagrams and detailed descriptions, can be found in the Advanced Digital Systems Laboratory.

## 7. CONCLUSIONS

*“The best way to make theoretical knowledge that we consider important valuable to a student is to give him or her a chance to put it to use. When we discover the gap of “messiness” that lies between theory and practical applications, we should not ignore it or toss it away as uninteresting, but encourage ourselves and our students to dive in and explore the complexity of putting ideas into practice.” [2]*

-- Dr. Fred G. Martin

One can clearly see that using LEGO robots as an educational tool in ECE 291 has been a success over the past three semesters. The students are gaining first-hand experience with engineering design and troubleshooting. They are learning about microcontrollers and embedded control systems. They are developing their skills in working with a team on a group effort. Most importantly, however, they are enjoying the experience.

### 7.1 Advantages and Limitations

The LEGO robots project is versatile enough that it can be used every semester with a different task for the robots to accomplish or a different game for them to play. Often the projects in traditional laboratory courses remain the same from semester to semester; this can become rather repetitive and uninteresting for the instructors and teaching assistants. However, this is not the case with LEGO robots. Every semester, the designs are guaranteed to be unique and creative, and the final contest will always be an exciting crowd pleaser.

Since the microcontroller board used in the LEGO robots project is based on the Intel 8088 microprocessor, it is code-compatible with Intel x86 based desktop PCs. This provides the advantage that students can write software for the microcontroller using already familiar languages and development tools. Since the use of Intel based PCs is so widespread in today's industries and homes, the students are gaining the skills that engineering companies are looking for in their future employees.

While learning about embedded control and microcontrollers should be a part of every Electrical and Computer Engineering student's background, one must understand that a LEGO robot is a very specific application of embedded control. It should not be implied that every student must participate in this project in order to learn about such systems. In a higher level course such as ECE 291, students should always be allowed to use their creativity to invent their own projects.

## **7.2 Additional Resources**

A page on the World Wide Web has been created to describe the LegoBot contest at the University of Illinois. The URL for this page, which can be viewed with any web browser such as NCSA Mosaic or Netscape, is:

<http://www.ece.uiuc.edu/~ece291/legobot/legobot.html>

Currently this web page contains short MPEG motion video clips, created by Professor Sridhar Iyer, from the Fall 1994 LegoBot contest. These clips show successful LEGO robots sinking the pool balls into the pockets and demonstrating other interesting behaviors.



In addition, a VHS video tape has been made which shows clips from the Fall 1994 LegoBot contest and also from Engineering Open House 1995. The video is approximately 15 minutes in length and may be used in the future as an introduction of the LEGO robots project to new students. Copies were distributed to the participants of the Fall 1994 contest as a souvenir for their hard work.

There have been several books and articles written [3-8] which discuss the use of robots in an engineering design course. These works explain the educational value of tools such as robots and also describe engineering design from a pedagogical standpoint.

### **7.3 Future Considerations**

As described in Chapters 3 and 4, there are many aspects of the LegoBot contest which were less than perfect during the Spring 1994 and Fall 1994 semesters. However, even after these minor wrinkles are ironed out, there is plenty of room for major improvements and enhancements to the whole LEGO robots project. Future undergraduate and graduate students are more than welcome to consider these ideas and implement solutions to these problems. Only then will the LegoBot contest continue to grow and expand and become more sophisticated with each semester.

The Vesta SBC88A microcontroller, while quite suitable for controlling LEGO robots, lacks some important features. For example, it does not currently have battery-backed RAM. This means that every time the power to the microcontroller is shut off or disconnected, the program stored in memory is erased. Thus, the students have to download their program every time they power up their LEGO robot. Also, even if the program did remain in the RAM, there

is currently no way to start execution of the program automatically without having to connect the microcontroller board to a PC and use the C\_thru\_ROM software to begin execution. This problem could potentially be solved by programming a special EPROM, which automatically jumps to the beginning of the user code if a certain switch is depressed upon power-up, but jumps to the beginning of the kernel program if the switch is not depressed.

An interesting tool to provide to the LEGO robot builders might be a LEGO CAD software package. Such a package would allow the user to put together virtual LEGO pieces in the CAD program and simulate how the robot will operate. One could view the robot from different angles, and print out schematics of how to build it. This could also be used, for example, to test gear ratios before actually constructing the robot. Such LEGO CAD software packages do currently exist, though their sources are unknown.

Several types of sensors are available which can be interfaced to the microcontroller for use on a LEGO robot. Some of these sensors are certainly more sophisticated and expensive than others, but nevertheless may be worth investigating. Compasses can be used to detect the direction the robot is facing. Ultrasonic or infrared range finders can be used to determine the distance to the nearest object. CCD cameras can be used in a variety of ways to help the robot discover its surroundings. A voice recognition chip could potentially be used to start the LegoBot contest; the judge could say "1 - 2 - 3 - GO!" and the robots would start automatically. Bend sensors might be of some use also, depending on the task to be accomplished. Another possible enhancement might be to add a speech synthesizer module to a robot which would allow the robot to speak words and phrases during the contest.

Finally, there are several possible enhancements to be made to the robot named Cliff.

This robot was capable of receiving infrared commands from a desktop PC equipped with a special expansion card. The next obvious step would be to allow the robot to communicate back to the PC, letting it know the status of its sensors. In this way, one could write software to run on the PC to completely control the robot and allow it to accomplish a nontrivial task. Furthermore, the PC could track the exact movements of the robot and determine its relative position. This information could be used to plot the path of the robot on the computer monitor and help the robot maneuver through obstacles using motion planning algorithms [9]. This type of setup would even be useful to autonomous robot designers for testing their algorithms before porting them to the microcontroller.

## APPENDIX A. SOURCE CODE FOR "WOODY"

This appendix contains the source code for a robot called Woody. The robot responds to infrared commands sent to it via a Sony television remote control unit. A description of its functionality is given in Section 2.1.

### A.1 Listing of WOODY.C

The following is a listing of the file "WOODY.C" which contains the main program loop and major subroutines for Woody, written in the C programming language.

```
/* WOODY.C                               */
/* Rajeev Goel                             */
/* March 1994                               */
/* Robot car responds to                   */
/* Sony remote control                     */

int      COMMANDSET [] =
        {99, 0b00101011, 0b01111001, 0b01011011, 0b11001010,
         1, 0b01111111, 0b01111011, 0b11111011, 0b11011111,
         2, 0b00111111, 0b01111001, 0b11111011, 0b11001111,
         3, 0b01011111, 0b01111010, 0b11111011, 0b11010111,
         4, 0b00011111, 0b01111000, 0b11111011, 0b11000111,
         5, 0b01101111, 0b01111011, 0b01111011, 0b11011011,
         6, 0b00101111, 0b01111001, 0b01111011, 0b11001011,
         7, 0b01001111, 0b01111010, 0b01111011, 0b11010011,
         8, 0b00001111, 0b01111000, 0b01111011, 0b11000011,
         9, 0b01110111, 0b01111011, 0b10111011, 0b11011101};
        /* the command numbers and 32-bit patterns to match */

int      NUMCOMMANDS = 10;                 /* total # commands in above array */
int      POWERBUTTON = 99;                 /* command # for "Power" button */

void waitforescape () {
    /* Waits until the "Escape" button has been pressed and released. */

    while (!escape_button ());
    while (escape_button ());
}

int getbit (int commandindex, int bitnumber) {
```

```

/* Searches for a particular bit in the above defined COMMANDSET. */
/* "commandindex" is the index of the command to search (0 -> */
/* NUMCOMMANDS - 1). "bitnumber" is the bit number from the left */
/* to check (0 -> 31). Returns 0 if the bit is zero and >0 if the */
/* bit is one. */

```

```

int          bitindex;
int          byteindex;

```

```

byteindex = bitnumber / 8;
bitindex = bitnumber % 8;
return COMMANDSET [commandindex * 5 + byteindex + 1] &
    (0b10000000 >> bitindex);
}

```

```

int remotecommand () {

```

```

/* Waits until the IR detector receives a command, then decodes */
/* the command. Returns the matched command number from the array */
/* COMMANDSET, or -1 if the command could not be matched. */

```

```

int          bitstream [100]; /* array of durations of high */
/* and low periods in IR signal */
int          bitstreamptr; /* pointer to above array */
int          HIGHEST1 = 55; /* the highest duration which */
/* will be considered a logic 1 */
int          LOWEST0 = 60; /* the lowest duration which */
/* will be considered a logic 0 */
int          thistest; /* the result (boolean) of the */
/* current test */
int          testcommand; /* which command are we trying */
/* to match right now */
int          testbit; /* which bit are we testing */

```

```

bitstreamptr = (int) bitstream;
readbitstream (bitstreamptr); /* read IR signal into array */
/* this routine is in WOODY.ASM */

```

```

testcommand = 0;

```

```

while (testcommand < NUMCOMMANDS)

```

```

{ /*test for all known commands */

```

```

    testbit = 0;

```

```

    thistest = 1;

```

```

    while ((testbit <= 31) && (thistest)) { /* test all 32 bits */

```

```

        if (getbit (testcommand, testbit)) {

```

```

            if (bitstream [testbit * 2] >= LOWEST0)

```

```

                thistest = 0;

```

```

        }

```

```

        else {

```

```

            if (bitstream [testbit * 2] <= HIGHEST1)

```

```

                thistest = 0;

```

```

        }

```

```

        testbit++;

```

```

    }

```

```

    if (thistest)

```

```

    return COMMANDSET [testcommand * 5];
    testcommand++;
}
return -1;
}

void calibrateremote (int calibratebutton) {
/* This routine returns values by which one can set the "LOWEST0" */
/* and "HIGHEST1" constants in the "remotecommand" routine above. */
/* "calibratebutton" is the command number of a known command. */

int      bitstream [100]; /* array of durations of high      */
/* and low periods in IR signal */
int      bitstreamptr; /* pointer to above array */
int      highest1; /* of all the logic 1's, the */
/* duration of the longest */
int      lowest0; /* of all the logic 0's, the */
/* duration of the shortest */
int      loop; /* loops through all 32 bits */
int      commandindex; /* where in the array is this */
/* command */
int      checkbit; /* bit from the known pattern */

printf ("Waiting for      command #%d\n", calibratebutton);
bitstreamptr = (int) bitstream;
readbitstream (bitstreamptr); /* read in IR signal into array */
/* this routine is in WOODY.ASM */

highest1 = 0;
lowest0 = 1000;
commandindex = 0;
while (COMMANDSET [commandindex * 5] != calibratebutton)
    commandindex++;
for (loop = 0 ; loop <= 31 ; loop++) {
    checkbit = getbit (commandindex, loop);
    if (checkbit) {
        if (bitstream [loop * 2] > highest1)
            highest1 = bitstream [loop * 2];
    }
    else {
        if (bitstream [loop * 2] < lowest0)
            lowest0 = bitstream [loop * 2];
    }
}
printf ("Highest 1 = %d Lowest 0 = %d\n", highest1, lowest0);
waitforescape ();
}

void main() {
/* This program reads IR commands from a Sony TV remote control. */
/* The left motor should be plugged into Motor1, the right motor */
/* into Motor2. The IR detector should be plugged into Digital */
/* Input 0. There is also a Servo motor to control the steering. */

```

```

/* Use the number keypad on the remote control (1-9) to control */
/* the direction of motion of the robot car. Press "Power" to */
/* stop all motors, and disable any more commands. The Frob Knob */
/* controls the speed of the car. */

```

```

int      lastcommand = -1;
int      speed;

```

```

printf ("Remote-bot at your command...\n");

```

```

servo_on ();

```

```

servo_deg (90.0);

```

```

led_out0 (1);

```

```

while (lastcommand != POWERBUTTON) {

```

```

    lastcommand = remotecommand ();

```

```

    speed = (frob_knob () / 3) + 15;

```

```

    if (lastcommand == POWERBUTTON) {

```

```

        printf ("Power is off.\n");

```

```

        alloff ();

```

```

        servo_off();

```

```

        led_out0 (0);

```

```

    }

```

```

    if (lastcommand < 0)

```

```

        printf ("Unknown command.\n");

```

```

    if (lastcommand == 1) {

```

```

        printf ("Forward left\n");

```

```

        servo_deg (30.0);

```

```

        motor (1, speed / 2);

```

```

        motor (2, speed);

```

```

    }

```

```

    if (lastcommand == 2) {

```

```

        printf ("Forward straight\n");

```

```

        servo_deg (90.0);

```

```

        motor (1, speed);

```

```

        motor (2, speed);

```

```

    }

```

```

    if (lastcommand == 3) {

```

```

        printf ("Forward right\n");

```

```

        servo_deg (150.0);

```

```

        motor (1, speed);

```

```

        motor (2, speed / 2);

```

```

    }

```

```

    if (lastcommand == 4) {

```

```

        printf ("Stop left\n");

```

```

        servo_deg (30.0);

```

```

        alloff ();

```

```

    }

```

```

    if (lastcommand == 5) {

```

```

        printf ("Stop straight\n");

```

```

        servo_deg (90.0);

```

```

        alloff ();

```

```

    }

```

```

    if (lastcommand == 6) {

```

```

        printf ("Stop right\n");

```

```

        servo_deg (150.0);

```

```

    alloff ();
}
if (lastcommand == 7) {
    printf ("Back left\n");
    servo_deg (30.0);
    motor (1, -speed / 2);
    motor (2, -speed);
}
if (lastcommand == 8) {
    printf ("Back straight\n");
    servo_deg (90.0);
    motor (1, -speed);
    motor (2, -speed);
}
if (lastcommand == 9) {
    printf ("Back right\n");
    servo_deg (150.0);
    motor (1, -speed);
    motor (2, -speed / 2);
}
}
}
}

```

## A.2 Listing of WOODY.ASM

The following is a listing of the file "WOODY.ASM" which contains source code to sample an infrared waveform and store it in memory. It is written in Motorola 68HC11

Assembly language.

```

* WOODY.ASM                */
* Rajeev Goel              */
* March 1994                */
* Robot car responds to Sony */
* remote control           */

PORTA      EQU    $1000      ; address for Port A, 8 digital inputs
BITZERO    EQU    %00000001  ; we want to look at input 0, IR detector

ORG        MAIN_START

transitcount:
    FCB    0                ; # of transitions the incoming signal makes
dummyinteger:
    FDB    0                ; used for temporary storage

* This subroutine waits until a signal is detected by the IR detector */
* and then stores the lengths of the high periods and low periods    */
* in an array. The starting address of the array is given in         */

```



\* Accumulator D. The routine returns the number of transitions made \*/  
 \* by the signal in Accumulator D. \*/

subroutine\_readbitstream:

```

  ADDD #2 ; skip over first two bytes of array
  STD dummyinteger
  LDX dummyinteger ; store starting address into ACCX
  CLR transitcount ; transitcount = 0

```

```

  LDAA PORTA ; read incoming signal into ACCA
  ANDA #BITZERO

```

waitforsignal:

```

  LDAB PORTA ; read incoming signal into ACCB
  ANDB #BITZERO
  CBA ; compare ACCB with ACCA
  BEQ waitforsignal ; repeat until not equal (signal has changed)

```

readabit:

```

  LDY #0 ; ACCY = 0 (duration of the pulse)

```

waitforchange:

```

  INY ; ACCY = ACCY + 1
  CPY #2000 ; compare ACCY with 2000
  BEQ returnfromread ; if ACCY has reached 2000, quit
  LDAA PORTA ; read incoming signal into ACCA
  ANDA #BITZERO
  CBA ; compare ACCB with ACCA
  BEQ waitforchange ; repeat until not equal (end of pulse)
  LDAB PORTA ; read new signal into ACCB
  ANDB #BITZERO
  STY 0,X ; store pulse duration ACCY into address ACCX
  INX ; ACCX = ACCX + 2 since integer = 2 bytes
  INX
  INC transitcount ; transitcount = transitcount + 1

  LDAA transitcount ; compare transitcount with 100
  CMPA #100
  BEQ returnfromread ; if it's reached 100, then quit
  BRA readabit ; otherwise continue reading the signal

```

returnfromread:

```

  LDAB transitcount ; ACCD = transitcount
  CLRA
  RTS ; return to caller

```

## APPENDIX B. SOURCE CODE FOR "NORM"

This appendix contains the source code for a robot called Norm. The robot responds to infrared commands sent to it via a Pioneer CD player remote control unit. A brief description of its functionality is given at the end of Section 4.1. The following is a listing of the file

"NORM.C".

```
/* NORM.C                */
/* Rajeev Goel          */
/* November 1994        */
/* Robot car responds to */
/* Pioneer remote control */

void setmotorspeed (int motor, int speed)
/* This routine sets the speed of a motor to a value between 0 and 255, */
/* with 0 being the fastest.                                           */
{
    int i;

    outp (0x10, speed);
    outp (0x20, motor + 3);
    for (i = 0 ; i < 100 ; i++);
    outp (0x20, 0);
}

void motoroff (int motor)
/* This routine simply turns the specified motor off.                  */
{
    outp ((motor - 1) * 2, 0);
    outp ((motor - 1) * 2 + 1, 0);
}

void motorforward (int motor)
/* This routine simply sets the specified motor running forward.      */
{
    outp ((motor - 1) * 2, 0);
    outp ((motor - 1) * 2 + 1, 1);
}

void motorbackward (int motor)
/* This routine simply sets the specified motor running backward.     */
{
    outp ((motor - 1) * 2, 1);
    outp ((motor - 1) * 2 + 1, 0);
}
```

```

// binary codes for buttons on the remote control unit
#define LEFTFORWARD      "0000000011111111"    // Button "1"
#define FORWARD          "1000000101111111"    // Button "2"
#define RIGHTFORWARD     "1000000001111111"    // Button "3"
#define LEFT              "0100000010111111"    // Button "5"
#define STOP              "1100000100111111"    // Button "6"
#define RIGHT             "1100000000111111"    // Button "7"
#define LEFTBACKWARD     "0010000011011111"    // Button "9"
#define BACKWARD         "0000000111111111"    // Button "0"
#define RIGHTBACKWARD    "0110010110011011"    // Button "Program Memory"

void main(void)
{
    char                command [50];
    unsigned char       irsignal [100];
    unsigned char       c1, c2;
    unsigned char       count, count2;
    unsigned char       edges;

    setmotorspeed (1, 0);    // set left motor to full speed
    setmotorspeed (2, 0);    // set right motor to full speed
    motoroff (1);           // turn left motor off
    motoroff (2);           // turn right motor off

    while (1)
    {
//      printf ("Waiting for signal...\n");

        c1 = inp (0x10);      // wait until IR sensor reading changes
        c2 = c1;
        while (c2==c1)
            c2 = inp (0x10);

        edges = 0;           // initialize variables
        count = 0;

//      read IR data until too many edges have been detected or until
//      IR signal hasn't changed for a long time
        while ((edges < 100) && (count < 255))
        {
            c1 = c2;           // save old value of IR
            count = 0;         // reset counter
            while ((c2==c1) && (count < 255)) // wait until IR changes
            {
                c2 = inp (0x10); // read new value of IR
                count++;         // increment counter
            }
            irsignal [edges] = count; // store time in array
            edges++;           // next position in array
        }

//      printf ("IR signal detected.\n");

        command [0] = 0;     // initialize variables

```

```

count = 0;
count2 = 0;

while (count < edges)           // eliminate all time values
{                               // which are only 1 for this
    if (irsignal [count] > 1)   // considered noise
    {
        irsignal [count2] = irsignal [count];
        count2++;
    }
    count++;
}

edges = count2;                 // initialize variables
count = 37;                     // skip over the first 37 time values ...
count2 = 0;                     // these will all be the same regardless which
                                // button was pressed

while (count < edges)           // decode data stored in the array
{
    if (irsignal [count] < 8)   // if pulse width is less than 8,
        command [count2] = '0'; // the bit is a 0,
    else                          // otherwise
        command [count2] = '1'; // the bit is a 1
    printf ("%3d, ", irsignal [count]);
    count+=2;                     // skip the next value because the
                                // time between bits is irrelevant
    count2++;
}

//
command [count2] = 0;
printf ("%s\n", command);

if (!strcmp (command, LEFTFORWARD)) // forward, turning left
{
    motoroff (1);                 // left motor off
    motorforward (2);            // right motor forward
    printf ("Left Forward.\n");
}

if (!strcmp (command, FORWARD)) // straight forward
{
    motorforward (1);            // left motor forward
    motorforward (2);            // right motor forward
    printf ("Forward.\n");
}

if (!strcmp (command, RIGHTFORWARD)) // forward, turning right
{
    motorforward (1);            // left motor forward
    motoroff (2);                // right motor off
    printf ("Right Forward.\n");
}

```

```

if (!strcmp (command, LEFT))           // spin CCW in place
{
    motorbackward (1);                 // left motor backward
    motorforward (2);                  // right motor forward
    printf ("Left.\n");
}

if (!strcmp (command, STOP))           // stop
{
    motoroff (1);                      // left motor off
    motoroff (2);                      // right motor off
    printf ("Stop.\n");
}

if (!strcmp (command, RIGHT))          // spin CW in place
{
    motorforward (1);                  // left motor forward
    motorbackward (2);                 // right motor backward
    printf ("Right.\n");
}

if (!strcmp (command, LEFTBACKWARD))   // backward, turning left
{
    motoroff (1);                      // left motor off
    motorbackward (2);                 // right motor backward
    printf ("Left backward.\n");
}

if (!strcmp (command, BACKWARD))       // straight backward
{
    motorbackward (1);                 // left motor backward
    motorbackward (2);                 // right motor backward
    printf ("Backward.\n");
}

if (!strcmp (command, RIGHTBACKWARD))  // backward, turning right
{
    motorbackward (1);                 // left motor backward
    motoroff (2);                      // right motor off
    printf ("Right backward.\n");
}
} // go back to beginning and wait for new IR signal

exit(0); // program never actually gets here!
}

```

## APPENDIX C. SOURCE CODE FOR "CLIFF"

This appendix contains listings of files used in the development of a LEGO robot called Cliff. This robot responded to infrared commands sent to it from an expansion card in a PC. Details on the operation of Cliff can be found in Section 5.1.

### C.1 Listing of CLIFF.C

The software interface on the PC was in the form of a Microsoft Windows program written in C. The user could click on the buttons in the window with the mouse to make the robot move in different directions. This program was responsible for communicating with the expansion card to send the appropriate infrared commands to the robot. The following is a listing of the file "CLIFF.C".

```
/* CLIFF.C */
/* Rajeev Goel */
/* March 1995 */
/* Windows program which sends infrared */
/* commands to a LEGO robot, based on */
/* buttons which the user clicks. */

#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include "resource.h"

#define BUTTONWIDTH 32
#define BUTTONHEIGHT 32
#define BUTTONSPACING 10
#define WINDOWWIDTH 250
#define WINDOWHEIGHT 250
#define WINDOWCENTERX ((WINDOWWIDTH / 2) - 1)
#define WINDOWCENTERY ((WINDOWHEIGHT / 2) - 11)
#define LEFTBUTTON (WINDOWCENTERX - (3*BUTTONWIDTH/2) - BUTTONSPACING)
#define TOPBUTTON (WINDOWCENTERY - (3*BUTTONHEIGHT/2) - BUTTONSPACING)
#define MIDDLEBUTTON (WINDOWCENTERX - (BUTTONWIDTH / 2))
#define CENTERBUTTON (WINDOWCENTERY - (BUTTONHEIGHT / 2))
#define RIGHTBUTTON (WINDOWCENTERX + (BUTTONWIDTH / 2) + BUTTONSPACING)
#define BOTTOMBUTTON (WINDOWCENTERY + (BUTTONHEIGHT / 2) + BUTTONSPACING)
```

```

HWND          hWndGlobal;
HWND          hWndForwardStraight;
HWND          hWndBackwardStraight;
HWND          hWndStop;
HWND          hWndStopRight;
HWND          hWndStopLeft;
HWND          hWndForwardRight;
HWND          hWndForwardLeft;
HWND          hWndBackwardLeft;
HWND          hWndBackwardRight;
HINSTANCE     hInstGlobal;

```

```

LRESULT CALLBACK myWndProc (HWND hwnd, UINT uMsg, WPARAM wParam,
LPARAM lParam)

```

```

/* message handler for main application window */
{

```

```

LPDRAWITEMSTRUCT lpdis;
HICON            hIcon;
int              offset;

```

```

switch (uMsg)
{

```

```

    case WM_CREATE:
        break;

```

```

    case WM_DRAWITEM:

```

```

        // draws the appropriate icon on the face of the buttons

```

```

        lpdis = (LPDRAWITEMSTRUCT) lParam;
        if (lpdis->itemState & ODS_SELECTED)
            offset = 1;

```

```

        else
            offset = 0;

```

```

        if (lpdis->hwndItem == hWndForwardStraight)
            hIcon = LoadIcon (hInstGlobal, MAKEINTRESOURCE (FS_ICON1 +
                offset));

```

```

        if (lpdis->hwndItem == hWndForwardLeft)
            hIcon = LoadIcon (hInstGlobal, MAKEINTRESOURCE (FL_ICON1 +
                offset));

```

```

        if (lpdis->hwndItem == hWndForwardRight)
            hIcon = LoadIcon (hInstGlobal, MAKEINTRESOURCE (FR_ICON1 +
                offset));

```

```

        if (lpdis->hwndItem == hWndBackwardStraight)
            hIcon = LoadIcon (hInstGlobal, MAKEINTRESOURCE (BS_ICON1 +
                offset));

```

```

        if (lpdis->hwndItem == hWndBackwardLeft)
            hIcon = LoadIcon (hInstGlobal, MAKEINTRESOURCE (BL_ICON1 +
                offset));

```

```

        if (lpdis->hwndItem == hWndBackwardRight)
            hIcon = LoadIcon (hInstGlobal, MAKEINTRESOURCE (BR_ICON1 +
                offset));

```

```

        if (lpdis->hwndItem == hWndStop)
            hIcon = LoadIcon (hInstGlobal, MAKEINTRESOURCE (SS_ICON1 +

```

```

        offset));
if (lpdis->hwndItem == hWndStopLeft)
    hIcon = LoadIcon (hInstGlobal, MAKEINTRESOURCE (SL_ICON1 +
        offset));
if (lpdis->hwndItem == hWndStopRight)
    hIcon = LoadIcon (hInstGlobal, MAKEINTRESOURCE (SR_ICON1 +
        offset));
DrawIcon (lpdis->hDC, 0, 0, hIcon);
DestroyIcon (hIcon);
return 0;

case WM_COMMAND:
    // if a button is clicked, send an I/O write to expansion card
    if (HIWORD(wParam) == BN_CLICKED) {
        if ((HWND) lParam == hWndForwardStraight)
            outpw (0x300, 0x0050);
        if ((HWND) lParam == hWndBackwardStraight)
            outpw (0x300, 0x00A0);
        if ((HWND) lParam == hWndStop)
            outpw (0x300, 0x0000);
        if ((HWND) lParam == hWndForwardLeft)
            outpw (0x300, 0x0010);
        if ((HWND) lParam == hWndForwardRight)
            outpw (0x300, 0x0040);
        if ((HWND) lParam == hWndBackwardLeft)
            outpw (0x300, 0x0020);
        if ((HWND) lParam == hWndBackwardRight)
            outpw (0x300, 0x0080);
        if ((HWND) lParam == hWndStopLeft)
            outpw (0x300, 0x0090);
        if ((HWND) lParam == hWndStopRight)
            outpw (0x300, 0x0060);
    }
    return 0;

case WM_KEYDOWN:
case WM_CHAR:
    // also allow user to use arrows on the numeric keypad
    switch (wParam)
    {
        case VK_UP:
            outpw (0x300, 0x0050);
            break;
        case VK_DOWN:
            outpw (0x300, 0x00A0);
            break;
        case VK_LEFT:
            outpw (0x300, 0x0090);
            break;
        case VK_RIGHT:
            outpw (0x300, 0x0060);
            break;
        case VK_HOME:
            outpw (0x300, 0x0010);
    }

```



```

        break;
    case VK_END:
        outpw (0x300, 0x0020);
        break;
    case VK_PRIOR:
        outpw (0x300, 0x0040);
        break;
    case VK_NEXT:
        outpw (0x300, 0x0080);
        break;
    }
    return 0;

case WM_CLOSE:
    DestroyWindow (hWndGlobal);
    return 0;

case WM_DESTROY:
    PostQuitMessage (0);
    return 0;
}

return DefWindowProc (hwnd, uMsg, wParam, lParam);
}

void CreateButtons (void)
/* Create nine different button windows within main application window */
{
    hWndForwardStraight = CreateWindow ("BUTTON", "Forward",
        BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE | BS_OWNERDRAW,
        MIDDLEBUTTON, TOPBUTTON, BUTTONWIDTH, BUTTONHEIGHT,
        hWndGlobal, NULL, hInstGlobal, NULL);
    hWndForwardLeft = CreateWindow ("BUTTON", "Forward",
        BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE | BS_OWNERDRAW,
        LEFTBUTTON, TOPBUTTON, BUTTONWIDTH, BUTTONHEIGHT,
        hWndGlobal, NULL, hInstGlobal, NULL);
    hWndForwardRight = CreateWindow ("BUTTON", "Forward",
        BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE | BS_OWNERDRAW,
        RIGHTBUTTON, TOPBUTTON, BUTTONWIDTH, BUTTONHEIGHT,
        hWndGlobal, NULL, hInstGlobal, NULL);
    hWndStop = CreateWindow ("BUTTON", "Forward",
        BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE | BS_OWNERDRAW,
        MIDDLEBUTTON, CENTERBUTTON, BUTTONWIDTH, BUTTONHEIGHT,
        hWndGlobal, NULL, hInstGlobal, NULL);
    hWndBackwardStraight = CreateWindow ("BUTTON", "Forward",
        BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE | BS_OWNERDRAW,
        MIDDLEBUTTON, BOTTOMBUTTON, BUTTONWIDTH, BUTTONHEIGHT,
        hWndGlobal, NULL, hInstGlobal, NULL);
    hWndBackwardLeft = CreateWindow ("BUTTON", "Forward",
        BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE | BS_OWNERDRAW,
        LEFTBUTTON, BOTTOMBUTTON, BUTTONWIDTH, BUTTONHEIGHT,
        hWndGlobal, NULL, hInstGlobal, NULL);
    hWndBackwardRight = CreateWindow ("BUTTON", "Forward",
        BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE | BS_OWNERDRAW,

```

```

    RIGHTBUTTON, BOTTOMBUTTON, BUTTONWIDTH, BUTTONHEIGHT,
    hWndGlobal, NULL, hInstGlobal, NULL);
hWndStopRight = CreateWindow ("BUTTON", "Forward",
    BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE | BS_OWNERDRAW,
    RIGHTBUTTON, CENTERBUTTON, BUTTONWIDTH, BUTTONHEIGHT,
    hWndGlobal, NULL, hInstGlobal, NULL);
hWndStopLeft = CreateWindow ("BUTTON", "Forward",
    BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE | BS_OWNERDRAW,
    LEFTBUTTON, CENTERBUTTON, BUTTONWIDTH, BUTTONHEIGHT,
    hWndGlobal, NULL, hInstGlobal, NULL);
}

void InitInstance (void)
/* Register window class and create main application window. */
{
    WNDCLASS          wndClass;

    wndClass.style = 0;
    wndClass.lpfnWndProc = myWndProc;
    wndClass.cbClsExtra = 0;
    wndClass.cbWndExtra = 0;
    wndClass.hInstance = hInstGlobal;
    wndClass.hIcon = NULL;
    wndClass.hCursor = LoadCursor (NULL, IDC_ARROW);
    wndClass.hbrBackground = GetStockObject (LTGRAY_BRUSH);
    wndClass.lpszMenuName = NULL;
    wndClass.lpszClassName = "IRROBOT";
    RegisterClass (&wndClass);

    hWndGlobal = CreateWindow ("IRROBOT", "Infrared Robot Controller",
        WS_CAPTION | WS_POPUP | WS_SYSMENU | WS_VISIBLE | WS_MINIMIZEBOX,
        CW_USEDEFAULT, CW_USEDEFAULT, WINDOWWIDTH, WINDOWHEIGHT,
        NULL, NULL, hInstGlobal, NULL);

    CreateButtons ();
}

int PASCAL WinMain(HINSTANCE hinstCurrent, HINSTANCE hinstPrevious,
    LPSTR lpszCmdLine, int nCmdShow)
/* contains message loop */
{
    MSG msg;

    hInstGlobal = hinstCurrent;

    InitInstance();

    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg); /* translates virtual key codes */
        DispatchMessage(&msg); /* dispatches message to window */
    }
    return (int) msg.wParam; /* return value of PostQuitMessage */
}

```

## C.2 Listing of CLIFF.ASM

The robot itself was controlled by a M68HC11EVBU microcontroller board. The software running on this microcontroller was responsible for decoding the infrared signal and activating the appropriate motors. The following is a listing of the file "CLIFF.ASM".

```
* CLIFF.ASM                */
* Rajeev Goel              */
* March 1995                */
* HC11 program which decodes infrared */
* signals and drives the robot in one */
* eight different directions. Each */
* command consists of 16 bits. The */
* first 8 bits are the command code and */
* the last 8 bits are the data.      */

PORTA      EQU    $1000      ; 8-bit I/O port A
PORTB      EQU    $1004      ; 8-bit I/O port B (unused)
PORTC      EQU    $1003      ; 8-bit I/O port C (unused)
PORTD      EQU    $1008      ; 6-bit I/O port D
PORTE      EQU    $100A      ; A/D input port E (unused)

EXTDEV     EQU    $00A8
IODEV      EQU    $00A7
HOSTDEV    EQU    $00A9
OPTION     EQU    $1039      ; options register
PACTL      EQU    $1026      ; control for port A
DDRD       EQU    $1009      ; data direction for port D

INIT       EQU    $FFA9      ; subroutine to initialize I/O device
OUTSTR     EQU    $FFC7      ; subroutine to send string to terminal
OUTA       EQU    $FFB8      ; subroutine to send character to terminal
OUTPUT     EQU    $FFAF      ; subroutine to write I/O device
OUT2BS     EQU    $FFC1      ; subroutine to output hex format to term
OUT1BS     EQU    $FFBE      ; subroutine to output hex format to term
OUTCRL     EQU    $FFC4      ; subroutine to send CR/LF to terminal
INCHAR     EQU    $FFCD      ; subroutine to read character from terminal

IRBITS     EQU    %00110000  ; mask for port D where IR sensor is located
MAXPULS    EQU    $500      ; maximum duration allowed for any pulse
THRESH     EQU    $60       ; threshold between a 0 bit and a 1 bit
SETMOTR    EQU    $00       ; incoming command to set motor states

IRCODE     RMB    16
IRCODE2    RMB    16
IRBYTE1    FCB    0
IRBYTE2    FCB    0
```

```

        ORG      $0100

        LDAA    #%10010000    ; set up options and port directions
        STAA    OPTION
        LDAA    #%10000000
        STAA    PACTL
        LDAA    #%00001111
        STAA    DDRD

IRREAD:  JSR     IREDGE        ; wait for IR receiver to detect something
        BEQ     IRREAD
        LDX     #IRCODE        ; initialize memory pointer

        JSR     IREDGE        ; wait for another IR edge
        BEQ     IRREAD        ; if not detected, start over

IRLOOP:  JSR     IREDGE        ; wait for another IR edge
        BEQ     IRREAD        ; if not detected, start over
        STAB   0,X            ; store the pulse width into the array
        INX
        JSR     IREDGE        ; wait for the next edge
        BEQ     IRREAD        ; if not detected, start over
        STAB   0,X            ; store the pulse width into the array
        INX                    ; increment the pointer
        CPX     #IRBYTE1      ; have we received all 16 bits yet?
        BEQ     DECODE        ; if so, decode the data

        JMP     IRLOOP        ; otherwise grab another bit

DECODE:  LDX     #IRCODE        ; initialize array pointer
        CLRB
        CLRA                    ; initialize registers ACCA and ACCB

DECOD2:  LSLD
        XGDY                    ; logical shift left of ACCD
        LDAA   0,X            ; exchange register X with ACCD
        INX                    ; read pulse width into ACCA
        INX                    ; increment pointer to the next bit
        CMPA   #THRESH        ; determine whether bit is a 0 or a 1
        XGDY                    ; exchange back
        BLO    DECOD1        ; if pulse width not less than threshold,
        INCB                    ; then add 1 to the current accumulator

DECOD1:  CPX     #IRBYTE1      ; have we decoded all 16 bits?
        BNE    DECOD2        ; go back and decode the next bit
        STAA   IRBYTE1        ; store decoded word into ACCA and ACCB
        STAB   IRBYTE2

        CMPA   #SETMOTR      ; is it the command to set motor status?
        BNE    PRINT1
        STAB   PORTA          ; store data byte into port A, motor port
        JSR    DELAY          ; delay for about half a second
        JSR    DELAY
        LDAB   #%00000000    ; turn all the motors off again
        STAB   PORTA

```

```

        JMP      IRREAD          ; wait for the next IR command

PRINT1:  LDX      #IRBYTE1      ; output the command to the terminal
        JSR      OUT2BS        ;   in hex format
        JSR      OUTCRL        ; output carriage return, line feed

        LDX      #IRCODE       ; output the individual pulse widths
PRINT2:  JSR      OUT1BS        ;   to the terminal in hex format
        CPX      #IRBYTE1
        BNE     PRINT2
        JMP      IRREAD        ; wait for the next IR command

IREEDGE: LDAA     PORTD         ; store current value of IR sensor
        ANDA     #IRBITS
        LDY     #0             ; reset counter
IREDG1:  LDAB     PORTD         ; wait until IR sensor changes
        ANDB     #IRBITS
        INY
        CPY     #MAXPULS      ; check to see if counter exceeds the
        BEQ     IREDG2        ;   maximum pulse width
        CBA     ; (compare ACCB to ACCA)
        BEQ     IREDG1
        XGDY     ; store counter value into ACCB
IREDG2:  RTS                 ; return from subroutine

DELAY:   LDAA     #$FF         ; delay for 255 x 255 counts
DELAY2:  LDAB     #$FF
DELAY1:  DECB
        BNE     DELAY1
        DECA
        BNE     DELAY2
        RTS                 ; return from subroutine

```

### C.3 Listing of CLIFF.PDS

The expansion card which transmitted the infrared signals contained one PAL to handle all of the combinational logic. This logic is described below in the file "CLIFF.PDS".

```
;PALASM Design Description
```

```

----- Declaration Segment -----
TITLE   Control for IR transmitter/receiver board interface w/ ISA bus
PATTERN
REVISION 1.0
AUTHOR   Rajeev Goel
COMPANY  University of Illinois
DATE     02/08/95

```

CHIP \_irboard1 PALCE22V10

```
----- PIN Declarations -----
;
PIN 1      PEqQ_684_1                ; INPUT
PIN 2      PEqQ_684_2                ; INPUT
PIN 3      IOW_ISA                   ; INPUT
PIN 4      IOR_ISA                   ; INPUT
PIN 5      Output_555_1              ; INPUT
PIN 6      Output_555_2              ; INPUT
PIN 7      Output_555_3              ; INPUT
PIN 8      TC_161                    ; INPUT
PIN 12     GND                       ; INPUT
PIN 24     VCC                       ; INPUT
PIN 23     CP_323                    COMBINATORIAL ; OUTPUT
PIN 22     S1_323                    COMBINATORIAL ; OUTPUT
PIN 21     S0_323                    COMBINATORIAL ; OUTPUT
PIN 20     E_245                     COMBINATORIAL ; OUTPUT
PIN 19     Dir_245                   COMBINATORIAL ; OUTPUT
PIN 18     Trigger_555_1             COMBINATORIAL ; OUTPUT
PIN 17     Trigger_555_2             COMBINATORIAL ; OUTPUT
PIN 16     CP_161                    COMBINATORIAL ; OUTPUT
PIN 15     IR_out                    COMBINATORIAL ; OUTPUT
;
----- Boolean Equation Segment -----
EQUATIONS
CP_323 = (PEqQ_684_1 + PEqQ_684_2 + IOW_ISA) * /Output_555_2
S1_323 = /(PEqQ_684_1 + PEqQ_684_2 + IOW_ISA)
S0_323 = /(PEqQ_684_1 + PEqQ_684_2 + IOW_ISA) + Output_555_2
E_245 = PEqQ_684_1 + PEqQ_684_2
Dir_245 = /IOR_ISA
Trigger_555_1 = PEqQ_684_1 + PEqQ_684_2 + IOW_ISA
Trigger_555_2 = Output_555_1 + (Output_555_3 * /TC_161)
CP_161 = Output_555_2 + (Output_555_1 * /(PEqQ_684_1 + PEqQ_684_2 + IOW_ISA))
IR_out = /(Output_555_1 + Output_555_3)
;
----- Simulation Segment -----
SIMULATION
;
-----
```

## APPENDIX D. SOURCE CODE FOR "SAM"

This appendix contains the source code for a robot called Sam, written by John Knapowski and Rajeev Goel. This robot solved part of the task for the Spring 1995 LegoBot contest. It was able to find a basket, line itself up for a direct shot, and shoot a ping-pong ball into the basket from behind the three-point line. A brief description of the algorithm is given in Section 6.2. The code is written in C and is intended to run on a Vesta SBC88A microcontroller board with the add-on board designed for the Fall 1994 semester. The following is a listing of the file "SAM.C".

```
/* SAM.C */
/* Rajeev Goel & John Knapowski */
/* April 1995 */
/* Robot car shoots a ping-pong */
/* into basket from behind the */
/* three-point line. */

#define OPEN 1 // microswitch state
#define CLOSED 0 // microswitch state
#define FRONT_BUMP 1 // microswitch port
#define ARM_BUMP 2 // microswitch port
#define FRONT_SPIN_BUMP 3 // microswitch port
#define BACK_SPIN_BUMP 4 // microswitch port
#define LEFT_PHOTO 1 // phototransistor port
#define RIGHT_PHOTO 2 // phototransistor port
#define FRONT_RIGHT_REF 1 // reflectance sensor port
#define FRONT_LEFT_REF 2 // reflectance sensor port
#define BACK_RIGHT_REF 3 // reflectance sensor port
#define BACK_LEFT_REF 4 // reflectance sensor port
#define IR_SENSOR 1 // infrared receiver port
#define POWER_DIP 1 // DIP switch number
#define WHICH_HOOP_DIP 2 // DIP switch number

#define SEARCH_IR 1 // modes of operation
#define ALIGN_IR 2
#define SEARCH_HOOP 3
#define ALIGN_HOOP1 4
#define ALIGN_HOOP2 7
#define SHOOT 5
#define GOTO_HOOP 6
#define UNKNOWN 8
```

```

#define      FAST          0          // motor speeds
#define      SLOW          120
#define      MEDIUM      50
#define      REALSLOW     160

#define      PHOTO_THRESHOLD 100
#define      FOUND_LIGHT(sens) (sens>PHOTO_THRESHOLD)
#define      REFL_THRESHOLD 230
#define      MILLISECONDS 20
#define      SEARCH_DELAY  (200 * MILLISECONDS)
#define      ALIGN_DELAY   (1000 * MILLISECONDS)
#define      SHOOT_DELAY   (2000 * MILLISECONDS)
#define      DOWN_ARM_DELAY (700 * MILLISECONDS)
#define      ON_ORANGE(sens) (sens<=REFL_THRESHOLD)
#define      ON_BLACK(sens)  (sens>REFL_THRESHOLD)

void delay (unsigned int howmuch) // busy wait
{
    int delay;

    for (delay=0 ; delay<howmuch ; delay++);
}

void robot_stop () // turn drive motors off
{
    outp (0x00, 0x00);
    outp (0x01, 0x00);
    outp (0x02, 0x00);
    outp (0x03, 0x00);
}

void arm_stop () // turn catapult motor off
{
    outp (0x04, 0x00);
    outp (0x05, 0x00);
}

void arm_up () // raise the catapult quickly
{
    outp (0x04, 0x01);
    outp (0x05, 0x00);
}

void arm_down () // lower the catapult arm
{
    outp (0x04, 0x00);
    outp (0x05, 0x01);
}

void robot_forward () // turn both drive motors spinning forward
{
    outp (0x00, 0x00);
    outp (0x01, 0x01);
    outp (0x02, 0x00);
}

```



```

    outp (0x03, 0x01);
}

void robot_backward ()                // turn both drive motors backward
{
    outp (0x00, 0x01);
    outp (0x01, 0x00);
    outp (0x02, 0x01);
    outp (0x03, 0x00);
}

void robot_spin_right ()             // right motor back, left motor forward
{
    outp (0x00, 0x01);
    outp (0x01, 0x00);
    outp (0x02, 0x00);
    outp (0x03, 0x01);
}

void robot_spin_left ()              // left motor back, right motor forward
{
    outp (0x00, 0x00);
    outp (0x01, 0x01);
    outp (0x02, 0x01);
    outp (0x03, 0x00);
}

void robot_forward_right ()          // left motor forward, right motor stop
{
    outp (0x00, 0x00);
    outp (0x01, 0x00);
    outp (0x02, 0x00);
    outp (0x03, 0x01);
}

void robot_forward_left ()           // right motor forward, left motor stop
{
    outp (0x00, 0x00);
    outp (0x01, 0x01);
    outp (0x02, 0x00);
    outp (0x03, 0x00);
}

void robot_backward_right ()         // left motor backward, right motor stop
{
    outp (0x00, 0x00);
    outp (0x01, 0x00);
    outp (0x02, 0x01);
    outp (0x03, 0x00);
}

void robot_backward_left ()          // right motor backward, left motor stop
{
    outp (0x00, 0x01);
}

```

```

    outp (0x01, 0x00);
    outp (0x02, 0x00);
    outp (0x03, 0x00);
}

void set_robot_speed (unsigned char speed)
/* set the speed of both of the drive motors: 0=fastest, 255=slowest */
{
    int delay;

    outp (0x10, speed);
    outp (0x20, 4);
    for (delay=0 ; delay<10 ; delay++);
    outp (0x20, 5);
    for (delay=0 ; delay<10 ; delay++);
    outp (0x20, 0x00);
}

void set_arm_speed (unsigned char speed)
/* set the speed of the catapult arm: 0=fastest, 255=slowest */
{
    int delay;

    outp (0x10, speed);
    outp (0x20, 6);
    for (delay=0 ; delay<10 ; delay++);
    outp (0x20, 0x00);
}

unsigned char get_bump_sensor (unsigned char which_bump_sensor)
/* read status of a particular microswitch, bump sensor */
{
    return (1 && ((0x08 << which_bump_sensor) & inp (0x10)));
}

unsigned char get_IR_sensor (unsigned char which_IR_sensor)
/* read status of a particular IR receiver module */
{
    return (1 && ((0x01 << (which_IR_sensor)) & inp (0x20)));
}

unsigned char get_reflectance_sensor (unsigned char which_reflectance_sensor)
/* read value of a particular reflectance sensor, 0=reflective, 255=dark */
{
    int delay;

    outp (0x10, which_reflectance_sensor - 1);
    outp (0x30, 0x00);
    for (delay=0 ; delay<1000 ; delay++);
    return inp (0x30);
}

unsigned char get_light_sensor (unsigned char which_light_sensor)
/* read value of phototransistor, 0=dark, 255=light */

```

```

{
    int delay;

    outp (0x10, which_light_sensor+3);
    outp (0x30, 0x00);
    for (delay=0 ; delay<1000 ; delay++);
    return inp (0x30);
}

unsigned char get_DIP_switch (unsigned char which_DIP_switch)
/* read position of a particular DIP switch */
{
    return (1 && ((0x01 << (which_DIP_switch-1)) & inp (0x10)));
}

int main (void)
{
    int mode;
    int light_sensor;
    int right_reflectance_sensor;
    int left_reflectance_sensor;

    mode = SEARCH_HOOP;          // start robot off searching for hoop
    while (1)
    {
        if (get_DIP_switch (POWER_DIP) == OPEN)
        {
            robot_stop ();          // if "power" DIP switch is open,
            mode = SEARCH_HOOP;    // stop the robot, and reset the
            // program.
        }
        else
        {
            switch (mode)
            {
                case SEARCH_HOOP:
                    set_robot_speed (SLOW);
                    robot_spin_right ();          // spin clockwise
                    delay (SEARCH_DELAY);        // delay for short period
                    robot_stop ();              // stop the robot
                    light_sensor = get_light_sensor (LEFT_PHOTO);
                    if (FOUND_LIGHT (light_sensor)) // if light detected,
                        mode = GOTO_HOOP;        // goto next phase
                    break;

                case GOTO_HOOP:
                    set_robot_speed (SLOW);
                    robot_backward ();            // travel straight backward
                    right_reflectance_sensor =
                        get_reflectance_sensor (BACK_RIGHT_REF);
                    if (ON_BLACK (right_reflectance_sensor))
                    {
                        mode = ALIGN_HOOP1;      // (3-point region)
                        // until black surface
                        // is detected.
                    }
                    break;
            }
        }
    }
}

```

```

left_reflectance_sensor =
    get_reflectance_sensor (BACK_LEFT_REF);
if (ON_BLACK (left_reflectance_sensor))
    mode = ALIGN_HOOP1;
break;

case ALIGN_HOOP1:
    set_robot_speed (REALSLOW);
    right_reflectance_sensor =
        get_reflectance_sensor (BACK_RIGHT_REF);
    left_reflectance_sensor =
        get_reflectance_sensor (BACK_LEFT_REF);
    if (ON_BLACK (right_reflectance_sensor) &&
        ON_BLACK (left_reflectance_sensor))
        robot_forward ();
    if (ON_BLACK (right_reflectance_sensor) &&
        ON_ORANGE (left_reflectance_sensor))
        robot_forward_left ();
    if (ON_ORANGE (right_reflectance_sensor) &&
        ON_BLACK (left_reflectance_sensor))
        robot_forward_right ();
    if (ON_ORANGE (right_reflectance_sensor) &&
        ON_ORANGE (left_reflectance_sensor))
        mode = ALIGN_HOOP2;
    break;

case ALIGN_HOOP2:
    set_robot_speed (REALSLOW);
    right_reflectance_sensor =
        get_reflectance_sensor (BACK_RIGHT_REF);
    left_reflectance_sensor =
        get_reflectance_sensor (BACK_LEFT_REF);
    if (ON_BLACK (right_reflectance_sensor) &&
        ON_BLACK (left_reflectance_sensor))
    {
        set_robot_speed (REALSLOW);
        robot_forward ();
        delay (ALIGN_DELAY);
        robot_stop ();
        mode = SHOOT;
    }
    if (ON_BLACK (right_reflectance_sensor) &&
        ON_ORANGE (left_reflectance_sensor))
        robot_backward_right ();
    if (ON_ORANGE (right_reflectance_sensor) &&
        ON_BLACK (left_reflectance_sensor))
        robot_backward_left ();
    if (ON_ORANGE (right_reflectance_sensor) &&
        ON_ORANGE (left_reflectance_sensor))
        robot_backward ();
    break;

case SHOOT:
    delay (SHOOT_DELAY);

```

```
set_arm_speed (FAST);  
arm_up ();  
while (get_bump_sensor (ARM_BUMP));  
arm_stop ();  
mode = UNKNOWN;  
break;
```

```
case UNKNOWN:  
break;
```

```
}  
}  
}  
}
```

## **APPENDIX E. LEGOBOT CONTEST HANDOUT: CAPTURE THE TORCH**

The following is a reformatted version of the handout which was given to the Spring 1994 LegoBot Contest participants. It contains a description of the task, a parts list, and some programming tips. In addition, the participants also received portions of the 6.270 course notes to help them in using the 6.270 microcontroller board and the IC software package.

### **E.1 Introduction**

The goal of this project is to have fun. Hopefully, as a side benefit, everyone involved will also learn about and gain valuable practical experience in embedded control systems and mobile autonomous robots. There will be two teams of two participants each. Each team will design and program a mobile autonomous robot to accomplish the task of the competition, which is described further below. The robots will be constructed primarily from LEGO parts. The embedded control system to be used is the MIT 6.270 rev. 2.21 microcontroller board, using a Motorola M68HC11 microprocessor.

Embedded control refers to a control system (in this case, a microcontroller) which is embedded inside of the physical system which it is designed to control. Thus the microcontroller is programmed to accomplish certain specific tasks, and is not meant to be used as a general-purpose computer such as a desktop PC.

Sometimes it can be a difficult task within itself simply to decide what defines a "mobile autonomous robot." A "robot" must be equipped with sensor(s) to gather information about the environment. Based on continuous input from such sensor(s), the robot proceeds to take actions

in order to accomplish specific tasks. By "mobile," of course, we simply mean that the robot moves. And by "autonomous," we mean that no outside intervention is allowed once the robot begins its task. In other words, the robot must gather all of its information and decide what to do using only devices which are actually a part of the robot itself and which move with the robot. A system in which a desktop PC is sending commands to the motors through a serial cable is an example of a NON-autonomous robot.

## **E.2 The Task**

This semester's "environment" is a 4' x 4' playing field with short walls along the edges. The two teams will initially position their robots at opposite corners of the field. In the center of the field will be a small device containing a light bulb which we will call the "torch." The torch will have a rod coming out of the bottom, and this rod will be stuck into a hole in the center of the field. The task of each robot is to somehow place the torch in the opponent's corner. In general, this will require the robot to accomplish four subgoals:

- 1) Find and move to the center of the field where the torch stands.
- 2) Lift the torch out of its hole.
- 3) Find and move to the opponent's corner.
- 4) Drop the torch.

The task is considered to be accomplished if any part of the torch touches the area of the field defined by the opponent's corner. The team whose robot accomplishes the task first is the winner of that round. The winner of the contest will be determined by the best of three rounds.

Each round will begin with a gunshot. If either of the robots makes any motions before the gunshot, the round will be restarted. The round ends when either one of the robots completes the task or after two minutes have elapsed (whichever comes first). If after two minutes, one of the robots is still in possession of the torch, that robot loses the round. Therefore, it is important to drop the torch within two minutes even if not in the opponent's corner. The robot is considered in possession of the torch if it is in any way touching the torch.

If at the end of two minutes, neither robot is in possession of the torch, yet neither robot has accomplished the task, the winner will be determined based on the most points acquired.

Points will be given as follows:

- +5 points for responding in some way to the gunshot
- +5 points for moving outside of own corner (any part of the robot)
- +1 point for whichever robot is closer to the torch for each inch that it is closer
- +10 points for reaching/touching the torch at least once
- +10 points for lifting the torch out of the hole
- +10 points for reaching the opponent's corner with the torch in possession
- +10 points for aesthetic quality and creativity of robot.

### **E.3 The Playing Field**

A top view diagram of the playing field is shown in Figure E.1. The two corners will be solid white. The rest of the field will be colored in shades of gray as shown. Two IR transmitters emitting at different frequencies will also be placed at opposite corners of the field to aid the robots in determining their location.



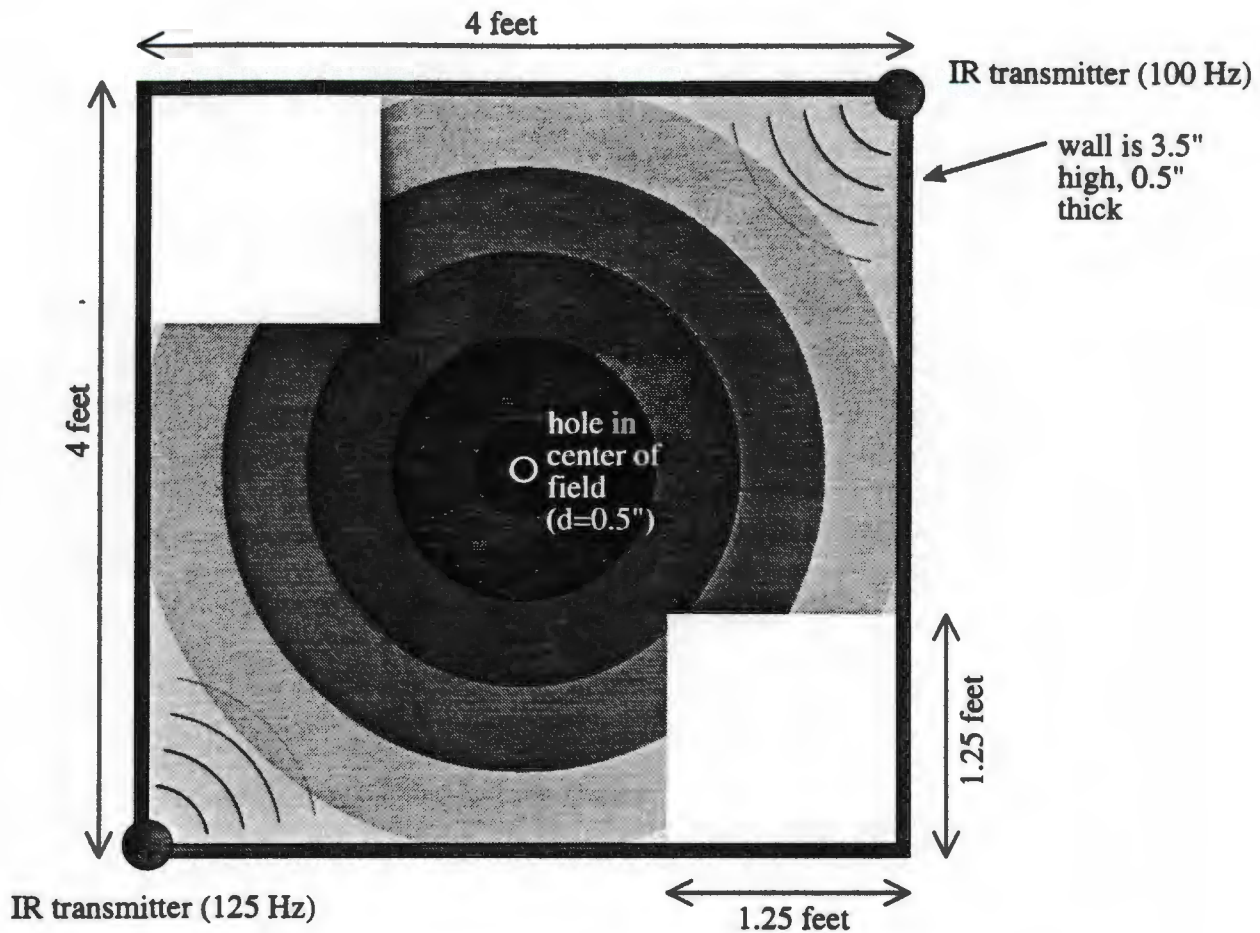


Figure E.1. Top View of Playing Field for "Capture the Torch"

Before the first round begins, a coin will be tossed to determine which corner your robot will start in. Thereafter, the robots will switch corners each round. Once the coin is tossed, you may not make any adjustments to your robot. Your robot will then be placed in its corner at a random orientation such that no part of the robot extends over the boundary of the corner. Therefore, you may not assume in your software that the robot will be initially facing any particular direction.

## E.4 The Torch

A diagram of the torch is shown in Figure E.2. It contains a 6 V, 250 mA light bulb which emits white light in all directions. This will help you locate the torch and also locate your opponent's robot if it is carrying the torch. At the beginning of the round, the torch will be placed standing up in the hole in the center of the field. It will be loose enough to allow your robot to easily pick it up out of the hole, but tight enough to prevent it from coming out simply by pushing on it from the side.

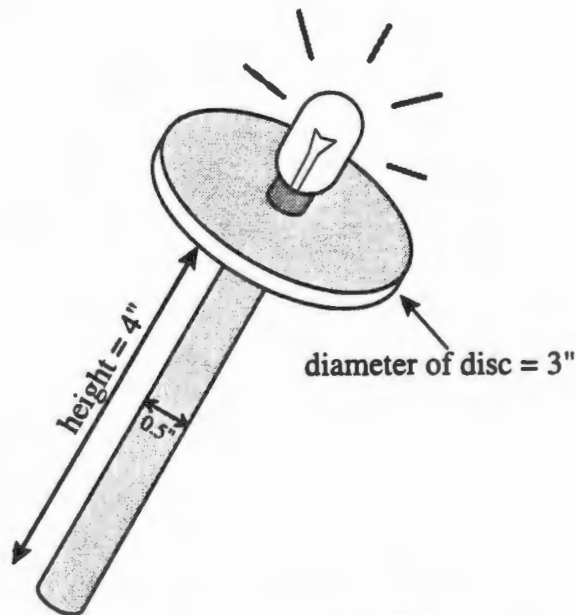


Figure E.2. The Torch

## E.5 Parts List

You will receive the following parts for use in the design of your robot:

- one 6.270 M68HC11 microcontroller board with serial cable
- one 6 V rechargeable battery pack for motors (three 2 V Gates cells)
- one 6 V AA battery pack for microcontroller (four 1.5 V AA cells)
- one manual motor switch board
- assortment of LEGO parts including wheels, gears, and axles
- wire and connectors

- four 4.5 V Polaroid motors
- one Futaba FP-S148 servo motor
- one solenoid
- two GP1U52X IR sensors (good for sensing IR transmissions from corners of field)
- two OPB730F reflectance sensors (ideal for detecting the color of the field surface)
- two MRD370 phototransistors (excellent for detecting light coming from torch)
- four microswitches (wonderful for detecting collisions with wall)
- one Radio Shack 270-092 microphone (superb for detecting sound of gunshot)
- two single-turn, 100 k $\Omega$  potentiometers

In addition, a battery charger will remain in the ECE 291 laboratory for your use. Simply plug your battery pack into the battery charger board, and turn on the power supply. In "Slow" mode, it is okay to leave it unattended, as long as you check it after the expected charge time. In "Fast" mode, keep a close eye on the batteries and unplug them when they become warm to the touch.

If either of the teams would like to use additional part(s) not listed above, it must first get the approval of all organizers of the contest. Then the part(s) will be provided to both teams for use in their designs, so as to eliminate any chance of an unfair disadvantage.

## **E.6 Rules and Restrictions**

- 1.) The size of the robot is restricted as follows. At the beginning of a round, when the robot is being placed in its corner, it must be able to completely fit inside an open-top box which is 11" by 7". There are no restrictions on the height or the weight of the robot.
- 2.) You may use only the above listed parts to construct your robot. Use of additional parts requires prior approval.

## **E.7 The Microcontroller and Software**

The teams will develop their software on IBM PC-compatible machines using IC (Interactive-C) by Fred Martin and Randy Sargent at MIT. This software has been installed on all the PCs in the ECE 291 laboratory. The executables are in the directory "\ic". You will write your code in C and/or 68HC11 assembly. A copy of the 6.270 manual will be given to you for your reference. Chapter 6 is titled "Robot Control" and is an excellent chapter to read before you start writing your software. Chapter 7 explains how to use IC (Interactive C). IC is a subset of the C language, and therefore does not support all of the data structures that ANSI C supports. Therefore, you will need to read Chapter 7 to find out exactly what you can and cannot do with IC. IC does, however, support multitasking and makes it very easy to program the robot to execute many processes "simultaneously."

Here are some relevant specifications about the 6.270 microcontroller board you might find useful:

- eight digital input ports
- 20 analog input ports
- four bi-directional motor output ports
- two digital output ports
- four additional digital outputs OR two additional motor outputs (with braking ability)
- one servo motor output port
- one IR output port
- 32 Kbytes battery-backed RAM for your software
- 2x16 character LCD display
- two push-button switches
- four DIP switches
- one Frob Knob (potentiometer) for use as an analog input

## **E.8 Helpful Tips and Hints**

- 1.) Any decent software should be designed with all constants clearly labeled and defined at the top of the file. Use constants for each port that you use, and label them according to what that port is being used for. Example: "#define leftwheel 3". In this way, the constants can easily be looked up or changed.
- 2.) Comment your code as you proceed.
- 3.) Since the behavior of the sensors may vary and fluctuate slightly depending on the conditions, make sure you have a quick and easy (yet sophisticated and reliable) way of recalibrating your software to account for this.
- 4.) Before you even begin building the robot, try to determine where the microcontroller and the battery packs will sit. The motor battery pack is quite heavy, and will destroy your robot's balance if not placed properly.
- 5.) Before attaching wheels and motors, make sure your geartrain is well-built and smooth. In other words, ensure that the gear teeth mesh together comfortably (not too tight, not too loose). Also make sure there is not too much friction or too much play in the gears.
- 6.) Have fun!

## **E.9 Resources**

The following resources are available for your use if you have questions, comments, suggestions, problems, etc.:

### Participants:

Dan Moore  
Jason Wessel

332-4747  
332-4252

mooredan@uxa.cso.uiuc.edu  
jwessel@uiuc.edu

Michael Landauer	332-4128	mlandaue@ux4.cso.uiuc.edu
Ted Briggs	332-4104	tbriggs@uiuc.edu

**Organizers:**

Rajeev Goel	332-2271	jeev@uiuc.edu
Kevin Safford	328-5448	safford@uiuc.edu

**Professor:**

Professor W. Kent Fuchs	fuchs@crhc.uiuc.edu
-------------------------	---------------------

**MIT Anonymous FTP site: [cherupakha.media.mit.edu](http://cherupakha.media.mit.edu)**

When you FTP to this site as "anonymous," and go into the "/pub/6270" directory, you will find the documentation for the 6.270 board and IC, among other things.

**Robotics Newsgroup: [comp.robotics](http://comp.robotics)**

Type "nn comp.robotics" for a discussion of current problems and discoveries in robotics. Many of the active users of this newsgroup have 6.270 boards and build robots out of LEGO, just like you.

**6.270 Mailing List: [robot-board@oberon.com](mailto:robot-board@oberon.com)**

This mailing list is for users of the 6.270 board and the Miniboard. Fred Martin, the original organizer of the 6.270 course at MIT, frequently responds to questions and problems via this mailing list. To subscribe, send mail to "listserv@oberon.com". The body of your message should be "subscribe robot-board *your name*" and nothing else.

## **APPENDIX F. LEGOBOT CONTEST HANDOUT: EIGHT BALL**

The following is a reformatted version of the handout which was given to the Fall 1994 LegoBot Contest participants. It contains a description of the task, a parts list, and instructions on how to write programs for and download code to the Vesta SBC88A microcontroller.

### **F.1 Introduction**

The goal of this project is to learn about and gain valuable practical experience in embedded control systems and mobile autonomous robots. Your team of three or four members will be given a kit containing various LEGO parts, motors, sensors, and a microcontroller. You will design and program a mobile autonomous robot to accomplish the task described below, and at the end of the semester your robot will compete against robots designed by other teams. The robots will be constructed primarily from LEGO parts.

Embedded control refers to a control system (in this case, a microcontroller), which is embedded inside of the physical system it is designed to control. Thus the microcontroller is programmed to accomplish certain specific tasks, and is not meant to be used as a general purpose computer such as a desktop PC. The embedded control system to be used is the Vesta SBC88A microcontroller board, which uses an Intel 8088 microprocessor. The microcontroller is supplemented with some extra hardware to drive the motors and control their speed. Convenient ports have been added to facilitate the connection of motors and sensors.

A "robot" is a piece of machinery equipped with sensor(s) to gather information about its environment. Based on continuous input from these sensor(s), the robot proceeds to take actions

in order to accomplish specific tasks. By "mobile," of course, we simply mean that the robot moves. By "autonomous," we mean that no outside intervention is allowed once the robot begins its task. In other words, the robot must gather all of its information and decide what to do using only devices which are actually a part of the robot itself and which move with the robot. A system in which a desktop PC is sending commands to the motors through a serial cable is an example of a NON-autonomous robot.

## **F.2 The Task**

This semester's task takes place on a specially designed pool table. Each round of the contest involves a battle between two robots. Before each 3-minute round, one robot will be assigned to the red balls and the other robot to the green balls. The task is for each robot to sink as many of its own balls as possible within the time limit. The robot that has more of its own balls sunk at the end of the round wins that round.

- 1.) Both robots will start out motionless in their designated corners. The beginning of the round will be signaled by the IR transmitters (at each of the six pockets) being turned on. If either robot moves before the signal, it is considered a false start and the round will be restarted. If a robot false starts three times, it automatically loses that round.
- 2.) There may be no human (or other life forms) intervention of any kind once the round has begun.
- 3.) Except in the event of early termination or sudden death overtime, all rounds last exactly 3 minutes. At the end of this time, the robot that has more of its own balls sunk wins the round.



- 4.) A robot may only sink a ball by placing it in one of the six pockets.
- 5.) If a robot sinks its opponent's ball, it is counted as a sink for the opponent.
- 6.) If a ball leaves the perimeter of the table in any way, that ball is out of play and is not considered sunk.
- 7.) If a robot puts the 8-ball out of play at any time, the round ends immediately, and that robot loses.
- 8.) If a robot sinks the 8-ball before all seven of its own balls are sunk, the round ends immediately, and that robot loses.
- 9.) If a robot sinks the 8-ball after all seven of its own balls are sunk, the round ends immediately, and that robot wins.
- 10.) If at the end of 3 minutes, both robots have an equal number of balls sunk, the round will continue without interruption for a maximum of 2 more minutes. This period is called "sudden death overtime." Sudden death overtime ends as soon as any single ball is sunk.
- 11.) In sudden death overtime, the robot whose ball is sunk first wins the round.
- 12.) If the 8-ball is the first ball to get sunk during sudden death overtime, then the robot who sunk it wins if all of his balls have already been sunk, and loses otherwise.
- 13.) If sudden death overtime lasts 2 minutes without any balls being sunk, the winner of the round will be determined by the judges based on creativity and aesthetic appeal of the robots.

- 14.) A maximum of one minute will be allowed in between rounds for teams to make any repairs, should they be necessary. A team whose robot is not ready when the round is about to begin will be automatically disqualified.

### F.3 The Pool Table

A top view diagram of the pool table in its initial configuration is shown below in Figure F.1. The dimensions of the pool table, as shown above, are 8' by 4'. The surface of the table will be green. The 8-ball is painted solid black, with only two small white circles at opposite ends of the ball. The remaining balls are painted solid green or solid red, and contain no markings. Other than in color, the balls will resemble standard billiard balls (same size, same weight).

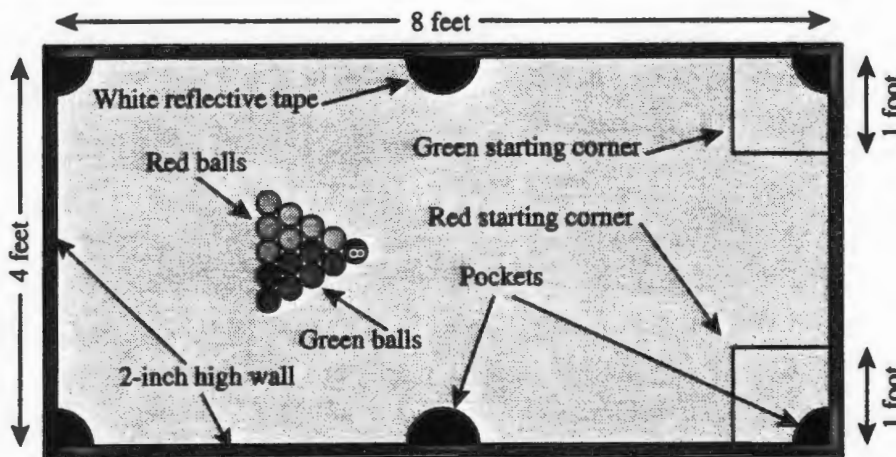


Figure F.1. Top View of Pool Table

At the beginning of each round, a coin will be flipped and a member from one of the two competing teams will be asked to call "heads" or "tails." If he calls correctly, his team will either choose a ball color for their robot, or choose the order in which the two robots will be placed on the board. Since each team can place its robot on the table in any orientation within the starting corner, it may be advantageous for a team to place its robot after the opponents have placed

theirs. The other team will make the remaining decision. If he calls the coin incorrectly, of course, the roles will be reversed.

Each team will place its robot in its designated starting corner in the order established.

No part of the robot may extend beyond the perimeter of the marked square in its starting configuration. The round will be started with a gunshot which the robots must detect using the microphone element.

An IR (infrared) transmitter is mounted at each of the six pockets oriented such that the signal is transmitted toward the center of the board. Each team is given two IR receivers which can detect this signal if oriented correctly.

#### **F.4 Rules and Restrictions**

In order to have a fair and fun contest, and to preserve the life of the robot kits, the teams must adhere to the following restrictions when constructing their robots:

- 1.) Only the parts given in the kit may be used to construct the robot. Use of additional parts requires prior approval from the contest judges. The sum of the costs of all additional parts may not exceed \$10. Save all your receipts; the judges may request to see them.
- 2.) Tape, glue, or any other forms of adhesive may not be used in the construction of your robot.
- 3.) LEGO parts may not be broken or damaged in any way.
- 4.) LEGO parts may not be removed from the motors and sensors to which they have been glued.
- 5.) Any form of intentional permanent damage to the pool table, the balls, or any of the teams kits or robots will not be permitted.

- 6.) Do not leave your kit unattended in any location which is accessible to more than a few people. The parts in the kits are expensive and can be stolen easily.

## F.5 Resources

The following resources are available for your use if you have questions, comments, suggestions, problems, etc.:

### Organizers:

Rajeev Goel	328-8196 or 333-1693	jeev@uiuc.edu
Jonathan Kua	332-4071 or 244-7180	kua@crhc.uiuc.edu

### Professor:

Professor Michael C. Loui

### ECE 291 TA's:

Dennis Culley  
Brandon Long  
Chuck Fuoco  
A. Monte Krol  
Doug Stirrett

### Robotics Newsgroup: comp.robotics

Type "nn comp.robotics" for a discussion of current problems and discoveries in robotics. Many of the active users of this newsgroup have built LEGO robots.

## F.6 Parts List

The following is a complete list of parts contained in each ECE 291 LegoBot kit:

<u>Qty</u>	<u>Part</u>	<u>Supplier</u>
1	LEGO ROBOT KIT I containing...	6270 Technologies <sup>9</sup>
	1 LEGO 9851 (Connectors, Piston Rod)	
	1 LEGO 9852 (Chain Links)	
	2 LEGO 9853 (Gear Set)	

<sup>9</sup> 6270 Technologies is a company started by Pankaj (PK) Oberoi, the purpose of which is to distribute parts used in MIT's 6.270 course. For information contact: 6270 Technologies / One Kendall Square, #312 / P.O. Box 9171 / Cambridge, MA 02139 / (617) 492-5425 / pkoeroi@delphi.com.

	1 LEGO 9854 (Gear Racks)	
	1 LEGO 9855 (Tires, Pulley Wheels)	
	1 LEGO 9856 (Cross Axles)	
	2 LEGO 9857 (Plates)	
	1 LEGO 9858 (Red/Blue Beams)	
	1 LEGO 9862 (Universal Joints)	
	.33 LEGO 9869 (Base Plate)	
	1 LEGO 9871 (Yellow Beams)	
1	SHARPIR containing 4 GP1U52X IR sensors	6270 Technologies
1	ACTUATOR set containing 4 motors and a servo	6270 Technologies
1	GATES set containing three 2V Gates cells	6270 Technologies
4	74LS373 (8-bit latch)	ECE Storeroom <sup>10</sup>
4	74LS684 (8-bit comparator)	ECE Storeroom
1	74F579 (8-bit counter)	ECE Storeroom
1	74LS139 (Dual 4-to-1 multiplexer)	ECE Storeroom
2	74LS04 (Hex inverter)	ECE Storeroom
1	74LS05 (Open-collector hex inverter)	ECE Storeroom
2	74LS02 (Quad NOR gates)	ECE Storeroom
1	74LS390 (Dual decade counter)	ECE Storeroom
2	10 kW resistor packs	ECE Storeroom
2	1 kW resistor packs	ECE Storeroom
4	220 W resistors	ECE Storeroom
1	1 MHz crystal clock	ECE Storeroom
100	strips of 3M connectors	ECE Storeroom
5	strips of male header	ECE Storeroom
5	strips of female header	Digikey <sup>11</sup>
1	4-switch SPST DIP switch	ECE Storeroom
2	small microswitches with rollers	ECE Storeroom
2	large microswitches with rollers	ECE Storeroom
4	OPB730F reflectance sensors	ECE Storeroom
2	MRD370 phototransistors	ECE Storeroom
2	single-turn, 100 kW potentiometers	ECE Storeroom
1	Radio Shack 270-092 microphone	Radio Shack
2	light bulbs	Radio Shack
7	TIP122 NPN power transistors	ECE Storeroom
6	TIP127 PNP power transistors	ECE Storeroom
2	Vector prototyping boards	ECE Storeroom
1	Vesta SBC88A microcontroller board	Vesta Technology, Inc. <sup>12</sup>
1	serial cable	ECE Storeroom
1	battery pack for microcontroller (4 AA batteries)	ECE Storeroom

<sup>10</sup> The ECE Storeroom supplies electronic parts and equipment for the University of Illinois. It is located at 60 Everitt Lab / 1406 W. Green St. / Urbana, IL 61801 / (217) 333-1916 / stores@ece.uiuc.edu.

<sup>11</sup> Digikey / (800) 344-4539.

<sup>12</sup> Vesta Technology, Inc. / 7100 W. 44th Ave., Suite 101 / Wheat Ridge, CO 80033 / (303) 422-8088.

## F.7 Sample Code

It is recommended that you study the following sample C source code to obtain a general idea of how to program the Vesta SBC88A microcontroller. The code in this section has NOT been tested thoroughly and, therefore, probably contains bugs. Note that in C, a "0x" in front of a number simply means that the number is represented in hexadecimal notation.

```
void all_motors_off ()
{
    // This code should always get executed before anything else.
    outp (0x00, 0x01);    // ports 0 and 1 control motor 1
    outp (0x01, 0x01);
    outp (0x02, 0x01);    // ports 2 and 3 control motor 2
    outp (0x03, 0x01);
    outp (0x04, 0x01);    // ports 4 and 5 control motor 3
    outp (0x05, 0x01);
    outp (0x06, 0x00);    // port 6 controls the servo motor
}

void motor_forward (unsigned char whichmotor)
{
    // whichmotor can be either 1, 2, or 3
    outp (whichmotor * 2 - 2, 0x00);
    outp (whichmotor * 2 - 1, 0x01);
}

void motor_backward (unsigned char whichmotor)
{
    // whichmotor can be either 1, 2, or 3
    outp (whichmotor * 2 - 2, 0x01);
    outp (whichmotor * 2 - 1, 0x00);
}

void motor_off (unsigned char whichmotor)
{
    // whichmotor can be either 1, 2, or 3
    outp (whichmotor * 2 - 2, 0x01);
    outp (whichmotor * 2 - 1, 0x01);
}

void set_motor_speed (unsigned char whichmotor, unsigned char speed)
{
    // whichmotor can be either 1, 2, or 3
    // speed is a number from 0 - 255, with 0 being full speed.
}
```

```

    int delay;

    outp (0x10, speed);
    outp (0x20, whichmotor + 3);
    for (delay = 0 ; delay < 10 ; delay++);
    outp (0x20, 0x00);
}

void servo_on ()
{
    outp (0x06, 0x01);    // note this change from the original!
}

void servo_off ()
{
    outp (0x06, 0x00);    // note this change from the original!
}

void set_servo_angle (unsigned char angle)
{
    // angle is a number within the range of 140 - 240. Do
    // not attempt to set an angle outside of this range.

    int delay;

    outp (0x10, angle);
    outp (0x20, 0x07);
    for (delay = 0 ; delay < 10 ; delay++);
    outp (0x20, 0x00);
}

unsigned char get_DIP_switch (unsigned char whichswitch)
{
    // whichswitch can be either 1, 2, 3, or 4
    return (0x01 << (whichswitch - 1)) & inp (0x10);
}

unsigned char get_bump_sensor (unsigned char whichbump)
{
    // whichbump can be either 1, 2, 3, or 4
    return (0x08 << whichbump) & inp (0x10);
}

unsigned char get_IR_sensor (unsigned char whichIR)
{
    // whichIR can be either 1 or 2
    return (0x01 << (whichIR - 1)) & inp (0x20);
}

unsigned char get_reflectance_sensor (unsigned char whichreflectance)
{
    // whichreflectance can be either 1, 2, 3, or 4,
    // accessing A/D channel 0, 1, 2 or 3

```

```

int delay;

outp (0x10, whichreflectance - 1);
outp (0x30, 0x00);
for (delay = 0 ; delay < 10 ; delay++);
    // need to wait 200 microseconds for A/D conversion
return inp (0x30);
}

unsigned char get_light_sensor (unsigned char whichlight)
{
    // whichlight can be either 1 or 2, accessing A/D channel 4 or 5

int delay;

outp (0x10, whichlight + 3);
outp (0x30, 0x00);
for (delay = 0 ; delay < 10 ; delay++);
    // need to wait 200 microseconds for A/D conversion
return inp (0x30);
}

unsigned char get_potentiometer (unsigned char whichpot)
{
    // whichpot can be either 1 or 2, accessing A/D channel 6 or 7

int delay;

outp (0x10, whichpot + 5);
outp (0x30, 0x00);
for (delay = 0 ; delay < 10 ; delay++);
    // need to wait 200 microseconds for A/D conversion
return inp (0x30);
}

```

To compile, link, download, and run a C program called `pool.c`, perform the following steps. It is recommended that you create batch files for yourself so that you do not have to type these lengthy commands every time.

1.) Make sure that your path contains the following directories:

```

C:\C600\BIN
C:\C600\BINB
C:\CTR20\BIN

```

Also make sure that your LIB environment variable contains these directories:

```

C:\C600\LIB
C:\CTR20\LIB

```



There is a program called SETENV.BAT in the C:\CTR20 directory which sets these variables for you.

2.) `cl /zi /Od /c pool.c`

3.) `link st+pool,pool,,sctr_m /map /co /noe;`

The file `st.obj` must be in your current directory for this to work. It has been placed in the C:\CTR20 directory for your convenience.

4.) Apply power to your microcontroller by plugging the AA battery pack into port J7. Also make sure the serial cable is connected between the microcontroller port J6 and the COM1 port of the PC.

5.) `rdeb /BP=nnn /sbc=FFFF /com1 /B4800 pool`

Replace `nnn` with the number of lines of source code in your program. These command line options work for the machines in the ECE 291 laboratory. If you are running on a slower machine such as a 386, try `sbc=2000` instead of `sbc=FFFF`.

6.) If the software does not begin downloading, type:

`load pool`

7.) When it is done downloading, type `g` to run the program on the microcontroller. At this point, you may disconnect the serial cable. Remember that the motor battery pack must be plugged in to drive the motors.

8.) To save on batteries, remove power from the microcontroller whenever it is not in use (i.e., when you are editing your code).

To assemble, link, download, and run an assembly program called `pool.asm`, perform

the same steps as above, except replace steps 2 and 3 with the following:

2.) `masm pool.asm;`

3.) `link pool,pool /map /co /noe;`

## **APPENDIX G. LEGOBOT CONTEST HANDOUT: BASKETBALL**

The following is a reformatted version of the handout which was given to the Spring 1995 LegoBot Contest participants. It contains a description of the task which the LEGO robots had to accomplish. This specification was written by Doug Gerwitz.

### **G.1 Introduction**

With a team of two other ECE 291 students you will learn about embedded real-time control systems by designing and building a robot that will successfully perform the task outlined below. The robots will be constructed from LEGO parts, motors, sensors, and an Intel 8088-based microcontroller. Various sensors are available that acquire information about the robot's environment. The microcontroller has access to environmental information through the hardware input ports which are directly connected to the sensors. Based on this input, a software program stored in the microcontroller's memory will determine the robot's actions. These actions are executed by sending signals from the microcontroller to the robot's motors via hardware output ports. Since the motors require high-current and non-TTL level voltages, the microcontroller is unable to power the motors directly. Therefore, a custom-built board has been provided to drive the motors. Details about this board's operation will appear in additional handouts.

In this contest, the robots are completely autonomous. Robots gather information from their sensors, and control programs use this information when determining a course of action. Therefore, once the contest begins, nobody is allowed to touch their robot or send signals to it via a remote control, serial cable, or by any other means.

## **G.2 The Basketball Game**

The objective in LegoBot Basketball is simply to score more points than the opponent. The scoring system is described in Table G.1. The contest is a series of single-elimination rounds. Each round is a battle between exactly two robots. A round begins with the flip of a coin. If a team calls the coin correctly, they will shoot towards basket #1 during the first half and basket #2 during the second half. A DIP switch will be provided on the microcontroller so that this information can be set before the round begins. The team must ensure that their robot is fully capable of playing either basket.

Each robot is placed on the court so that it is on the same half as its opponent's basket. Any initial orientation is acceptable. The only restriction is that the robot may not be placed inside of the jump circle. Both teams must place the robots at the same time so that neither team has an positional advantage.

Play will begin when the starting gun (a toy cap-gun) is fired. Teams will be provided with a microphone capable of detecting this signal. The first half lasts for two minutes. Since placing the robot on the court might cause the microphone to register a start signal, it might be a good idea to have a DIP switch on the robot that enables/disables the microphone.

The referee signals when each half is over by blowing a whistle. Balls that are still touching the robot when the whistle is blown can not count for points. However, "buzzer-beaters" are allowed. Balls that are no longer touching any part of the robot when the whistle sounds but eventually enter the basket (without external intervention) *do* count.

At half-time, teams are allowed to pick up their robots. Modifications are not permitted on the robot during half-time. Teams are only allowed to change DIP switch settings during half-time. During the second half, each robot will shoot towards the opposite basket. The robots are then placed again on the table (opposite the basket they are shooting at and outside the jump circle) and await the cap-gun that starts the second half. Half-time will last exactly one minute.

Approximately 20% of the balls dispensed are orange "bonus" balls. When sunk, these balls count 3 times the value of a white ball.

Table G.1. Scoring System

<u>Color</u>	<u>"lay-up"</u>	<u>"dunk"</u>	<u>"three-pointer"</u>
white	1	3	5
orange	3	9	15

Balls that are sunk into your opponent's basket count for your opponent. A "lay-up" occurs when a Ping-Pong enters the hole beneath a basket without traveling through the rim. A "dunk" occurs when a robot is inside the three-point arc and the Ping-Pong ball travels through the rim (the rim is 6" above the hole and the hole is at table level.) "Three-pointers" occur when all parts of the robot are behind the three point arc and the Ping-Pong ball travels through the rim.

### **G.3 The Basketball Court**

The robots play on a "basketball court" 4' wide by 8' long (see Figure G.1). As in basketball, there are two baskets located on opposite sides of the court. The rims are 8" in diameter and 6" above the surface of the court. Below each basket there is a hole in the court that balls can fall through. Backboards extend 6" above the rim. The interior of the three-point

arcs are painted black and are located 12" from the center of the rim. The walls surrounding the court will be 2" high.

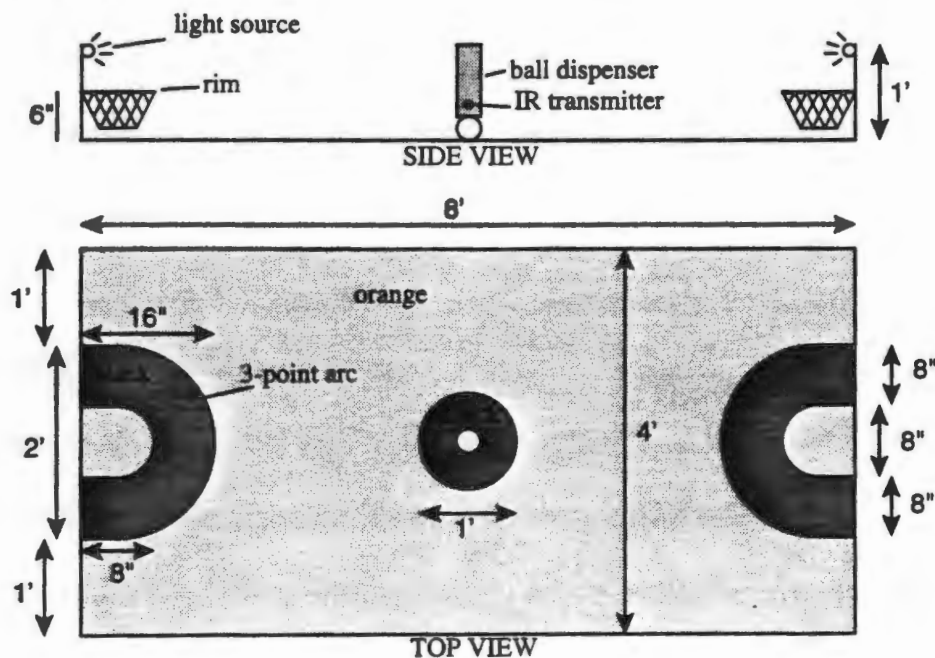


Figure G.1. Top and Side Views of Basketball Court

Each basket has a light source behind it so that robots can find it more easily. In front of each basket's light source, there will be a polarizer. When light shines through two polarizing films, it will be transmitted if the two films are aligned in the same direction. If their alignments are perpendicular, no light is transmitted. This property of polarizing films can be used to differentiate between the baskets.

Basket #1 will have a light source that is polarized 45 degrees clockwise (with respect to the vertical). Basket #2 will have a light source that is polarized 45 degrees counterclockwise. Each team will be given two phototransistors and two polarizing films.

In the center of the court there will be a "basketball dispenser." This is where both robots will find the "basketballs" used in the contest. Basketballs in the contest will be simulated by

The add-on hardware has enough circuitry to support the following input and output devices:

- 3 dc motors
- 1 servo motor
- 2 infrared receivers
- 2 phototransistors (light sensors)
- 4 reflective object sensors
- 2 potentiometers
- 4 DIP switches
- 4 microswitches (bump sensors)

The remainder of this document describes how to use the microcontroller to control the motors and receive input from the sensors.

## **H.2 Connecting Devices to the Microcontroller**

Figure H.1 shows a top-view diagram of the microcontroller board. Use this diagram to plug the motors and sensors into the correct ports. All of the ports are either 2-pin, 3-pin, or 4-pin ports. Ports where one of the pins is highlighted are directional, and one should take care to plug the device in correctly. The highlighted pin denotes pin 1; the devices also have some sort of marking to denote pin 1.

In the early stages of your design, try to place the microcontroller in an easily accessible location on the robot since you will be modifying the connections often. Also try to place the microcontroller as close to the center of the robot as you possibly can so that the length of the wires which reach out to the sensors and motors can be kept at a minimum. Keep in mind that your robot must be large enough and sturdy enough to support the microcontroller, the motor battery pack, and the AA battery pack.

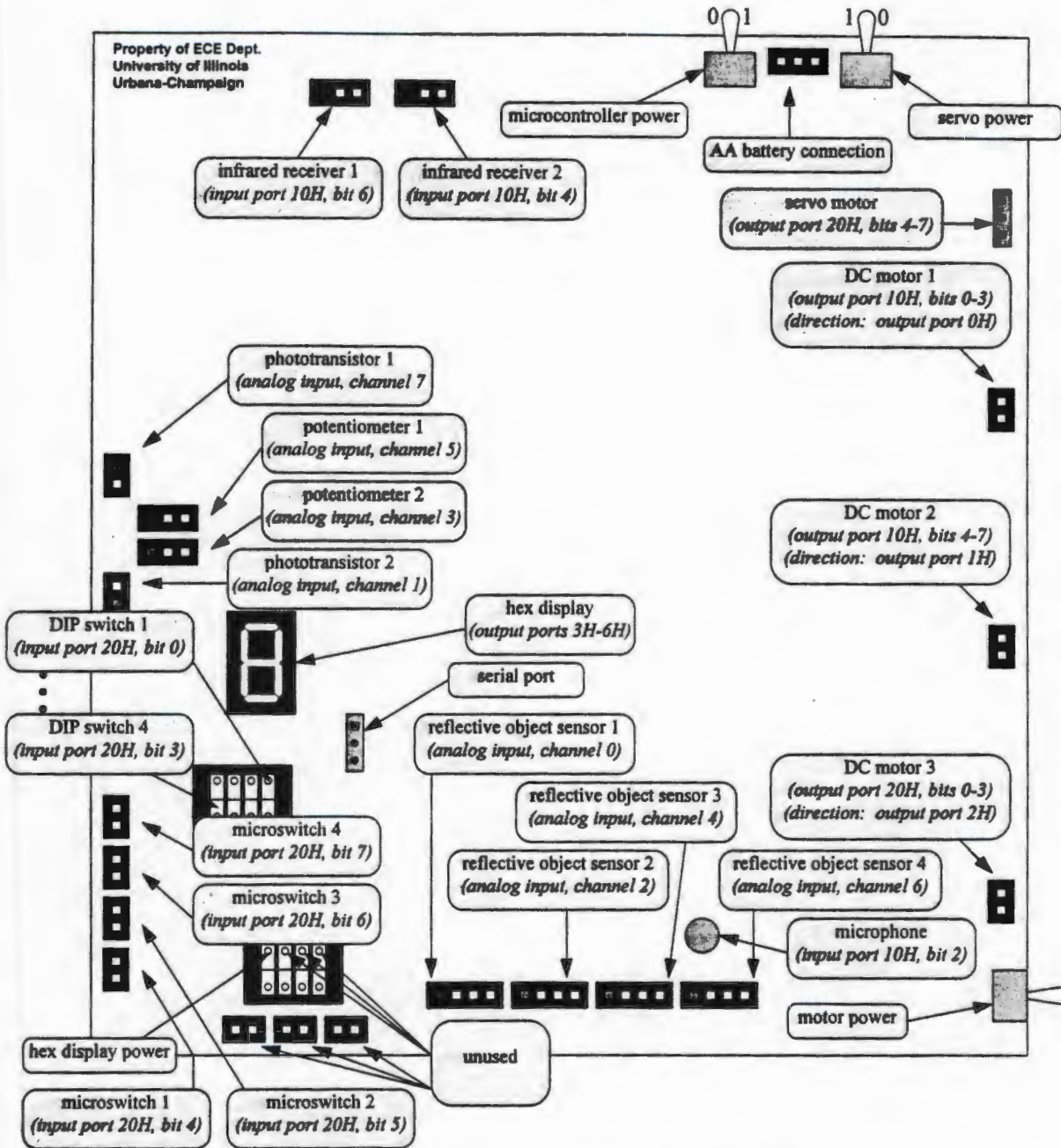


Figure H.1. Microcontroller Port Locations

### H.3 Reading the Value of a Digital Sensor

Digital sensors return either 0 or 1. Each digital sensor is connected directly to a single bit of one input port. To read data in from a digital input port, one should use `inp` in C language

or in Assembly language. Then one must mask out the appropriate bits. When using `inp` in C language, one should data types of `unsigned char`. The following snippets of code demonstrate how to read the current status of microswitch (bump sensor) #3, which is located on port 20H, bit 6.

In C language:

```
unsigned char    bump3;

bump3 = inp (0x20);
    // 0x means hexadecimal notation
if ((bump3 & 0x40) == 0)
    // microswitch #3 is depressed
else
    // microswitch #3 not depressed
```

In Assembly language:

```
IN    AL, 20h
AND   AL, 01000000b
CMP   AL, 0
JNE   MS1
MS0:  ;microswitch depressed
      JMP  EXIT
MS1:  ;not depressed
EXIT: ;...
```

#### H.4 Reading the Value of an Analog Sensor

Analog sensors return a voltage between 0 and 5 V. This voltage is converted by an analog-to-digital (A/D) converter to a byte value between 0 and 255. The Vesta microcontroller has eight analog input channels. Reading the value of an analog is slightly more complicated. We must first write to port 10H to “tell” the microcontroller which channel we wish to read from (only the least 3 significant bits are important). Then we must write a 0 to port 30H to initiate the conversion. Approximately 200 microseconds later, the digital byte can be read from port 30H. To write a value to a port, one should use `outp` in C language or `out` in Assembly language. When using `outp` in C language, one should data types of `unsigned char`. Notice that when one writes to port 10H, the motor speeds of dc motors 1 and 2 will be affected. So it will be important to set output port 10H back to its original value after the analog read is complete. The following snippets of code demonstrate how to read the current position of potentiometer #2, located on analog channel 3.



#### In C language:

```
unsigned char    pot2;
int              delay;

outp (0x10, 3);
outp (0x30, 0);
for (delay = 0 ; delay < 10 ;
     delay++); // short delay
pot2 = inp (0x30);
// pot2 now has a value between 0 and
// 255 depending on the position of
// potentiometer #2.
```

#### In Assembly language:

```
MOV    AL, 3
OUT    10h, AL
MOV    AL, 0
OUT    30h, AL
MOV    CX, 30
DELAY: LOOP DELAY
IN     AL, 30h
; register AL now has a value
; between 0 and 255 depending
; on the position of
; potentiometer #2.
```

## H.5 Writing to the Motor Ports

Each dc motor is controlled by a combination of two ports. One of the ports is used to control the speed of the motor, and the other is used to control the direction of the motor. The speed of each dc motor is a 4-bit value, with 0 being the slowest speed (essentially off) and 15 being the fastest speed. The direction is a single bit value, with 0 being forward and 1 being reverse. The following snippets of code demonstrate how to set the speed of dc motor 1 to about 50% (speed value = 7) and set it running forward. The speed of motor 1 is located on port 10H, bits 0-3. The direction of motor 1 is located on port 0H.

#### In C language:

```
outp (0x10, 0x07);
// notice that we have simultaneously
// set dc motor 2 (which is located at
// port 10H, bits 4-7) to a speed of 0.
outp (0x00, 0); // motor forward
```

#### In Assembly language:

```
MOV    AL, 07h
OUT    10h, AL
MOV    AL, 0
OUT    00h, AL
```

The servo motor is controlled by a 4-bit value which is directly proportional to the desired angle. Using this microcontroller, the servo motor has a total of approximately 90 degrees of rotation. One cannot control the speed of the servo motor; it moves at a fixed low speed with high torque. The following snippets of code demonstrate how to set the servo motor

to roughly 45-degrees. The servo will move there slowly and stop automatically when it gets there. The servo is located on port 20H, bits 4-7.

In C language:

```
outp (0x20, 0x70);  
// notice that we have simultaneously  
// set dc motor 3 (which is located at  
// port 20H, bits 0-3) to a speed of 0.
```

In Assembly language:

```
MOV AL, 70h  
OUT 20h, AL
```

## H.6 Downloading Your Program to the Microcontroller

Once you have written your program in C language or Assembly language, you will need to compile it or assemble it into an executable file. This file gets downloaded to the RAM in the microcontroller through a serial cable using a program called RDEB. A step-by-step process is given below. It is recommended that you create batch files for steps 2 and 3 as you will be performing these two steps frequently.

1.) Make sure that your PATH environment variable contains the following directories:

```
C:\C600\BIN  
C:\C600\BINB  
C:\CTR20\BIN
```

Also make sure that your LIB environment variable contains these directories:

```
C:\C600\LIB  
C:\CTR20\LIB
```

There is a program called SETENV.BAT in the C:\CTR20 directory which sets these variables for you.

2.) If you have written a C language program called mp6.c, compile and link it:

```
cl /Zi /Od /c mp6.c  
link st+mp6,mp6,,sctr_m /map /co /noe;
```

The file `st.obj` must be in the current directory for this to work. It has been placed in the `C:\CTR20` directory for your convenience.

If you have written an Assembly language program called `mp6.asm`, assemble and link it:

```
masm mp6.asm;
```

- 3.) Turn on the power for the microcontroller (reset it if power was already on), and make sure the serial cable is connected between the microcontroller serial port and the COM1 port of the PC. If you are using an HP Vectra 386 machine, run the RDEB software by typing the following command:

```
rdeb /BP=0 /sbc=2000 /com1 /B4800 mp6
```

If you are using a Pentium machine, type the following command:

```
rdeb /BP=0 /sbc=FFFF /com1 /B4800 mp6
```

- 4.) Begin downloading your code by typing the following command within RDEB:

```
load mp6
```

- 5.) When the code is done downloading, type `g` to run the program on the microcontroller. At this point, you may disconnect the serial cable unless your program uses the serial port to communicate with the PC (i.e., `printf`). To quit out of RDEB, press `<Ctrl-Break>` and type `q`.

- 6.) To save on batteries, turn the microcontroller off whenever it is not in use (i.e., when you are editing your code).

## H.7 Serial Communication

Since the robots you are building must be entirely autonomous, there is actually no need for any kind of serial communication once the program is running on the robot's microcontroller.

However, for debugging purposes, it might be useful to be able to communicate with the PC (assuming you leave the serial cable still connected even after you are done downloading your program). There are a few ways to communicate with the PC:

1.) If you are writing your program in C language, then you may use the standard `printf` subroutine to send output to the computer monitor, even if the code is running on the microcontroller. The reason this works is that the standard `printf` routine is replaced by another routine which sends the output through the serial port of the Vesta to the PC. As long as RDEB is still running on the PC, the output will appear on the monitor. The following snippet of C code demonstrates how one might use `printf`:

```
unsigned char      reflectivel;

while (1)
{
    reflectivel = get_reflective_sensor (1);
    // This subroutine would, of course, have to be written.
    printf ("The reflective value is %d\n", reflectivel);
}
```

2.) Another way to send data out from the Vesta to the PC is by using interrupt 10H. This technique will not work with RDEB running on the PC. Therefore, after you have downloaded your program to the microcontroller, you must exit RDEB and run a terminal program such as PROCOMM or TELIX or another program which can receive data from the serial port. However, if the serial cable is still connected when you try to quit RDEB, then RDEB will try to "kill" the program running on the microcontroller. So here are the steps you must follow: connect the serial cable, run RDEB, download your code, type `g`, disconnect the serial cable, quit RDEB (`<Ctrl-Break>` and `q`), run PROCOMM, and reconnect the serial cable. The program running on the PC (i.e., PROCOMM) should be set to

send/receive at 4800 baud. The following snippets of code demonstrate how to send the character '\*' through the serial port from the Vesta to the PC using interrupt 10H.

In C language:

```
union REGS      regs;

regs.h.al = (unsigned char) '*';
regs.h.ah = 14;
int86 (0x10, &regs, &regs);
```

In Assembly language:

```
MOV  AL, '*'
MOV  AH, 14
INT  10h
```

3.) Finally, one can also receive characters from the PC keyboard by using interrupt 16H.

Again, this will not work with RDEB running; there must be a terminal program or other software running on the PC which can transmit characters through the serial port at 4800 baud. The following snippets of code demonstrate how to read in a character from the PC keyboard:

In C language:

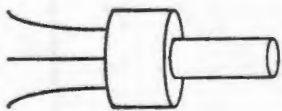
```
union REGS      regs;

regs.h.ah = 0;
int86 (0x16, &regs, &regs);
// regs.h.al now contains the
// ASCII code of the key pressed
if (regs.h.al == 'F')
    robot_forward ();
if (regs.h.al == 'B')
    robot_backward ();
```

In Assembly language:

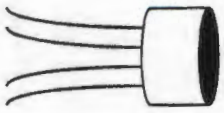
```
MOV  AH, 0
INT  16h ; AL now contains
         ; the ASCII code
         ; of key pressed
CMP  AL, 'F'
JE   FORWRD
CMP  AL, 'B'
JE   BAKWRD
```

## H.8 The Sensors



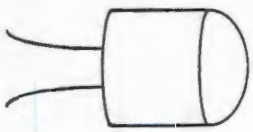
A potentiometer is used to measure rotation. The shaft of the potentiometer can be interfaced directly to a gear on your robot to perhaps measure the position of an arm. It is an analog sensor which will return a

value of 0 at one extreme and 255 at the other. The microcontroller has two potentiometer ports which can be read from analog input channels 3 and 5.



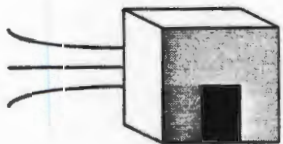
The reflective object sensor, OPB730F, can measure how reflective a surface is in the infrared range. For example, you could use this sensor to detect when your robot is over a black surface as opposed to an orange surface since the

orange surface reflects more. This sensor is an analog sensor which returns approximately 45 for complete reflection and 255 for zero reflection. In order for the sensor to work it must be placed approximately 1/16" to 1/8" from the surface you are trying to measure. The distance from the surface WILL affect the reading, so the separation must be made consistent. The reflective object sensors are mounted on red and blue LEGO pieces. The microcontroller has four reflective object sensor ports which can be read from analog input channels 0, 2, 4, and 6.



The phototransistors, MRD370, can measure the intensity of visible light reaching it. They are highly directional (they must be pointed directly at the light source to detect it). One possible use for the phototransistor is to place

a light bulb (which is always lit) directly across from it, and use the sensor to detect when an object (e.g., a ping-pong ball) has come in the way. The phototransistor is an analog sensor which reads 0 when no light reaches it and higher values (up to 255) depending on the amount of light reaching it. The microcontroller has two phototransistor ports which can be read from analog ports 1 and 7.



The infrared receivers detect light only in the infrared range at 40 kHz.

They can receive from a wide range of directions, and so they may need to be shielded with some type of a tunnel-shaped device around it. They are

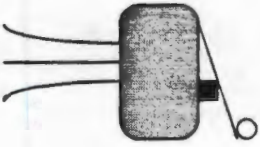
digital sensors which return 1 when not receiving IR and 0 when receiving IR. The

microcontroller has two IR receiver ports which can be read from digital port 10H, bits 4 and 6.



The DIP switches, which are mounted directly on your microcontroller, can be used to set options or modes for your robot. They return digital values of 0 when closed and 1 when open. The microcontroller has four DIP switches which you can use as inputs.

They can be read from digital port 20H, bits 0, 1, 2, and 3.



The microswitches, or bump sensors, are simply low-force switches which are excellent for detecting collisions with obstacles. They return digital values of 0 when depressed and 1 when depressed. The microcontroller has

four microswitch ports which can be read from digital port 20H, bits 4, 5, 6, and 7.



The hexadecimal display is mounted directly on the microcontroller, and can be used for debugging purposes or however you see fit. It can display hexadecimal values from 0 to

F. To display a hexadecimal number, you must do four separate writes to ports 3H, 4H,

5H, and 6H. The following table summarizes how this works:

Port 6H	Port 5H	Port 4H	Port 3H	Hex Display
1	1	1	1	0
1	1	1	0	1
1	1	0	1	2
1	1	0	0	3
1	0	1	1	4
1	0	1	0	5
1	0	0	1	6
1	0	0	0	7
0	1	1	1	8
0	1	1	0	9
0	1	0	1	A
0	1	0	0	B
0	0	1	1	C
0	0	1	0	D
0	0	0	1	E
0	0	0	0	F

To save on batteries, you may turn the hexadecimal display off when you are not using it by flipping one of the DIP switches located on the microcontroller (see Fig. 1).

## **H.9 The Motors**

The microcontroller supports three dc motors and can drive them forward or backward at any of sixteen different speeds. A speed of 0 will essentially turn the motor off, and a speed of 15 will turn the motor on at its top speed.

The microcontroller supports one high-torque servo motor. A servo motor can be commanded to go to any angle (within its range). It will start moving towards that position at a fixed speed and will stop automatically when it gets there. Setting the servo port to 0 will send it to one extreme, and setting it to 15 will send the servo to the other extreme. The two extremes are approximately 90 degrees apart.

## **H.10 Resources**

The following resources are available for your use if you have questions, comments, suggestions, or problems. They are also quite useful if you would like to explore certain topics in greater detail.

### Spring 1995 LegoBot staff:

Dennis Culley, Doug Gerwitz, Rajeev Goel, Matt Merten, Nate Myers, John Knapowski, Jonathan Kua

### ECE 291 TA's:

Dennis Culley, Joseph Gebis, John Knapowski, Brandon Long, Nate Myers, Doug Stirrett

### ECE 291 Professor:

Professor W. Kent Fuchs



**Robotics Newsgroup: *comp.robotics***

Type "nn comp.robotics" from your EWS account for a discussion of current problems and discoveries in robotics. Many of the active users of this newsgroup have built LEGO robots.

**Robotics Mailing List: *robot-board@oberon.com***

To subscribe to this mailing list, send e-mail to "listserv@oberon.com" with a subject of "subscribe robot-board *your\_name*". Discussions on this mailing list range from topics such as LEGO robots to different types of sophisticated sensors.

**MIT 6.270 Course Manual**

The 1992 and 1994 versions of MIT's 6.270 course manual [10] are located in the ECE 291 laboratory. This is the course at MIT which has inspired many universities (including U of I now) to develop courses where students design LEGO robots. The course manual contains oodles of useful information.

**Mobile Robots -- Inspiration to Implementation**

This book, by Joseph L. Jones and Anita M. Flynn, is also located in the ECE 291 laboratory, and is a very well-written book. It contains examples of robots, many of which are built from LEGO parts [11].

**Robot-Building Lab and Contest at the 1993 National AI Conference**

This is an article by Carl Kadie which describes his experiences when he participated in a LEGO robot contest sponsored by AAAI [12].

**Vesta Hardware Manual**

The beige binder contains more detailed information about the Vesta SBC88A microcontroller you are using. It is actually rather poorly written and somewhat cryptic in nature, but might still be of some use with enough patience.

**C-Thru-ROM User's Manual**

This manual contains information on how to use RDEB, the software we use to download programs to the microcontroller. It also supposedly allows you to debug the code while it is running on the microcontroller, but this feature has never been exploited yet.

## LIST OF REFERENCES

- [1] Fred G. Martin. *The 6.270 Robot Builder's Guide*, MIT Media Laboratory, Cambridge, MA, 1992.
- [2] Fred G. Martin. *Circuits to Control: Learning Engineering by Designing LEGO Robots*. Cambridge, MA: Massachusetts Institute of Technology. 1994.
- [3] E. W. Banios. "Teaching engineering practices." *Proceedings of the Frontiers in Education conference*, pp. 161-168. I.E.E.E. and A.S.E.E. 1991.
- [4] E. S. Ferguson. *Engineering and the Mind's Eye*. Cambridge, MA: The MIT Press. 1992.
- [5] W. C. Flowers. "On engineering students' creativity and academia." *ASEE Conference Proceedings*. 1987.
- [6] J. B. Jones. "Design at the frontier of engineering education." *Proceedings of the Frontiers in Education conference*, pp. 107-111. I.E.E.E. and A.S.E.E. 1991.
- [7] Fred G. Martin. "Building robots to learn design and engineering." *Proceedings of the Frontiers in Education conference*. I.E.E.E. and A.S.E.E. 1992.
- [8] Fred G. Martin and Randy Sargent. "Learning engineering through robotic design." *Proceeding of the Ninth International Conference on Technology and Education*, pp. 1191-1193. 1992.
- [9] J. C. Latombe. *Robot Motion Planning*. Norwell, MA: Kluwer Academic Press. 1991.
- [10] *1994 6.270 LEGO Robot Design Competition Course Notes*, MIT Media Laboratory, Cambridge, MA, 1994.
- [11] Joseph L. Jones and Anita M. Flynn. *Mobile Robots: Inspiration to Implementation*. Wellesley, MA: A K Peters. 1993.
- [12] Carl Kadie. "Robot-building lab and contest at the 1993 National AI Conference." *AI Magazine*, pp. 73-77. 1993.